Mind the Buffer: Dissecting the LSM-Buffer Design Space

Boao Chen*
Brandeis University
boaochen@brandeis.edu

Shubham Kaushik*
Brandeis University
kaushiks@brandeis.edu

Subhadeep Sarkar
Brandeis University
subhadeep@brandeis.edu

ABSTRACT

Log-structured merge (LSM)-trees are widely used as the storage layer data structure in modern NoSQL key-value stores due to their superior ingestion throughput, competitive query performance, and efficient space utilization. To enable fast ingestion, LSM-based storage engines first batch the incoming data in an *in-memory buffer* and then opportunistically write the data to slower secondary storage as a collection of *immutable sorted runs*. We point out that while the data is largely storage-resident, the overall performance of an LSM-based storage engine is critically bottlenecked by the (i) implementation, (ii) tuning, and (iii) size of the in-memory buffer. In fact, even with the same buffer configuration, the performance of an LSM-engine may vary by several orders of magnitude if there is a shift in workload. Choosing the appropriate buffer design, thus, is crucial for performance, but, at the same time, hard, as the LSM-buffer design space is vast and largely unexplored.

In this paper, we evaluate and analyze the performance of LSM-engines with 11 different buffer implementations – including three *hash-hybrid designs* – while varying the tuning and size for each implementation to understand their implications on performance and the associated trade-offs. For each configuration, we further vary the workload composition and distribution, the LSM tuning, and the underlying hardware to quantify their impact on performance. Additionally, we build an `Adapt` buffer that switches implementation *on-the-fly* based on the workload shift. To the best of our knowledge, this is the first comprehensive analysis of LSM-buffer design space. Finally, for practitioners, we present a handbook with 10 key guidelines for choosing the appropriate buffer design and tuning for a given workload and performance target.

PVLDB Reference Format:
Boao Chen, Shubham Kaushik, and Subhadeep Sarkar. Mind the Buffer: Dissecting the LSM-Buffer Design Space. PVLDB, 20(3): 784-797, 2026.
doi:10.14778/3860016.3860040

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SSD-Brandeis/LSMMemoryProfiling.

1 INTRODUCTION

LSM-trees are Everywhere. Log-structured merge (LSM) trees are widely used in the storage layer of modern NoSQL key-value data stores, as they offer (i) high throughput for data ingestion, (ii) competitive query performance, and (iii) efficient utilization of storage space [35, 44, 48]. To support efficient writes, LSM-engines buffer the incoming data in memory and opportunistically write it to slower secondary storage in form of *immutable sorted runs* [54, 55]. Immutability of the runs allows for compact storage of data on storage [29], but it also means that updates and deletes must be applied *out of place*, i.e., by inserting a new version of a key that logically invalidates the older target entries [7, 57]. To facilitate efficient reads, runs are logically organized into exponentially larger levels on storage, limiting the number of probes necessary to locate a key [17, 56]. Due to these benefits, commercial key-value stores, including LevelDB [28] and BigTable [13] at Google, RocksDB [26] at Meta, X-Engine [31] at Alibaba, CockroachDB [14] at Cockroach Labs, and AsterixDB [1], Cassandra [4], HBase [5], and Accumulo [3] at Apache, FoundationDB at Apple [6], Speedb [61] at Redis, SplinterDB [16] at VMWare, and ScyllaDB [60], have widely adopted LSM-engines as their storage.

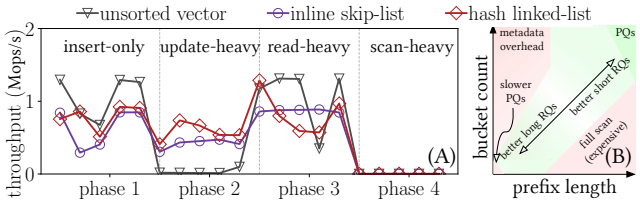


Fig. 1: (A) Dynamic LSM-buffer improves performance by adapting to workload patterns. (B) Hash-hybrid structures provide fine-grained control to tune based on query size.

LSM-Buffer: A Hidden Bottleneck. While the LSM-tree is a storage-optimized data structure [48], any data ingested must first be written to an *in-memory buffer* [54]. Once the buffer reaches capacity, the entries are sorted on key and are written to storage. Batching inserts in memory and opportunistically flushing them to storage allows LSM-trees to (A) avoid frequent accesses to the slower persistent storage and (B) facilitate faster reads on recently ingested data [11, 23]. The **LSM-buffer serves as an entry point** for operations; thus, the **design and efficacy of the buffer significantly impact the overall performance of the LSM-engine**.

Commercial LSM-engines employ a number of different data structures, including vector, skip-list, or hashmaps, to implement the buffer [11, 25, 38]. Each implementation comes with several tunable design knobs, offering very different performance trade-offs. Fig. 1A shows the performance of different buffer implementations under shifting workloads. In *Phases 1 and 3*, the vector-based buffer offers up to 1.7× higher throughput compared to the skip-list- or hash-based buffer. As the workload shifts from insert-only to update-heavy in *Phase 2*, hash linked-list outperforms vector by 24×. In scan-heavy *Phase 4*, due to high I/O overhead, all buffer implementations perform comparably. We highlight that under

*Authors with equal contribution.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 3 ISSN 2150-8097.
doi:10.14778/3860016.3860040

workload shifts, no single buffer consistently offers the best performance. Also, the tuning of each buffer implementation, such as the number of buckets for hash-based buffers or the allocation policy and the size of a vector buffer, significantly impacts performance.

We point out that a suboptimal implementation and tuning of the buffer substantially deteriorates the performance of LSM-engines, effectively acting as a major performance bottleneck.

Problem: The Largely Unexplored LSM-Buffer Design Space. Despite being such a critical component of the LSM-structure, the design space of the LSM-buffer remains largely unexplored. Thus, users typically stick to a storage engine-specific "default" buffer implementation and tuning, resulting in a significant loss in performance, as demonstrated in Fig. 1A. In fact, production systems often offer a limited number of buffer implementations to choose from, and the lack of comprehensive experimental analyses makes it incredibly difficult for users to reason about the appropriate buffer implementation for a given use case [4, 25, 38]. The problem is further exacerbated by the fact that modern application workloads often shift over time [10, 33, 34, 43], and relying on a single buffer implementation leads to suboptimal performance. To this end, we identify the following key challenges.

Challenge 1: Navigating the Buffer Design & Tuning Space. Despite being a critical component, production LSM-engines either (A) resort to default buffer implementation, agnostic to the workload characteristics, losing out on performance [1, 3, 5, 6, 13, 14, 16, 28, 31, 60], or (B) offer a limited number of buffer implementations and tuning knobs to the users, without any usage and tuning guidelines [4, 27, 53, 61]. Fig. 1B shows that the performance of a prefix-hashmap-based buffer varies widely depending on its tuning. A small prefix length with a high bucket count leads to a high metadata overhead, while a long prefix length with a limited bucket count forces expensive full scans. The only work to analyze the buffer design space is tightly coupled to RocksDB and limited to workloads that fit in memory [38]. This limits the scope of the study, as LSM-engines are designed to support applications with larger-than-memory data [48, 59]. The work also does not focus on tuning the buffer or on the impact of workload shifts on the buffer performance, leaving the largest part of the design space unexplored. Finding the appropriate buffer and its tuning, thus, remains a challenging task, even for experienced DBAs, as there is a lack of comprehensive performance analysis as the (i) buffer data structure, (ii) access method, and (iii) tuning are varied.

Challenge 2: Switching Buffer Under Shifting Workloads. Modern application workloads are often *dynamic*, where the workload characteristics shift over time [10, 12, 30, 34, 37, 46, 47]. Thus, relying on a static buffer design, i.e., implementation and tuning, leads to suboptimal performance as the workload changes. For instance, a vector-based buffer offers the best performance for ingestion-heavy workloads as appending incoming entries is quick. However, if the workload sees even a small fraction of point queries, the throughput drops sharply, as every query must (i) create a snapshot of the buffer and (ii) probe the snapshot for the target key, rendering vector as a poor choice for a buffer. An ideal, workload-aware, and performance-conscious LSM-buffer should dynamically adapt to workload shifts and switch to the appropriate buffer implementation and tuning *on the fly*.

Contributions. We dissect the design space of LSM-buffers to unveil its performance implications as we vary (i) the underlying data structure, (ii) the access methods, (iii) the buffer tuning, and (iv) the workload. We evaluate eleven buffer data structures under varied setups and present a handbook on buffer selection and tuning to help practitioners. Our contributions are as follows.

(1) **Evaluating all State-of-the-art Buffer Implementations:** We analyze and evaluate eight buffer data structures, namely (i) *vector*, (ii) *linked-list*, (iii) *skip-list*, (iv) *inline skip-list*, (v) *hash linked-list*, (vi) *hash skip-list*, (vii) *B+ tree*, and (viii) *adaptive radix tree* (*ART*) that are used in commercial LSM-engines.
(2) **Proposing New Efficient Buffer Implementations:** We augment the buffer design space by proposing five new buffer implementations, which include three vector variants: (i) a *sorted vector*, (ii) a *scan-on-query vector*, and (iii) a hash-hybrid: *hash vector*. We also propose a *workload-aware adaptive LSM-buffer* (`Adapt`) that switches implementation subject to workload shifts. We show that the proposed implementations outperform the state of the art under specific conditions.
(3) **Outlining the Performance Trade-offs:** We perform more than 1200 experiments with different buffer types, under varying workload characteristics, to understand their performance implications and the memory vs. performance trade-offs.
(4) **Finding the "Optimal" Tuning:** For each buffer implementation, we vary a number of tuning knobs, such as the allocation policy of vector-variants, the lookahead for skip-list, as well as the buffer size, to determine the optimal tuning. Further, for the hash-hybrids, we vary the *bucket count* and *length of the key prefix* used for hashing to ascertain the optimal setting.
(5) **Answering *What-if* Questions:** Finally, we consolidate our observations in form of 10 key takeaways and compile a handbook with guidelines for choosing and tuning the LSM-buffer "optimally" to help answer *What-if* questions, such as:
   (a) What is the best buffer design for a given workload?
   (b) How to transition to a new buffer design if the workload shifts from write-heavy to read-heavy?
   (c) How do we configure the bucket count for a hash-based buffer given a workload and memory budget?

2 LSM BACKGROUND

LSM Design. The ingestion-optimized design of LSM-trees batches incoming writes in memory before writing them to slower storage as a hierarchical collection of *immutable, sorted runs* across multiple levels [44, 48, 54]. A level may have one or more sorted runs, and each run typically contains multiple files [24, 32, 65]. As we go deeper, the capacity of each level increases by a factor T, which is the *size ratio* of the tree, bounding the number of levels [17, 42].

The LSM-Buffer. The LSM-buffer acts as a transient staging area for incoming data. Batching entries in buffer minimizes accesses to slower storage, facilitating fast ingestion [25]. Once the buffer is full, the entries are sorted on the key, and the buffer is made *immutable*. The immutable buffer is then written opportunistically to the first level on storage during a *flush* in form of a *sorted run*. Note that every operation – writes or reads – in an LSM-engine must begin at the buffer, before traversing to storage. The buffer, thus, acts as a critical bottleneck to a storage engine's performance [38].

Compaction. To ensure competitive query performance and efficient space utilization, the sorted runs on storage are periodically merged to create longer, but overall fewer runs through *compactions* [20, 22, 55, 64]. A compaction reads a target file from Level i, merges it with overlapping files from Level $i+1$, and writes the merged data as new files in Level $i+1$ [2, 22]. This bounds the number of runs probed during a point query and the number of levels in the tree, as any invalid entries are purged during compaction [19].

Basic Operations. LSM-engines support the following operations.

Put: A *put* simply inserts a new entry into the buffer, which later gets flushed to the first level on storage. Note that updates and deletes in LSM-trees are realized *out of place*, i.e., when updating or deleting an entry, instead of modifying the target entry in place, LSM-trees insert a new version of the entry for updates or a special key-value pair, *tombstone* for deletes into the buffer [7, 58, 70]. The newly ingested entry or meta-entry logically invalidates the target entry, which is eventually purged during a future compaction.

Get: A *get* returns the value corresponding to the latest version of a target key, if it exists. The search begins at the buffer and continues from the smallest to the largest level of the tree [18, 56]. For levels with multiple sorted runs, the runs are probed from youngest to oldest [52]. Commercial LSM-engines employ auxiliary metadata structures, such as filters and indexes, to optimize lookups [21, 63, 66, 68]. A search terminates as soon as the key is located.

Scan: A *scan* retrieves all entries within a specified range. During a scan, an LSM-tree initiates iterators on all qualifying sorted runs in the tree to find the qualifying entries and return them in the sorted order. The iterators locate the first key at each run and sequentially read the qualifying entries from that level, while filtering out any invalid entries and merging the data in memory [36, 45, 62, 67].

3 THE LSM-BUFFER DESIGN SPACE

The LSM-buffer serves as the entry point for all operations to the engine. Thus, the implementation and tuning of the buffer affect the engine's performance significantly. This section analyzes the different buffer data structures and access methods in terms of memory and performance. Table 1 lists the analysis parameters.

3.1 Vector

A vector-based buffer implementation allocates a contiguous chunk of memory to store key-value pairs with *minimal metadata overhead* and supports *efficient probes*. A **statically allocated vector**, where the buffer size is pre-allocated, requires the entire memory budget to be reserved upfront, which can lead to inefficient memory usage. However, it offers predictable performance with low tail latency, as memory reallocation is not required. On the other hand, **dynamically allocated vectors** primarily allocate space for a small number of entries (typically, 2) at the beginning – reserving one slot per entry, and then grow geometrically until they reach capacity. Dynamic implementations incur overhead due to (i) periodic memory re-allocation and (ii) copying data to new locations upon resizing, resulting in intermittent latency spikes.

Memory Footprint: Due to their compact layout and minimal metadata overhead, vector-based buffers have the *lowest memory footprint* overall, offering maximum logical buffer capacity. To facilitate efficient access to variable-length entries, instead of storing the actual entry, a *pointer to entry* is stored within the vector itself [25]. In addition to a pointer per key, most commercial LSM-engines store additional metadata, such as a timestamp or sequence number, to maintain the entry order in buffer, which is crucial for updates and deletes logically. The number of entries that would saturate the buffer and trigger a flush is given as: $K^{vec} = \frac{M}{E+p+c}$, where M is the buffer size in bytes, E is the average entry size, p is the pointer size, and c is constant metadata overhead per entry. While a large M minimizes accesses to slower storage, it also increases the compaction granularity [59], and thus, the tail latency for compactions.

Table 1: Parameters used in this paper.

Symbol	Description
M	size of memory buffer in bytes
E	average size of key-value pair in bytes
c	per-entry metadata size (sequence number/timestamp)
N	total number of entries in the LSM-tree
S	total number of scans
s	average selectivity of a scan
T	size ratio of the tree
H	total number of buckets in hash-hybrids
X	length of a prefix from key to hash
b	a branching factor for skip-list
L	number of levels exist on disk holding N entries
l	height of a skip-list buffer (random variable)
l_{max}	maximum skip-list node height
p	size of a pointer

The Vector Variants. While the memory footprints remain unchanged across all vector implementations, the ingestion and query performance vary widely with the access path, offering very diverse performance trade-offs. Below, we present the trade-offs of three vector variants and analyze their suitability for different workloads.

3.1.1 Sort-on-Query Vector (`V-Qsort`).

A sort-on-query vector, `V-Qsort`, is the default vector-based buffer implementation in several production systems, including RocksDB [25] and Speedb [61]. `V-Qsort` is optimized for ingestion-heavy workloads, as incoming inserts (as well as updates and deletes) are simply appended to `V-Qsort` using the *put* API, with a $O(1)$ cost.

Slower Queries: To realize a point query (PQ), `V-Qsort` (i) first takes a snapshot of the buffer containing K^{vec} entries, (ii) sorts the snapshot on the keys, and then (iii) performs a binary search to find the target key. Range queries (RQs) follow the same access path to find the first qualifying key in the snapshot, and sequentially scan the subsequent keys. Operating on a snapshot (i) enables executing multiple queries in parallel and (ii) prevents write stalls during query execution, particularly for long-running RQs. However, this also means that the cost of both point and RQs is dominated by *sorting*, as the binary search and sequential scan are relatively inexpensive. The cost of accessing `V-Qsort` during a point or range query, thus, is asymptotically the same: $O(K^{vec} \log K^{vec})$. Thus, when writes are interleaved with queries, `V-Qsort` performs poorly.

3.1.2 Scan-on-Query Vector (`V-Qscan`).

We propose the scan-on-query vector, `V-Qscan`, with an implementation that is optimized for workloads with interleaved queries and writes. The write path of `V-Qscan` is the same as `V-Qsort`, allowing ingestion in $O(1)$.

Faster Queries: During a PQ, instead of performing an expensive sort, `V-Qscan` performs a *reverse linear scan* on the buffer snapshot to find the target key. This cuts the cost of PQs to $O(K^{vec})$ without any extra overhead. The backward scan ensures query correctness in the presence of updates or deletes; a scan can terminate early as soon as it finds the first match. RQs, however, still require sorting as keys must be returned in order, requiring $O(K^{vec} \log K^{vec})$ time.

3.1.3 Sorted Vector (V-Sorted).

We also propose V-Sorted, a sorted vector-variant, that instead of appending the incoming entries, inserts them in sorted order. By maintaining the entries in sorted order at any point in time, V-Sorted brings down the cost of point and range queries, which makes it conducive to workloads that (i) are RQ-intensive, (ii) have infrequent inserts, or (iii) ingest data with some degree of sortedness [50, 51].

Amortized Writes: Ingestion to the buffer entails performing a binary search to find the correct position for the new entry and shifting the subsequent entries as required. The cost of writes: $O(K^{vec})$, thus, is dominated by shifting the entries post-ingestion and is proportional to the buffer size.

Faster Reads: The additional price paid in the write path, however, improves the query performance significantly. Specifically, RQs can be realized simply by scanning through the *already-sorted* snapshot in $O(K^{vec})$, without the need for an expensive sort. PQs are realized by performing a binary search on the snapshot; however, the cost is dominated by the snapshot creation, which takes $O(K^{vec})$ time.

3.2 Linked-list

A linked-list (Link-L) is a node-based structure that allocates memory dynamically from the heap. Nodes are connected through pointers, with each pointer access incurring one random memory access.

Memory Overhead: Link-L is implemented as a **sorted doubly linked-list**, where each node maintains pointers to its predecessor and successor. These additional pointers add to the memory overhead. The number of entries Link-L can hold before triggering a flush is given by: $K^{link} = \frac{M}{E+2 \cdot p+c}$. Note that every ingestion allocates a node containing two pointers, exacerbating the structural and heap-allocation overhead. This reduces the physical capacity of the buffer, prompting frequent flushes and compactions.

Poor Standalone Performance: A sorted linked-list as a standalone data structure is very slow, as it incurs multiple random memory accesses to find a target key. A linear traversal required for every insert or query costs $O(K^{link})$ random accesses, which makes linked-list unviable for production systems. However, linked-list may be used in a hash-hybrid implementation, as discussed in §3.4.

3.3 Skip-list

A skip-list is a probabilistic data structure that provides logarithmic-time reads and writes by organizing data in a hierarchy of levels.

3.3.1 Classical Skip-list (Skip-L).

Skip-lists arrange data across l levels, with the upper $l-1$ levels acting as express lanes to optimize access. Nodes are allocated dynamically from the heap. Each node points to its successor and to a node in the subsequent level.

Memory Trade-off: The hierarchical layout of Skip-L duplicates key references across multiple levels to enable logarithmic search. The capacity of a Skip-L buffer is affected by this and is given as: $K^{skip} = \frac{M}{E+p \cdot (2+\mathbb{E}[l-1])+c}$, where $\mathbb{E}[l-1]$ estimates the pointer overhead in express lanes. Inserts to buffer entail (i) searching in $O(\log_b K^{skip})$ time to locate the target position – the base of the logarithm is the branching factor b (defaults to 4), (ii) creating a node and updating the pointers, and (iii) promoting a key to a higher level with probability $1/b$, copying pointers from other levels. Production systems often cap the skip-list height to a nominal value, e.g., $l_{max} = 12$ in production RocksDB [27]. Skip-L offers balanced performance for workloads at the cost of memory, but performs poorly compared to V-Qsort and V-Qscan for write-heavy workloads.

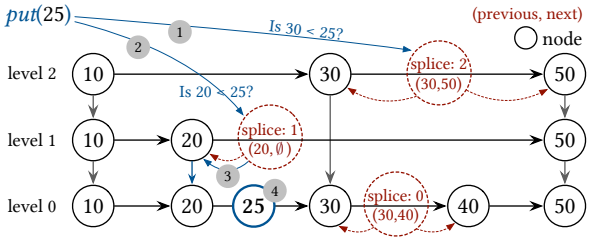


Fig. 2: *Splice* caches per-level recently accessed nodes, enabling faster traversal for temporal-locality access patterns.

3.3.2 Inline Skip-list (InSkip-L).

An inline skip-list, InSkip-L, is highly optimized for temporal-locality-based access patterns and is the default buffer data structure for RocksDB [25].

Optimizes for Temporal Locality: InSkip-L stores each entry *inline* within its node rather than behind a pointer, and caches its most recent search state in a cache, as a *splice*. A *splice* stores pointers to the most recently traversed node's predecessor and successor for each of the $l_{max}+1$ levels along with an integer recording the height up to which the cached state is valid, as shown in Fig. 2. We estimate the memory overhead InSkip-L as: $K^{inskip} = \frac{M-(2 \cdot (l_{max}+2) \cdot p+8)}{E+p \cdot (1+\mathbb{E}[l-1])+c}$, where the *splice* maintained by the buffer takes up $2 \cdot (l_{max}+2) \cdot p+8$ bytes. An insert or query probes the *splice* top-down. If the cached pointers are valid for the target key, they are used to locate the destination in InSkip-L. For instance, in Fig. 2, to insert key 25, we first probe the *splice* at level 2. Since 25 is smaller than the cached key, 30, we proceed to level 1 and use the stored predecessor pointer to continue the search. Once it reaches the destination, we create a new node and update the pointer. If none of the *splice* entries are valid, the operation follows the standard skip-list search path. InSkip-L reduces the expected query cost to $O(\log_b \delta)$, where δ is the distance from the cached position to the destination. *Splices* are updated after each operation based on the traversal path and are recomputed if invalid. For instance, after *put*(25) *splice: 0,* in Fig. 2, is updated to $previous = 25$ and $next = 30$.

Low Memory Footprint: The memory layout of InSkip-L significantly differs from that of Skip-L. Instead of storing a pointer to a key, every node in InSkip-L stores the actual bytes, saving a random memory access. Each node stores: (i) pointers to higher levels, (ii) the level 0 pointer, and (iii) actual bytes of the key in the same cache line. Fig. 2 shows the cache line content of the node storing 10 as: `[pointer to node 10 at level 2 | pointer to node 10 at level 1 | node 10]`. InSkip-L improves cache locality while significantly reducing the memory footprint.

3.4 Prefix-Hash-Hybrids

The hash-hybrid variants use hash functions to cluster entries into *hash buckets*, improving read and write performance [25, 53, 61]. To support RQs, instead of hashing whole keys, production LSM-engines use a fixed-sized key prefix for hashing, as shown in Fig. 3A. The *prefix length* (X) and the *bucket count* (H) are tunable to extract maximum performance for a wide range of workloads.

Design Space of Prefix-Hash-Hybrid Structures: Prefix-hash-hybrid structures construct a continuum of designs by varying X and H, as shown in Fig. 3B. The distribution of keys across buckets depends on the configured value for X and H. For instance, if the bucket count is set to $H = 1$ with any random value of X, all keys will be mapped to the same bucket. Conversely, if the prefix length is set to $X = 1$ with a million buckets, keys will be only mapped to $\mathcal{D}^X$ buckets, where $\mathcal{D}$ denotes the total unique characters in the key domain. For our analysis, we assume ASCII domain, i.e., $\mathcal{D} = 128$. The keys are uniformly distributed in the domain, i.e., each bucket is mapped to $\kappa = \frac{K^i}{\min(H,\ \mathcal{D}^X)}$, where K^i is the capacity of the respective hash-hybrid. Skewed workloads leave the buckets sparsely populated, which still benefits the query cost of hash-hybrids.

Bucket Implementation: The hash buckets are implemented using different data structures, such as vector, linked-list, and skip-list [26, 27]. The cost of a write is the sum of the costs of (i) hashing a key and (ii) locating the correct position within a bucket. The ingestion cost is, thus, dominated by accessing the bucket for the target key. The asymptotic complexity of hash-hybrids is dictated by the data structure of the buckets. However, in practice, with few entries per bucket, the cost of searching within a bucket becomes a constant.

Copy Overhead in Long RQs: Querying a prefix-hash-hybrid buffer requires snapshotting all buckets into a temporary data structure, typically a skip-list, as entries are not ordered across buckets. To return a qualifying key in order, RQs (i) iterate over and copy data from all buckets to a temporary sorted data structure, (ii) locate the first qualifying key, and (iii) sequentially iterate over the snapshot. Asymptotically, this requires $O(K^i \log K^i)$ time for such operations, where K^i is the respective buffer capacity.

3.4.1 Hash Skip-list (Hash-SL). A Hash-SL buffer uses a *classical skip-list* to implement buckets. The cost of probing Hash-SL is: $O(\log_b \kappa)$. Grouping keys with the same prefix into the same bucket allows efficient get and short scan performance for get-heavy workloads. However, it comes at the cost of additional metadata to organize the mapping between hash values and buckets, and memory overhead from express lanes. We compute the capacity of Hash-SL to be: $K^{hskip} = \frac{M - H \cdot p}{E + p \cdot (2 + \mathbb{E}[l-1]) + c}$. Note that many buckets with a small buffer size significantly reduce a buffer's usable capacity, leading to frequent flushes and fewer entries per flush.

3.4.2 Hash Linked-list (Hash-LL). A Hash-LL, on the other hand, uses a *sorted linked-list* for bucket implementation and trades off performance, reducing metadata overhead. Due to its single-level structure with no optimization, operations such as put and get require a linear search, costing $O(\kappa)$ time. Thus, Hash-LL stores more entries than Hash-SL, improving buffer usability and preventing frequent flushes. Due to less overhead per entry Hash-LL stores: $K^{hlink} = \frac{M - H \cdot p}{E + 2 \cdot p + c}$. For a skewed workload, the hash function may map many keys to the same bucket, slowing subsequent accesses to that bucket. Systems like RocksDB offer a knob, `threshold_use_skiplist`, to convert larger buckets into a skip-list on the fly [25].

3.4.3 Hash Vector (Hash-V). A Hash-V uses a *sorted, dynamically growing vector* to implement buckets. While Hash-SL and Hash-LL require additional memory to store metadata (pointers, level information, etc.), reducing the usable buffer capacity significantly, Hash-V offers a memory-optimized implementation with

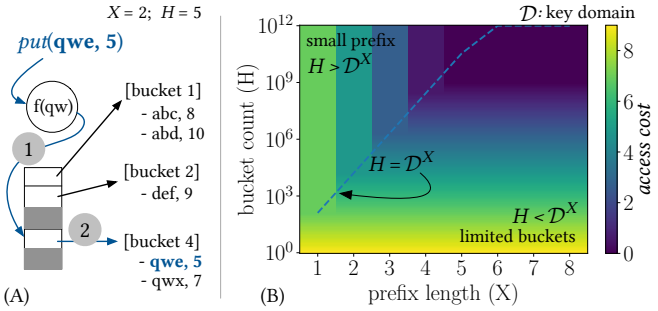


Fig. 3: (A) A flow of hash-hybrid buffer, configured with $X = 2$ and $H = 5$. (B) The higher values of X and H improve performance for *put*, *get*, and *short-scan* operations.

minimal metadata overhead. The number of entries required to saturate Hash-V is estimated as: $K^{hvec} = \frac{M - H \cdot p}{E + p + c}$. The PQs locate the target bucket and perform a binary search in $O(\log \kappa)$ time, where inserts additionally shift subsequent entries within the bucket, costing $O(\kappa)$. Hash-V offers the minimum memory footprint among hash hybrids due to its compact bucket storage.

We implement and evaluate two tree-based LSM-buffers: *B⁺-tree* [15] and *Adaptive Radix Tree* (ART) [41].

3.5 Tree

3.5.1 B⁺-tree. We implement a concurrent B⁺-tree with a fixed-fanout and optimistic lock coupling. The inner nodes hold 16 separator keys; leaves hold 32 entries and are chained into a doubly-linked list to facilitate faster scans. Each node preallocates a fixed-size array ahead in memory. A freshly split inner node holds only 8 of its 16 separators, linking to 9 children instead of 17. The buffer never merges on underflow, as deletes are realized *out of place* by adding a new entry to the buffer. Thus, rebalancing splits a node and leaves both newly created nodes half full. The worst-case node occupancy occurs when a leaf holds exactly 16 entries. We estimate it as: $K^{tr} = \lfloor \frac{M}{E + c + \frac{17p+8}{8} + \frac{33p+16}{128}} \rfloor$, where $\frac{17p+8}{8}$ is the cost per key, from the leaf array, half-empty in the worst case and $\frac{16+33p}{128}$ is the cost per key, from inner nodes also sitting half-empty, summed across every level above the leaves. Inserts and PQs run in $O(\log K^{tr})$, where inserts take an optimistic, lock-free path that compares-and-swaps the target leaf, but falls back to lock coupling on a split. RQs exploit the leaf-level next/prev chain and cost $O(\log K^{tr} + K^{tr} \cdot s)$.

3.5.2 Adaptive Radix Tree. An ART is a 256-way trie over the raw bytes of the key, and is also augmented with path compression and adaptively sized nodes (N4/N16/N48/N256) [41]. Unlike B⁺-tree, the ART's memory cost scales with its actual fan-out rather than a fixed value. Moreover, search cost depends on key length in bytes rather than the number of entries (K^{art}) stored in the buffer. Every node carries a fixed 24B header plus per-node-type child storage. A worst-case lower bound on buffer capacity, reached when every internal node remains at the smallest type (N4), is: $K^{art} = \lfloor \frac{M}{E + 28 + 4p + c} \rfloor$. Here, $(28 + 4p)$B is $N4$'s fixed per-node overhead, including a 24B header, 4B key array, and 4 child pointers, each of size p bytes. Insert and PQs both run in $O(k)$, where k is the average length of a key, with per-level work bounded by a

Table 2: Operations complexity per buffer († new buffers).

Buffer	*Put*	*Get*	*Short-scan*	*Long-scan*
V-Qsort	$O(1)$	$O(K^{vec} \log K^{vec})$	$O(K^{vec} \log K^{vec})$	$O(K^{vec} \log K^{vec})$
†V-Qscan	$O(1)$	$O(K^{vec})$	$O(K^{vec} \log K^{vec})$	$O(K^{vec} \log K^{vec})$
†V-Sorted	$O(K^{vec})$	$O(K^{vec})$	$O(K^{vec})$	$O(K^{vec})$
†Link-L	$O(K^{link})$	$O(K^{link})$	$O(K^{link})$	$O(K^{link})$
†Skip-L	$O(\log K^{skip})$	$O(\log K^{skip})$	$O(\log K^{skip})$	$O(\log K^{skip})$
InSkip-L	$O(\log_b \delta)$	$O(\log_b \delta)$	$O(\log_b \delta)$	$O(\log_b \delta)$
Hash-SL	$O(\log \kappa)$	$O(\log \kappa)$	$O(\log \kappa)$	$O(K^{hskip})$
Hash-LL	$O(\kappa)$	$O(\kappa)$	$O(\kappa)$	$O(K^{hlink})$
†Hash-V	$O(\kappa)$	$O(\log \kappa)$	$O(\log \kappa)$	$O(K^{hvec})$
†B⁺-tree	$O(\log K^{tr})$	$O(\log K^{tr})$	$O(\log K^{tr} + K^{tr} \cdot s)$	$O(\log K^{tr} + K^{tr} \cdot s)$
†ART	$O(k)$	$O(k)$	$O(k + K^{art} \cdot s)$	$O(k + K^{art} \cdot s)$

constant (a short linear scan for N4/N16 and direct index lookup for N48/N256); the RQ cost is expected to be $O(k + K^{art} \cdot s)$.

3.6 Adaptive Buffer (Adapt)

We propose an adaptive buffer, Adapt, that encapsulates all eleven implementations in a single buffer and automatically selects the appropriate type based on workload shifts. Adapt integrates two routines: a workload monitor and an advisor. The workload monitor maintains a lock-free ring buffer to store the proportions of operations, such as *puts*, *gets*, and *scans*, executed during the fixed-size window. Updates and deletes are recorded as *puts*, as they also add new items to the buffer. Later, when the buffer fills, these stats are shared with the advisor to determine the buffer type to use when instantiating a buffer after flush. The advisor aims to minimize the expected per-operation latency based on the proportion of operations recorded in the last window. Hash-hybrids are only eligible when the prefix extractor and bucket count are configured.

Cost Model. The advisor evaluates each buffer b and returns the one with minimum cost C; ties are resolved with InSkip-L. We define the cost as a weighted sum of all operation costs, *put* (P), *get* (G), and *scan* (S), as: $C(b) = P \cdot f_P(b) + G \cdot f_G(b) + S \cdot f_S(b)$ where, f_x represents asymptotic time for buffer b to execute corresponding operation x. The intuition is simple: V-Qscan gives $O(1)$ appends but pays $O(K^{vec})$ per *get* and $O(K^{vec} \log K^{vec})$ to sort-then-scan, so it only wins under write-heavy workloads with fewer reads. Hash-vector reduce *get* cost from $O(K^{vec} \log K^{vec})$ to $O(\log \kappa)$ where $\kappa \approx \frac{K^{vec}}{H}$, by partitioning keys across H buckets, thus improves overall latency for balanced workloads.

3.7 Workload Cost Estimation

The different buffer implementations have very different workload execution costs due to the varying fractions of operations and their access paths. For instance, using a vector-based buffer significantly improves write performance (see Table 2), but snapshotting and the unordered set of keys cause other operations to suffer. Operations such as *get*, *short-*, and *long-scan* are bottlenecked by the sorting cost of snapshots per query. An unsorted vector eliminates this by performing backward linear search for *get* operations, improving read performance. The vector- and skip-list-based buffers support concurrent operations using a mutex, offering even better performance, whereas the hash-hybrids are not thread-safe. In this section, we provide a cost model to evaluate LSM-buffer performance and its impact on overall system performance, see Fig. 4. We model the cost of different buffer implementations in terms of I/Os and time.

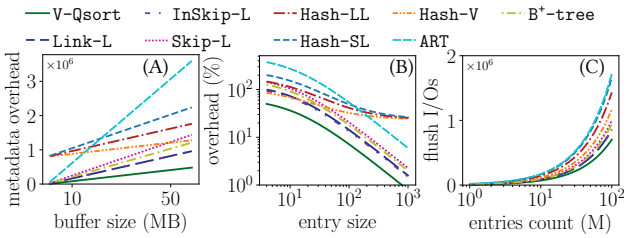


Fig. 4: (A) Metadata overhead is proportional to the #entries in a buffer and (B) entry size, increasing I/Os during (C) flush.

Workload I/O Cost. We define I/O cost as the total I/Os required to persist N entries to disk. For this analysis, we assume a page size of 4KB and a pointer size (p) of 8B to compute the overhead of node-based structures. In Fig. 4A, B, and C, we vary buffer size, entry size, and data size, respectively, along the x-axis and compute metadata overhead and flush I/Os along the y-axis. We model the overhead and number of flush I/Os as follows:

$$Overhead = M - (K^* \cdot E) \quad \& \quad \#flush\ I/Os = \frac{N \cdot E \cdot B}{M - Overhead}$$

where K^* is the number of entries in an active buffer and B is the average number of entries that fit in a page.

Fig. 4A, B show vector-based buffers have higher capacity, whereas the hash skip-list has the highest overhead due to its node-based structure and express lanes in each bucket. For a larger buffer size, the hash vector incurs a fixed memory overhead for the mapping, proportional to the number of buckets, and the classical skip-list exceeds it due to pointer duplication across multiple levels. However, for larger entry sizes, hash-hybrid memory overhead is dominated by bucket allocation, whereas standalone data structures use most of the buffer space, resulting in fewer flushes and fewer I/Os. Frequent flushes in hash-hybrids also lead to more rewriting at the base level (Level 1), as many production systems, such as RocksDB, treat Level 0 as a tiered queue and trigger flushes based on file count. In Fig. 4C, vector-based flushes the least, which means compact and more entries buffers are flushed, leading to fewer compactions.

4 BENCHMARK DESIGN

We benchmark eleven buffer implementations, widely used in commercial data systems, to understand their performance trade-offs.

Buffer Tuning. To analyze the buffer's performance across various workloads, we design the benchmark by varying buffer-specific knobs. For vector-based buffers, we switch between a *dynamic* and a *static memory allocation* strategy by configuring the count knob. We experiment with two skip-list-based buffers: *classical* and *inline*, to assess memory overhead and the effects of cache locality. The prefix hash-hybrids offer multiple knobs to trade memory for performance, which are configurable through *prefix length*, *bucket count*, threshold_use_skiplist, and total_seek_order. The knob threshold_use_skiplist defines an upper limit for converting linked-list-based buckets to a skip-list, as for skewed workloads, point reads suffer from larger buckets. total_seek_order disables snapshotting for prefix-hash-hybrids to reduce the search space by limiting it to a single bucket, improving short scans.

LSM-tree Tuning. To understand the performance implications of LSM tuning, we experiment by varying (i) the buffer size, (ii) the

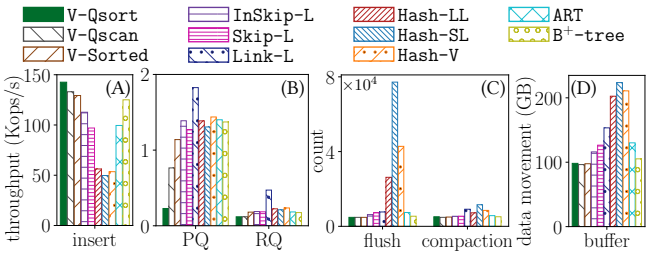


Fig. 5: (A) Vector-variants offer the best throughput for insert-heavy workloads at the (B) cost of slower queries, (C) but perform fewer flushes and (D) less data movement.

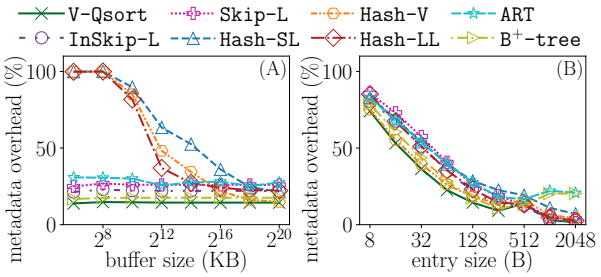


Fig. 6: (A) LSM-buffers have a constant per-entry metadata overhead which (B) reduces as entry size increases.

size ratio, (iii) the priority of writes and compactions, and (iv) write-ahead logging (WAL) settings. Note that prioritizing compactions over writes invokes a *rate limiter*, slowing down writes to prioritize compactions, but reduces space amplification and compaction debt.

Evaluation Metrics. We measure the following metrics.

(1) *Operational latency*: We measure the latency of each operation: insert, update, delete, PQ, and RQ, to quantify mean and tail latencies by analyzing the overall distributions (i.e., p95, p99).
(2) *Memory footprint*: We measure the total usable capacity for each buffer, accounting for structural metadata overhead.
(3) *Overall data movement*: We measure the total bytes transferred to/from slow storage to quantify metadata overhead.
(4) *Space amplification*: We define space amplification as the ratio of the actual database size to the unique entries size in a workload, to quantify the storage overhead due to stale entries.
(5) *Compaction debt*: By default, commercial systems over-saturate the shallower levels to ensure high ingestion throughput. This *pending work* accumulates over time, leading to huge compaction debts that make future writes slower.

5 EXPERIMENTAL EVALUATION

Goal of Study. We set out to answer the following three questions.

(1) *Performance implication*: How does the choice of a buffer data structure affect the LSM-tree performance?
(2) *Knobs influence*: How do buffer knobs and LSM tuning knobs change the entire state and performance of the LSM-tree?
(3) *Workload implication*: How do the workload characteristics, composition, and distribution impact the performance?

Experimental Setup. We use machines with two Intel Xeon Gold 6126 CPUs (2.60 GHz), supporting 24 cores and 48 threads. It has 192GB of DDR4 RAM, a 38.5MB L3 cache, and a 240GB SSD, running Ubuntu 24.04.2 LTS [39]. To perform a fair comparison, we use the existing four implementations from a widely used LSM-engine, RocksDB, including (i) `V-Qsort`, (ii) `InSkip-L`, (iii) `Hash-SL`, and (iv) `Hash-LL`, and implement the other seven, (i) `V-Qscan`, (ii) `V-Sorted`, (iii) `Link-L`, (iv) `Skip-L`, (v) `Hash-V`, (vi) `ART`, and (vii) `B+-tree` on top of RocksDB v10.10.1 [27].

Default Setup. Unless otherwise specified, we use SSDs for storage for the experiments. The `low_pri` knob is unset to prioritize writes over compactions, and WAL is disabled to measure the true overheads. The tree has a size ratio of 6. We point out that a dynamically growing vector provides limited benefit and periodically spikes tail write latency. Thus, we set `count` to K^{vec}, which pre-allocates memory. The `threshold_use_skiplist` is set to M/E to follow a linked-list-based structure for buckets in `Hash-LL`. By default, we set *prefix length* (X) to 6 and *bucket count* (H) to 100k. We use KVBench [69] and Tectonic [49] to generate the workloads.

5.1 Performance Implication

Setup 1: Preload the database with 80M inserts, then run 10k PQs mixed with 10M inserts, and issue 1k RQs mixed with 10M inserts.

Performance Varies across Buffers Substantially. We observe that `V-Qsort` achieves the highest insert throughput, 47% higher than `Skip-L` (the default), as shown in Fig. 5A. This is due to the insert-heavy workload, where vector-based buffers, `V-Qsort` and `V-Qscan`, benefit from constant-time appending of entry contiguous memory array. However, this comes at the cost of slower PQ and RQ performance, where `V-Qsort` is 82% slower, `V-Qscan` is 39% slower, and `V-Sorted` is 10% slower than `Skip-L` for PQs, and vector-variants are up to 35% slower for RQs than `Skip-L`. On the other hand, hash-hybrids, which show 45% less insert throughput than the default buffer, perform better for PQs and RQs, up to 13% and 27% better throughput, respectively, Fig. 5B. Furthermore, both `ART` and `B+-tree` shows comparable PQ and RQ throughput with `Skip-L`, 8 – 10% and 1 – 6%, respectively. The metadata overhead of hash-hybrids slows inserts, as we will see shortly. Linked-list has the worst insert throughput due to its inefficient search in maintaining the sorted order. We skip it for our evaluation.

Flushes improve Hash-hybrids performance. Hash-hybrids flush up to one order of magnitude more files, resulting in more data movement and improving PQs and RQs performance as shown in Fig. 5C, D. `Hash-SL` shows the highest number of flushes, 16× more than `V-Qsort`, resulting in 2× more data movement than in `V-Qsort`. In a uniform workload, `ART` likely splits branch at smaller levels, resulting in 18% more flushes and 12% higher data movement than `Skip-L`. RocksDB organizes Level 0 with overlapping files and triggers compaction based on file count rather than fullness, resulting in a compact base level for hash-hybrids, which improves read performance by reducing the number of overlapping files.

Higher Metadata Overhead Leads to Frequent Flushes. Hash-hybrids flush data more frequently due to their structural overhead, leading to smaller files, superfluous I/Os, and write stalls.

Setup 2: We run two experiments by: (i) varying buffer size along the x-axis from 64KB to 1GB with 128B entry size and (ii) varying entry size on the x-axis from 8B to 2KB with 128MB buffer size.

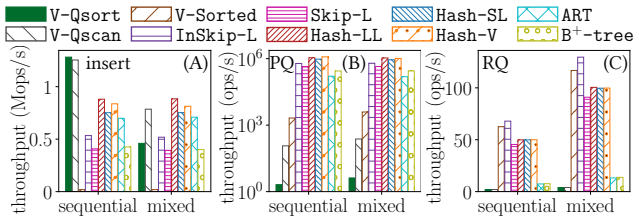


Fig. 7: (A) V-Qscan offers the best throughput for sequential and comparable for mixed workloads, (B) hash-hybrids work well for PQs, and (C) InSkip-L for RQs.

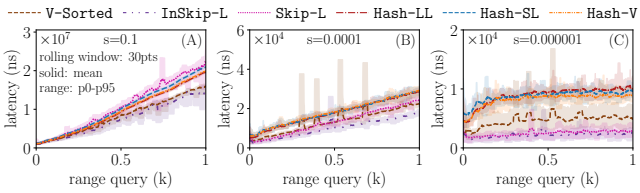


Fig. 8: Hash-hybrids add a snapshotting overhead and slow down RQs even for smaller selectivities

In Fig. 6A, for buffer sizes 64KB and 256KB, hash-hybrids initiated a flush right after a single insert, as the pointer-to-buckets overhead is more than the buffer size, around 780KB. V-Qsort, Link-L, and Skip-L show constant overhead of nearly 14%, 22%, and 25%, respectively. Moreover, ART and B+-tree also shows constant overhead of nearly 28% and 17%, respectively. This stems from the fact that each entry in RocksDB is padded with metadata such as sequence numbers, entry types (tombstones or range deletes), and protection bytes (checksum). In V-Qsort, each entry also adds a pointer-size overhead, whereas in skip-list-based structures, the data is organized in multiple levels, with a *splice* overhead in InSkip-L. As the buffer size increases, the bucket pointer overhead diminishes and plateaus due to per-entry overheads. In Fig. 6B, for entry size 8B, the metadata (sequence number and entry type) takes almost 74% (V-Qsort) to 85% (Hash-LL) of the buffer space to store per-entry extra information, including bucket pointers for hash-hybrids. As we increase the entry size, it becomes negligible due to the reduction in the fraction of metadata relative to the entry. V-Qscan and V-Sorted has the similar overhead as V-Qsort.

TA I: *Under low memory, hash-hybrids must be avoided as they increase the flush and compaction frequency, resulting in more I/Os.*

In-memory Buffer Impacts Performance Considerably. To study the raw performance, we experiment with two types of workloads: (i) sequential, where operations of the same type are grouped, and (ii) interleaved, where operations are mixed. The underlying data structure used significantly impacts in-memory performance.

Setup 3: We set the buffer size to 128MB, entry size to 128B, and key size to 8B. For hash-hybrids, we set H = 100k and X = 6, The workload fits in memory and issues 0.7M inserts, 50k empty PQs, 50k non-empty PQs, and 1k RQs with selectivity 0.1. (i) We run it sequentially to capture per-operation throughput; (ii) we preload 10% of unique inserts and issue the remaining operations randomly.

Fig. 7A shows that V-Qsort offers the highest throughput, about 2.3× and 1.7× higher than InSkip-L and the hash-hybrids, respectively, for sequential workloads. The throughput drops for mixed workloads as interleaved operations impact the cache locality. Due

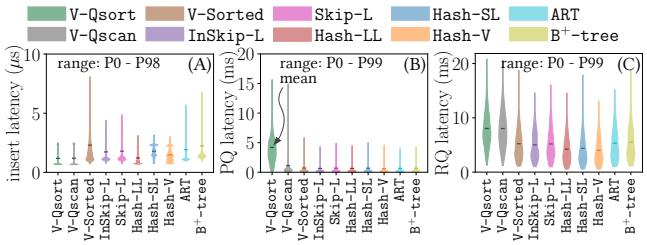


Fig. 9: (A) V-Sorted has the highest tail insert latency. The tail query latency depends on accesses to slow storage (B-C).

to the snapshotting and sorting overhead for PQs and RQs, V-Qsort is six orders of magnitude slower than hash-hybrids and five orders of magnitude slower than Skip-L for PQs. For both sequential and mixed workloads, ART outperforms Skip-L more than 30%.

Hash-hybrids Offer Faster PQs, InSkip-L Outperforms for RQs. Hash-hybrid buffers achieve the highest PQ throughput for both sequential and interleaved workloads, showing up to 2× better performance than the InSkip-L by leveraging hashing as shown in Fig. 7B. They cluster similar hash entries into the same bucket, store significantly fewer entries, and reduce the search space for every PQ. InSkip-L outperforms for RQs due to no snapshotting or sorting overhead and the use of splice, as shown in Fig. 7C. It is about 1.3× better than hash-hybrids for both workloads. Unlike other vector-variants, V-Sorted shows about 20% better throughput compared to hash-hybrids due to no sorting and using binary search.

Query Selectivity Significantly Affects the Buffer Choice. We vary RQ selectivity from 0.1 to 0.000001 and measure the impact on RQ latency for various buffer implementations. In Fig. 8A-C, we show latency for six buffers, including V-Sorted, InSkip-L, Skip-L, and three hash-hybrids, across 1k RQs on the x-axis. As RQ selectivity reduces, hash-hybrids, if not configured carefully, add unnecessary snapshotting overhead even when the qualifying keys reside in a single bucket. Due to the lack of an efficient method to search across buckets, hash-hybrids merge all entries into a temporary structure before scanning for qualifying keys. InSkip-L offers the best performance, 25% to 80% better than hash-hybrids, across selectivities, from large to small, and also outperforms V-Sorted, achieving 45% faster RQs for smaller selectivity.

TA II: *For write-heavy,* V-Qscan *offers the best in-memory performance; Hash-hybrids dominates for PQ-heavy workloads.*

Reads for V-Qsort has Higher Tail Latency. Fig. 9A-C show per-operational latencies for our *Setup 1* experiment. V-Sorted exhibits the highest tail latency due to the overhead of inserting target keys while maintaining the sorted order. V-Qsort performs poorly for PQs, showing 4× higher mean latency due to its slowest read path, which includes taking a snapshot, sorting it, and then performing a binary search. RQ performance is dominated by storage I/Os for all buffers. ART and B+-tree shows comparable tail latencies with skip-list-based buffers and V-Sorted across operations. Vector-variants are still the slowest due to their costly in-memory search.

5.2 Buffer Tuning

Dynamic Memory Allocation Adds Spikes. In Fig. 10A, dynamic memory allocation for V-Qsort adds spikes to the insert latency,

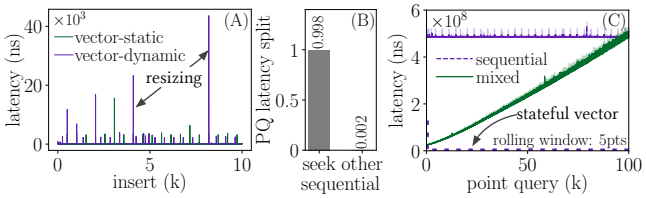


Fig. 10: In `V-Qsort`, (A) latency spikes due to resizing, and (B, C) sorting slows down PQs significantly for larger buffers.

showing 1.2× higher p100 than a static allocation. Whenever the allocated size fills, the resizing process kicks in and stalls writes while it moves entries to the larger allocated space. Moreover, the sorting cost in the default `V-Qsort` implementation bottlenecks PQs, taking 99% of the total time as shown in Fig. 10B. This cost is linear to the number of entries in the buffer. Also, `V-Qsort` even pays this cost when the PQs are consecutive. We optimize it by designing a stateful vector that reuses the sorted snapshot from the previous PQ, improving performance by 5× as shown in Fig. 10C.

Snapshoting and Sorting Impacts Performance. Vector-based buffers perform suboptimally due to the high cost of snapshotting and sorting for PQs and RQs. In Fig 11, we break down PQ performance for `V-Qsort`, `V-Qscan`, and `V-Sorted` in terms of time spent on snapshotting, sorting, and searching. `V-Qsort` shows a bottleneck on sorting, spending 99% of the time; `V-Qscan` linearly scans the vector on PQs and is thus bottlenecked by searching, performing 88× better than `V-Qsort`. `V-Sorted` runs a binary search and spends 99% time in copying pointers, showing up to 500× better latency than `V-Qsort`.

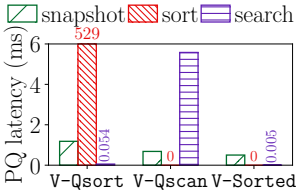


Fig. 11: Sorting and snapshoting slows down `V-Qsort`

Setup 4: We preload 0.7M entries before issuing 10k PQs, while varying the bucket count from 100 to 100k and the prefix length from 1 to 8, and then run 1k RQs in another experiment while varying the common prefixes for the start and end keys.

More Buckets Reduces Capacity for Hash-hybrids. In Fig. 12, we vary the bucket count on the x-axis and the mean capacity for hash-hybrid buffers on the y-axis. `Hash-SL` shows the largest drop in mean capacity, about 42%, due to its multi-level structure. `Hash-LL` drops by 7% drop, and `Hash-V` shows a 25% drop in the mean capacity due to the lower memory overhead than `Hash-SL`. `ART` and `B+-tree`, at 1M buckets, shows 58% and 80% more capacity than `Hash-SL`, respectively.

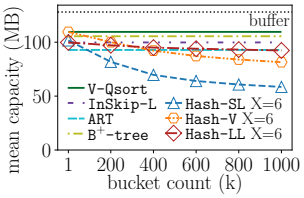


Fig. 12: Increase in bucket count reduces capacity.

More Buckets Improves Inserts and PQs Latency. Fig. 13A shows that hash-hybrids improve insert and PQ latency when using more buckets. As we increase bucket count to 100k, the `Hash-SL` shows 6.8× better ingestion throughput, the `Hash-V` shows 5.3× better, and the `Hash-LL` improves by 3 orders of magnitude compared to 1k buckets. The smaller search space, a bucket, helps efficiently locate the target position, improving both insertion and PQs throughput. In `Hash-V`, the contiguous memory layout of buckets improves PQ throughput by up to 14×, for 1k buckets, than`Hash-SL` and hash linked-list (Fig. 13B), as well as when the buckets increase to 100k, throughput improves by 48% over 1k buckets by itself and stays 30% above from hash skip-list.

Prefix Length and Domain Caps the Keys Distribution. The length of the configured prefix and key domain in a workload affects bucket fill. If the prefix is small, fewer buckets are used, resulting in more entries per bucket and lower operational throughput. In Fig. 13C,D, we vary the prefix length on the x-axis, where the key size is set to 8, offering a 62^8 size domain (using a-z, A-Z, 0-9). The bucket count is fixed to 100k. As we increase the prefix length, insert and PQ throughput improve by 2×, but saturate at a prefix length of 3. This is because the entries are distributed equally across all buckets and are bottlenecked by the number of buckets.

Short-RQs Perform Better using *total_seek_order*. In Fig. 13E, we run an adversarial workload where all RQs have a common prefix, varying between 0 and 8 characters. The length of the common prefix is inversely proportional to the RQ selectivity. For hash-hybrids, the prefix length is set to 6, which means short-RQs (from points 6 to 8) probe only a single bucket, reducing RQ latency. When the start and end keys share a common prefix of length greater than or equal to the prefix length, RQ scans a single bucket, enabling 2× higher throughput than the `InSkip-L`.

TA III: *To maximize hash-hybrid performance, a higher bucket count, preferably close to the domain cardinality, is the way to go.*

5.3 LSM tuning

Setup 5: We evaluate four settings by enabling and disabling the *low_pri* (`lp`) and *disableWAL* (`WAL`) knobs. We issue 10k PQs and 1k RQs mixed with 20M inserts on a database with 80M entries.

Prioritizing Writes with *low_pri* Improves Ingestion. Tuning LSM-tree for writes by disabling *low_pri* knob improves ingestion throughput, on average, 18% across different buffers. In Fig. 14A, `lp` = 0 and `wal` = 0 provide the highest throughput, where `V-Qsort` provides the best among different buffers, showing 30% improvement compared to `lp` = 1 and `wal` = 1. `Hash-LL` shows the minimal 3% improvement. This is due to no WAL overhead and the prioritization of writes over compactions, which, however, results in larger shallower levels. A belly LSM-tree, on the other hand, impacts the PQs and RQs performance, slowing them up to 2% and 31%, respectively, across different buffers, see Fig. 14B, C. The default buffer, `InSkip-L`, in RocksDB also shows up to a 1% drop in PQs and a 41% drop in RQs when writes are prioritized over compactions. `ART` and `B+-tree` also see a drop in RQ 45% and 48%, respectively.

Compaction Accumulates when Writes are Prioritized. In Fig. 15A, we show the level fullness for each LSM tuning that we experimented with for the same setup. For `lp` = 0 and `wal` = 0, the shallower levels, L1, show 4× more data, L2 show 3.7× more data, whereas L3 holds 2.3× more data than the configured capacity, leading to 2.5GB of compaction debt (Fig. 15B). Enabling WAL, i.e., `wal` = 1, with `lp` = 0, slows down writes to persist the WAL logs and gives more time for compactions to catch up, bringing it down to 2.4GB of compaction debt as shown in Fig. 15C. In the other two settings, when `lp` = 1, LSM-tree maintains its shape, keeping data close to ideal with 0.006GB compaction debt when *wal* = 0 and 0.005GB when `wal` = 1 as shown in Fig 15C, D. The pattern remains

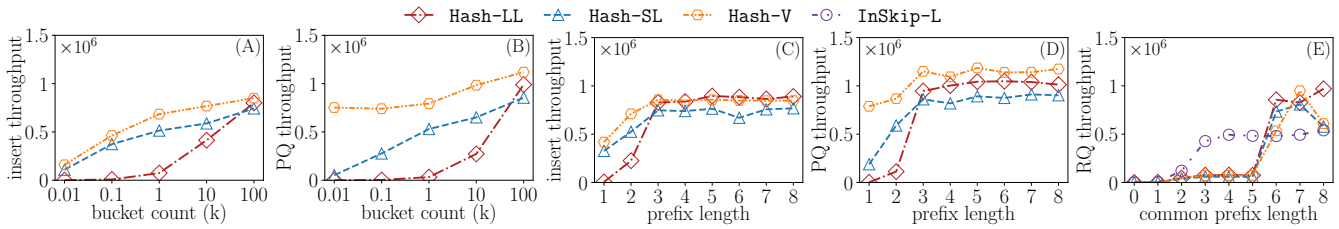


Fig. 13: (A, B) More buckets improve inserts and PQ throughput, adding a minimal per-bucket pointer size overhead. (C, D) Prefix length saturates performance at $H = \mathcal{D}^X$, whereas (E) RQs throughput is tunable through a common prefix knob.

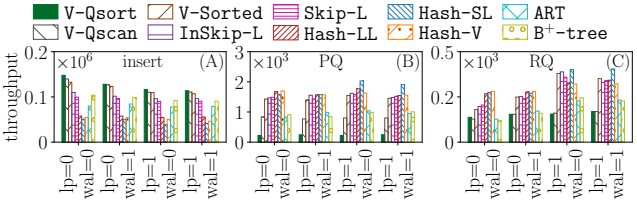


Fig. 14: Disabling `low_pri` improves (A) ingestion at the cost of compaction debt, (B) PQ latency, and (C) RQ latency.

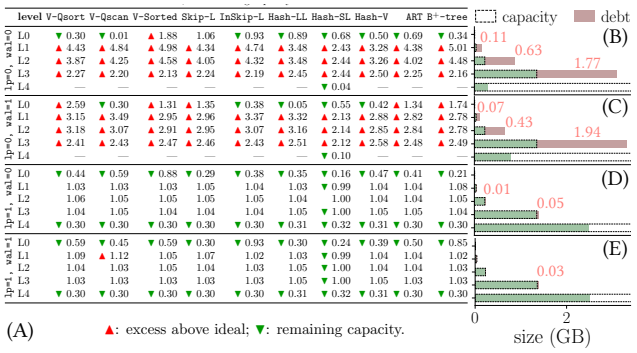


Fig. 15: Prioritizing compactions over writes keeps the LSM-tree in shape and does not accumulate huge compaction debt.

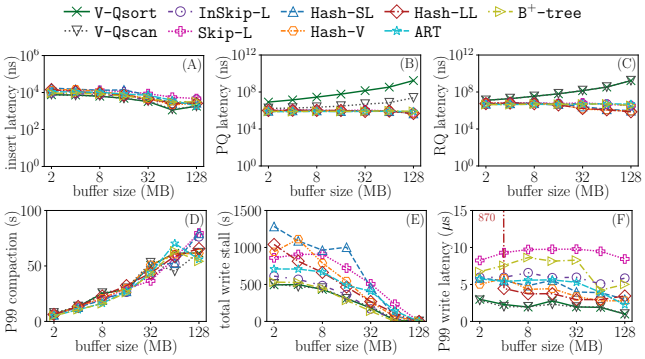


Fig. 16: Except for vector variants, larger buffers improve operational latencies at the cost of high compaction tail latencies. The tail write latency improves as stalls reduce.

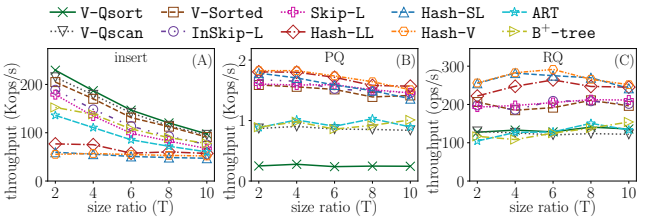


Fig. 17: Throughput drops as the size ratio increases.

the same across different file picking policies, as compaction merges data after Level 0 that contains similarly sized files.

Vector-variants Improves Ingestion at the Cost of PQ Latency. We vary the buffer size from 1MB to 128MB to evaluate its impact on operational latency, using the same workload setup. Fig. 16A shows that increasing the buffer size reduces the mean ingestion latency across buffers, `V-Qscan` dropping the most up to 89%. Further, in Fig. 16B, the mean PQ latency plateaus due to slower disk I/Os for most buffers. However, vector-variants show an increase in PQ latency as the overhead of sorting and snapshotting increases with the buffer size. For a 128MB buffer, `V-Qsort` and `V-Qscan` has a mean latencies 705× and 9.8× higher than `Skip-L`, respectively. The RQ in `V-Qsort` and `V-Qscan` are up to 145× slower than `Skip-L`, Fig. 16C, as both perform sorting to return keys in a certain order.

Fewer Write Stalls with High Compaction Tail Latency. In Fig. 16D-F, we show the impact of larger buffer size on compaction tail latencies and write tail latency due to stalls. As the buffer size increases from 2MB to 128MB, compactions take up to 13× longer because larger buffers mean larger files at each level, resulting in more data per compaction. However, increasing the buffer size reduces the overall time spent in stalling by up to 97% because each buffer holds more data before being flushed to slow storage. Fig. 16F shows the writes tail latency, in which the increase in buffer size improves vector-variant performance up to 67%, whereas node-based probabilistic structures like `Skip-L` and `InSkip-L` stay the same due to capped maximum height, i.e., 4. Hash-hybrids improve write tail latency by, on average, 37% as the buffer size increases.

> **TA IV:** *Larger sizes reduce ingestion latency across buffers, but PQs and RQs become expensive for unsorted vector-variants.*

Throughput Decreases for Larger Size Ratios. In Fig. 17A, increasing the size ratio from 2 to 10 results in a decrease in ingestion throughput by up to 62% for skip-list and 57% for vector. Hash hybrids are more resilient to changes in the size ratio; their insert throughput decreases by only 26%. In Fig. 17B, the PQ throughput drop is less significant than insert, with the `Skip-L` and `Hash-V` decreasing by 13% and 17%, respectively. `V-Qsort` and `V-Qscan` PQ throughput are not heavily affected by size ratio changes, only dropping by up to 4%. In Fig. 17C, RQ throughput remains largely unaffected by the size ratio, as it is dominated by expensive I/Os.

Setup 6: We evaluate scalability across three workloads while varying the thread count from 1 to 16: (i) a 300K insert-only workload, (ii) a 10K PQ-only workload issued on a 100K preloaded database, and (iii) a mixed workload with 10K inserts and 10K PQs.

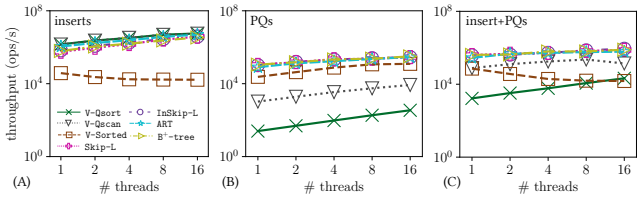


Fig. 18: As the number of threads increases, the throughput improves across buffers for different workloads: (A) write-only, (B) read-only, and (C) mixed.

Table 3: Performance under update-intensive workloads.

Buffer	Data movement (GB)					Flush count					Compaction count				
	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%
V-Qsort	**95**	96	**92**	91	**87**	**4803**	**4803**	**4803**	**4803**	**4803**	4704	4845	**4236**	4210	3764
V-Qscan	95	**94**	93	**90**	87	4803	4803	4803	4803	4803	**4676**	**4444**	4386	**4021**	**3599**
InSkip-L	115	112	110	109	106	6266	6266	6266	6266	6266	5151	4898	4632	4518	4052
Skip-L	140	140	138	136	139	15074	15074	15074	15074	15074	5950	5752	5468	5142	4957
Hash-LL	200	197	193	189	185	26230	26232	26230	26232	26231	6865	6726	6368	6085	5826
Hash-SL	**221**	**220**	**216**	**215**	**211**	**77173**	**77172**	**77170**	**77169**	**77169**	**11432**	**11204**	**11011**	**10900**	**10558**
Hash-V	208	207	205	199	194	42805	42804	42803	42803	42803	8635	8176	7965	7829	7230
B+-tree	93	103	101	99	97	4923	5407	5407	5407	5407	4342	4802	4593	4347	4183
ART	116	127	124	123	119	6744	7407	7407	7408	7408	4736	5295	4891	4681	4244

Throughput Scales with Threads. Fig. 18A-C demonstrates that concurrency scalability heavily depends on the buffer implementation and the workload pattern. As shown in Fig. 18A, under a write-intensive workload, node-based data structures like `InSkip-L`, `Skip-L`, `ART`, and `B+-tree` scale with threads as they either support lock-free or optimistic lock coupling. `InSkip-L` offers the highest scalability, increasing throughput by 6.74× as the thread count increases from 1 to 16. Moreover, `Skip-L` scales up to 5.66×, `B+-tree` by 5.54×, `ART` by 4.51×, `V-Qsort` by 3.95×, and `V-Qscan` by 3.93×. Only `V-Sorted` shows a drop in throughput due to its optimization for queries. `V-Sorted` preserves an always-sorted state where each thread begins by batching its newly inserted keys in a thread-local buffer and then merges it with the global vector, maintaining a sorted state. Under a read-heavy workload in Fig. 18B, `V-Qsort` scales the best, offering 13.87× higher throughput with 16 threads while `InSkip-L` scales the poorest (2.77×). For PQs, `V-Sorted` throughput scales by 5.17× as each PQ is realized from an always-sorted vector. In Fig. 18C, all buffers scale, except `V-Sorted`, due to the interleaved inserts with PQs, which invalidate the snapshot and limit the lock-free access to the global vector.

5.4 Workload Implication

Updates Reduce Data Movement Uniformly for all Buffer Types. To evaluate the impact of updates on performance, we ingest 80M entries while varying the fraction of updates. Table 3 shows that with more updates in a workload, the overall data movement reduces. This is because while the number of writes to buffer remains the same (as LSM-buffers realize updates *out of place*), leading to the same number of flushes, the invalid entries are eventually purged during compactions. However, the relative benefits of the different buffer implementations are unaffected by updates.

Skip-list-based Buffers Offers Best Performance for YCSB. In Table 4, we measure the performance of different buffers against standard YCSB A-F benchmarks. The default implementation of `V-Qsort` buffer is the slowest, 9.8× slower than the `Skip-L`. `V-Qsort` execution time is dominated by the slowest PQs which `V-Qscan` avoids through linear scan, making it 1.2× slower than `Skip-L`. However, in YCSB F, the `Skip-L` is outperformed by hash hybrids

Table 4: Throughput (Kops/s) for YCSB workloads

buffer	YCSB A	YCSB B	YCSB C	YCSB D	YCSB E	YCSB F
V-Qsort	3.13	1.69	7.60	4.75	1.36	0.26
V-Qscan	22.90	13.89	8.10	7.24	1.35	0.19
InSkip-L	26.19	16.28	8.13	7.67	1.60	0.25
Skip-L	26.50	16.59	8.25	7.68	1.56	0.29
Hash-LL	21.71	13.93	7.64	7.42	0.64	0.84
Hash-SL	23.16	14.60	7.56	7.50	1.31	0.89
Hash-V	23.73	14.70	7.44	7.46	0.64	0.50
B+-tree	21.62	12.50	5.98	5.84	0.14	0.29
ART	20.10	11.59	6.00	5.68	0.14	0.30

as the read-modify-write operation performs a PQ lookup before every update. The `Hash-SL` is 3.1× faster, the `Hash-LL` is 2.9× faster, and the hash vector is 1.7× faster than the `Skip-L`.

Throughput Drops Gracefully as Data Size Increases. To quantify the impact of data size on the buffer performance, we run an experiment with 80M inserts, followed by 10k PQs and 1k RQs mixed with 20M new inserts. We vary the scale from 1GB to 16GB and run for six buffers. In Fig. 19, the throughput of all buffers drops as the dataset scales up. Vector-based and node-based structures show drops of 1.77× and 1.69×, respectively. At 16GB, `V-Qsort` offers 2.61× higher throughput than `Hash-SL`. The performance varies due to the accumulated write stall associated with the number of compactions.

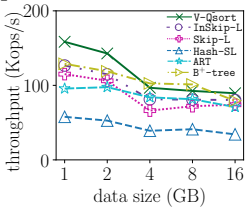


Fig. 19: Throughput scales with data size.

Setup 7: We run a 7-phase workload where P1 loads 8M entries, and the next six phases each contain 1M interleaved operations. P2 runs 99% PQs and 1% inserts, P3 runs 98% updates with 2% PQs, P4 is read-heavy, 99% PQs and 1% inserts, P5 issues 50% inserts and 50% deletes, P6 is runs 99% short-RQs with varying selectivity, between 10 and 1000 keys per RQ, and P7 ingests 1M entries.

Switching Buffer Wins for Shifting Workloads. One buffer does not fit all workload patterns; thus, we designed a switching buffer that selects the best option based on workload characteristics. Currently, our simple cost model switches the buffer to optimize the overall execution time of an operation. We consider inserts, updates, deletes, and read-modify-writes as write operations, PQs as reads, and RQs as scans while computing the cost over a window of operations. Then, the model selects the best buffer during flush.

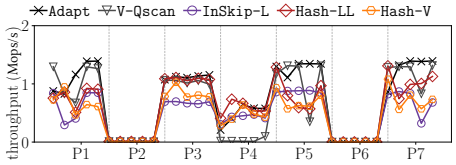
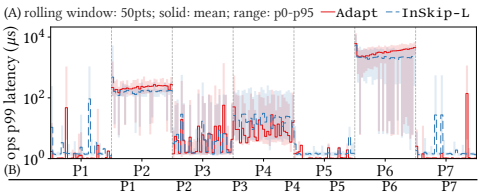


Fig. 20: `Adapt` benefits from all buffer switching as per workload characteristics

Fig. 20 shows the benefits of `Adapt`, which starts as an `InSkip-L` and switches based on the workload shifts. The Vector-variants outperform all other buffers in the write-heavy phase. In P1, `V-Qscan` buffer performs the best, offering an average of 1.15Mops/s, whereas `Adapt` offers 1.14Mops/s; however, the default buffer is slower, providing



	P1	P2	P3	P4	P5	P6	P7
Adapt (µs)	1074.45	1572.37	1386.93	544.79	8.08	13282.87	1156.09
InSkip-L (µs)	1080.31	5305.18	570.12	257.89	11.82	11863.14	552.55

Fig. 21: (A) `Adapt` buffer adds no overhead, as (B) phasewise maximum p99 latency is stable.

0.80Mops/s throughput. For P4, `Hash-LL` offers 0.64Mops/s, but `V-Qscan` suffer due to PQs, 0.04Mops/s, thus `Adapt` switches to `Hash-LL`, achieving the best performance, 0.51Mops/s. In P7, the workload switches back to insert-heavy, and `Adapt` switches to `V-Qscan`, achieving the best throughput of 1.21Mops/s. This switching of buffer happens on the fly without any overhead, as shown in Fig. 21A, avoiding the latency spikes seen in `InSkip-L` during phase transitions, such as from P1 and P2, as shown in Fig. 21B.

TA V: *Default buffer performs suboptimally under shifting workload, and no single buffer excels in all types of workload.*

Setup 8: An Intel Xeon E5-2670 v3 CPUs with 24 cores, 60MB L3 cache, and 512GB RAM is mounted with 558GB Seagate hard disk drive (HDD), 1.92TB SATA SSD, and 1.8TB NVMe PCIe.

Mechanical Constraints Limit HDD Throughput. In Fig. 22A, the benefits across buffers are consistent for different hardware. HDD shows, on average, 8.1× lower throughput than SSD. This is due to the bottleneck in mechanical arms for reading/writing data. Unlike HDDs, NVMe and SSDs rely on circuit-based hardware for data transfer, avoiding the inefficiencies of physical disk spinning. In Fig. 22A-C, NVMe offers the highest throughput across all three operations. The performance improves by several orders of magnitude through a direct connection between storage and CPU via the Peripheral Component Interconnect Express bus, which supports higher bandwidth [9]. As NVMe data access is faster than HDD access, the performance bottleneck shifted from hardware-bound to CPU-bound. In Fig. 22B, the throughput difference between `V-Qsort` and `Hash-V` is much wider on NVMe than on the HDD, 19.1× lower for NVMe and 1.9× in HDD than the `Hash-V`.

6 A HANDBOOK FOR PRACTITIONERS

We provide a handbook for navigating the LSM-buffer design space and selecting the best buffer and tuning for a given workload.

(1) **`V-Qscan` is preferred for write-intensive workloads:** Vector-based buffers offer the best throughput for write-heavy workloads with minimal memory overhead (Fig. 5A, Fig. 7A).
(2) **Use vector-variants for memory constraint environment:** Vector buffers add the lowest 15% memory overhead among all buffers. Thus, to store more entries in memory, a vector-based buffer is preferred (Fig. 6), also benefiting PQs on recent keys.
(3) **Vector-variants for fewer flushes and compactions:** Vector-variants stores more entries, resulting in fewer flushes and compactions, and is preferred to reduce data movement (Fig. 5C, D).
(4) **`InSkip-L` for frequently shifting workloads:** For workloads with less predictability, `InSkip-L` is the preferred choice, as it trades off write performance and memory (Fig. 20).
(5) **Use hash-hybrids when memory is sufficient:** If memory is not a constraint, hash-hybrid is the best choice for workloads consists of inserts, PQs, and/or short-RQs. `Hash-V` works best due to its less memory overhead among other hash-variants while offering comparable PQ and RQ performance (Fig. 13).
(6) **`V-Qsort` is worst for PQs and RQs:** Even though vector-based buffers excel at ingestion, `V-Qsort` performance degrades with the frequency of point queries, due to high sorting cost (Fig. 11).
(7) **Use Static allocation to avoid latency spikes:** The dynamic memory allocation in vector-based buffers adds an unnecessary spike to insert latency when resizing triggers, which can be reduced by using the `count` knob to pre-allocate more memory.
(8) **Set `low_pri` to reduce compaction debt:** To offer high write throughput, by unsetting `low_pri`, production systems overfill shallower levels, resulting in slower future operations due to accumulated compaction and write stalls (Fig. 15A-E).
(9) **Use larger buffers to improve latency:** Larger buffers improve latency for node- and hash-hybrid-based structures, storing more entries while offering efficient search (Fig. 16A-C).
(10) **For shifting workloads, `Adapt` is optimal:** For shifting workloads, the adaptive buffer that switches implementations while monitoring the workload on the fly is recommended (Fig. 20).

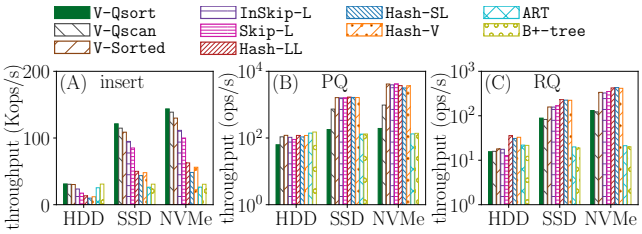


Fig. 22: NVMe consistently outperforms SSD and HDD.

7 RELATED WORK

Prior research has focused on empirical analysis for modeling and benchmarking key-value stores under realistic workloads [11, 20, 33, 34]. Cao *et al.* [11] perform characterization of three RocksDB production use cases at Facebook, and reveal that key-value size distribution and access patterns are highly application-specific. While these findings show the importance of aligning LSM-engine configuration with workload characteristics, the impact of the memory buffer component remains unexplored in such benchmarking.

Recent research has shown interest in exploring LSM-buffer performance. Kaushik *et al.* [8, 38, 40] provide the first comparative performance benchmark on four existing buffer implementations in `V-Qsort`, `InSkip-L`, `Hash-SL`, and `Hash-LL`—focusing primarily on benchmarking the performance of insert, PQ, and RQ. This study highlights the sensitivity of the LSM-tree with underlying buffer choices and provides initial guidelines on buffer selection based on workload composition; however, the operations benchmarked are limited to three operations and four buffer choices. Existing work does not offer the flexibility to use different implementations as per workload shifts, despite it being a critical design choice.

8 CONCLUSION

LSM-buffer is the only entry point for all types of operations in LSM-based storage engines; thus, its data structure and tunable knobs affect system performance to several orders of magnitude. Due to the limited choices available in production systems and the lack of awareness of how to tune the buffer for a given workload, LSM-engines perform suboptimally. We evaluate eleven LSM-buffers and propose an adaptive buffer that switches its implementation based on the active workload. Lastly, we provide a handbook with guidelines for choosing an appropriate buffer for a given workload.

ACKNOWLEDGMENTS

We thank Alexander H. Ott for contributing to the buffer designs.

REFERENCES

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R Borkar, Yingyi Bu, Michael J Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916. http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf

[2] Sattam Alsubaiee, Alexander Behm, Vinayak R Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014), 841–852. http://www.vldb.org/pvldb/vol7/p841-alsubaiee.pdf

[3] Apache. 2023. Accumulo. *https://accumulo.apache.org/* (2023).

[4] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).

[5] Apache. 2023. HBase. *http://hbase.apache.org/* (2023).

[6] Apple. 2018. FoundationDB. *https://github.com/apple/foundationdb* (2018).

[7] Manos Athanassoulis, Subhadeep Sarkar, Tarikul Islam Papon, Zichen Zhu, and Dimitris Staratzis. 2022. Building Deletion-Compliant Data Systems. *IEEE Data Engineering Bulletin* 45, 1 (2022), 21–36. https://dl.acm.org/doi/10.1145/3599724

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2020. SILK+ Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads. *ACM Transactions on Computer Systems (TOCS)* 36, 4 (2020), 12:1–12:27. https://dl.acm.org/doi/10.1145/3380905

[9] Mahsa Bayati, Janki Bhimani, Ronald Lee, and Ningfang Mi. 2019. Exploring Benefits of NVMe SSDs for BigData Processing in Enterprise Data Centers. In *5th International Conference on Big Data Computing and Communications, BIGCOM 2019, QingDao, China, August 9-11, 2019*. 98–106. https://doi.org/10.1109/BIGCOM.2019.00024

[10] Qiang Cao, Hongyan Li, Yan Yang, Makoto Takizawa, and Naixue Xiong. 2010. A tradeoff analysis of delayed reconstruction for storage clusters. In *Proceedings of the International Wireless Communications and Mobile Computing Conference (IWCMC)*. 834–838. https://doi.org/10.1145/1815396.1815588

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 209–223. https://dl.acm.org/doi/10.5555/3386691.3386712

[12] Abhishek Chanda, Shubham Kaushik, Artem Lavrov, and Subhadeep Sarkar. 2026. TexBench: A Unified Benchmarking Suite for Shifting Workloads. *Proceedings of the VLDB Endowment* 19, 12 (2026), 4730–4733. https://www.vldb.org/pvldb/vol19/p4730-sarkar.pdf

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218. http://dl.acm.org/citation.cfm?id=1267308.1267323

[14] CockroachDB. 2021. CockroachDB. *https://github.com/cockroachdb/cockroach* (2021).

[15] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[16] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. https://doi.org/10.1145/3035918.3064054

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[19] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. http://doi.acm.org/10.1145/3183713.3196927

[20] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 449–466. https://dl.acm.org/doi/10.1145/3299869.3319903

[21] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 365–378. https://doi.org/10.1145/3448016.3457273

[22] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084. https://www.vldb.org/pvldb/vol15/p3071-dayan.pdf

[23] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. https://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[24] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 26:1–26:32. https://doi.org/10.1145/3483840

[25] Facebook. 2021. MemTable. *https://github.com/facebook/rocksdb/wiki/MemTable* (2021).

[26] Facebook. 2023. MyRocks. *http://myrocks.io/* (2023).

[27] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).

[28] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).

[29] Pat Helland. 2016. Immutability Changes Everything. *Commun. ACM* 59, 1 (2016), 64–70. http://doi.acm.org/10.1145/2844112

[30] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 857–872. http://doi.acm.org/10.1145/3183713.3196911

[31] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 651–665. https://dl.acm.org/doi/10.1145/3299869.3314041

[32] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 749–763. https://doi.org/10.1145/3448016.3457297

[33] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2022. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1605–1618. https://doi.org/10.14778/3529337.3529345

[34] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24. https://doi.org/10.1007/s00778-023-00826-9

[35] Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2667–2672. https://dl.acm.org/doi/10.1145/3318464.3383133

[36] Shubham Kaushik, Manos Athanassoulis, and Subhadeep Sarkar. 2026. RangeReduce: Query-Driven LSM Compactions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 2601–2614. https://doi.org/10.1109/ICDE65706.2026.00194

[37] Shubham Kaushik, Abhishek Chanda, and Subhadeep Sarkar. 2026. TexBench: Harnessing LLMs for Efficient Key-Value Benchmarking. In *Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC)*.

[38] Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights & Implications. In *Proccedings of the International Workshop on Testing Database Systems (DBTest)*. 23–29. https://doi.org/10.1145/3662165.3662766

[39] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 219–233. https://www.usenix.org/conference/atc20/presentation/keahey

[40] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, Chen Luo, Ian Maxon, and Pouria Pirzadeh. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2799

[41] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 3:1–3:8. https://doi.org/10.1145/2933349.2933352

[42] Chen Luo. 2020. Breaking Down Memory Walls in LSM-based Storage Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2817–2819. https://dl.acm.org/doi/10.1145/3318464.3384399

[43] Chen Luo and Michael J Carey. 2019. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment* 13, 4 (2019), 449–462.

[44] Chen Luo and Michael J Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://link.springer.com/article/10.1007/s00778-019-00555-y

[45] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2071–2086. https://dl.acm.org/doi/10.1145/3318464.3389731

[46] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. 2015. CliffGuard: A Principled Framework for Finding Robust Database Designs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1167–1182.

http://doi.acm.org/10.1145/2723372.2749454

[47] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal* 29, 1 (2020), 569–591. https://doi.org/10.1007/s00778-019-00580-x

[48] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://dl.acm.org/doi/10.1007/s002360050048

[49] Alexander H Ott, Shubham Kaushik, Boao Chen, and Subhadeep Sarkar. 2025. Tectonic: Bridging Synthetic and Real-World Workloads for Key-Value Benchmarking. In *Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC)*, Vol. 16261. 69–88. https://doi.org/10.1007/978-3-032-18070-4_5

[50] Aneesh Raman, Konstantinos Karatsenidis, Shaolin Xie, Matthaios Olma, Subhadeep Sarkar, and Manos Athanassoulis. 2025. QuIT your B^+-tree for the Quick Insertion Tree. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 451–463. https://doi.org/10.48786/edbt.2025.36

[51] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for Near-Sorted Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1475–1488. https://dl.acm.org/doi/10.14778/3827998.3828099

[52] RocksDB. 2020. Universal Compaction. *https://github.com/facebook/rocksdb/wiki/Universal-Compaction* (2020).

[53] Rockset. 2024. Rockset. *https://rockset.com* (2024).

[54] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2489–2497. https://dl.acm.org/doi/10.1145/3514221.3522563

[55] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432. https://dl.acm.org/doi/10.1145/3514221.3520169

[56] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM Design Space and its Read Optimizations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3578–3684. https://doi.org/10.1109/ICDE55515.2023.00273

[57] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757

[58] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2023. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems (TODS)* 48, 3 (2023), 8:1–8:40. https://doi.org/10.1145/3599724

[59] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. http://vldb.org/pvldb/vol14/p2216-sarkar.pdf

[60] ScyllaDB. 2024. Online reference. (2024). https://www.scylladb.com/

[61] Speedb. Accessed: November 6, 2026. Online reference. https://github.com/speedb-io/speedb.

[62] Kapil Vaidya, Tim Kraska, Subarna Chatterjee, Eric R Knorr, Michael Mitzenmacher, and Stratos Idreos. 2022. SNARF: A Learning-Enhanced Range Filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644. https://www.vldb.org/pvldb/vol15/p1632-vaidya.pdf

[63] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 141. https://doi.org/10.1145/3654944

[64] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 124:1–124:28. https://dl.acm.org/doi/10.1145/3654927

[65] Weiping Yu, Siqiang Luo, Zihao Yu, and Gao Cong. 2024. CAMAL: Optimizing LSM-trees via Active Learning. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 4 (2024), 202:1–202:26. https://doi.org/10.1145/3677138

[66] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*. https://www.usenix.org/conference/hotstorage18/presentation/zhang

[67] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 51–64. https://www.usenix.org/conference/fast21/presentation/zhong

[68] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 1:1–1:10. https://dl.acm.org/doi/10.1145/3465998.3466002

[69] Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest)*. 9–15. https://doi.org/10.1145/3662165.3662765

[70] Zichen Zhu, Subhadeep Sarkar, and Manos Athanassoulis. 2023. Acheron: Persisting Tombstones in LSM Engines. In *Companion of the International Conference on Management of Data (SIGMOD)*. 131–134. https://dl.acm.org/doi/10.1145/3555041.3589719