Anatomy of the LSM Memory Buffer: Insights & Implications

Shubham Kaushik
kaushiks@brandeis.edu
Brandeis University

Subhadeep Sarkar
subhadeep@brandeis.edu
Brandeis University

ABSTRACT

Log-structured merge (LSM) tree is an ingestion-optimized data structure that is widely used in modern NoSQL key-value stores. To support high throughput for writes, LSM-trees maintain an in-memory buffer that absorbs the incoming entries before writing them to slower secondary storage. We point out that the choice of the data structure and implementation of the memory buffer has a significant impact on the overall performance of LSM-based storage engines. In fact, even with the same implementation of the buffer, the performance of a storage engine can vary by up to several orders of magnitude if there is a shift in the input workload.

In this paper, we benchmark the performance of LSM-based storage engines with different memory buffer implementations and under different workload characteristics. We experiment with four buffer implementations, namely, (i) *vector*, (ii) *skip-list*, (iii) *hash skip-list*, and (iv) *hash linked-list*, and for each implementation, we vary any design choices (such as bucket count in a hash skip-list and prefix length in a hash linked-list). We present a comprehensive performance benchmark for each buffer configuration, and highlight how the relative performance of the different buffer implementations varies with a shift in input workload. Lastly, we present a guideline for selecting the appropriate buffer implementation for a given workload and performance goal.

ACM Reference Format:
Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights & Implications. In *International Workshop on Testing Database Systems (DBTest '24), June 9, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3662165.3662766

1 INTRODUCTION

LSM-Based Key-Value Stores. Log-structured merge (LSM) trees are widely used in the storage engines of modern NoSQL key-value stores [18, 20, 21]. LSM-tree is a highly ingestion-optimized, out-of-place data structure that stores data on disk as a hierarchical collection of *immutable sorted runs* [22, 23]. Immutability in the file structure means updates and deletes in LSM-based storage engines are always realized out of place, by inserting new entries (or meta-entries, in case of deletes) that logically invalidate the older target entries [6, 25, 26, 30]. By avoiding in-place updates and deletes LSM-engines are able to offer superior ingestion performance. To facilitate fast queries, commercial LSM-engines often employ auxiliary data structures, such as Bloom filters and fence pointers, that reduces the overall number of accesses to slower disks [10, 24]. Due to these advantages, a large number of commercial NoSQL systems, including LevelDB [16] and BigTable [8] at Google, RocksDB [14] at Meta, X-Engine [17] at Alibaba, WiredTiger [28] at MongoDB, CockroachDB [9] at Cockroach Labs, and AsterixDB [1], Cassandra [3], HBase [4], and Accumulo [2] at Apache, have adopted LSM-trees to implement their storage layer.

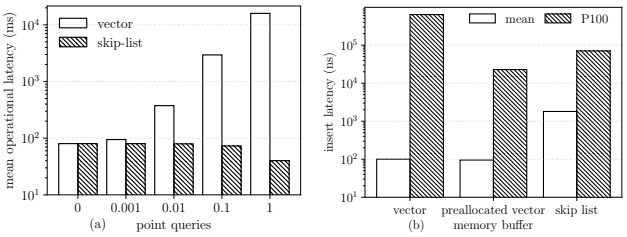

Fig. 1: (a) An increase in number of point queries affects the latency of other operations significantly in a *vector* buffer. (b) Pre-allocated *vector* offers better insert performance than dynamically allocated *vector* and *skip-list*.

LSM-Buffer and its Implementation. To ensure high throughput for inserts and to avoid writing data eagerly to slower secondary storage, LSM-trees buffer the incoming entries first in memory [19, 23]. The buffer in main memory acts as the primary storage component of an LSM-tree and has a predetermined capacity (typically, 1MB-128MB) [7, 11, 21]. Once full, entries in the buffer are sorted on the key and are written to the first level of a hierarchical collection of immutable sorted runs on disk. We refer to this process of persisting the contents of the buffer to disk as *flush* [22]. Buffering inserts in memory allows LSM-engines to amortize the number of disk I/Os performed while inserting data and makes them optimized for ingestion [27]. In practice, systems use different data structures with widely varying implementations to implement the memory buffer. Vector, skip-list, linked-list, and hybrid data structures, such as hash skip-list and hash linked-list, are some of the commonly used data structures in commercial LSM-based storage engines [12, 13]. Each data structure brings a different tradeoff between performance and memory footprint [5].

Challenge: Determining the "Best" Buffer Implementation. We point out that despite being a disk-based data structure, **the choice of data structure for the memory buffer has huge implications on the overall performance** of an LSM-engine. We highlight two scenarios where the choice of the buffer data structure and its implementation has critical performance implications. (A) For the same buffer data structure, the performance numbers may vary significantly with the *slightest shift in the workload characteristics*. In Fig. 1(a), we show that the average operational latency for an LSM-engine with a (static) vector buffer-implementation increases by 36× if the input workload shifts from insert-only to having only 0.1% point queries. (B) For the same data structure,

DBTest '24, June 9, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0669-1/24/06
https://doi.org/10.1145/3662165.3662766

different implementations and tunings also has a significant impact on performance. For example, as shown in Fig. 1(b), the tail latency ($P100$) of a dynamically allocated vector can be almost more than one order of magnitude higher than that of a pre-allocated vector.

The choice and implementation of the memory buffer in state-of-the-art LSM-engines is agnostic of workload and performance goal which leads to sub-optimal performance overall.

Benchmarking LSM-Buffers. To this end, in this work, we benchmark the performance of LSM-based storage engines with different implementation and tuning of the memory buffer. We take four buffer implementations: (i) *vector*, (ii) *skip-list*, (iii) *hash skip-list*, and (iv) *hash linked-list*, commonly used in LSM-engines for the benchmark. For the *vector* implementation, we perform experiments with both (v) fixed and (vi) dynamically allocated memory space. For *hash skip-list* and *hash linked-list*, we vary (vii) the number of hash-buckets and (viii) the length of the key-prefix used for hashing. In addition, we perform experiments by varying the workload characteristics to highlight the performance implication of workload shifts. The overarching objective of this endeavor is to construct a set of guidelines for researchers and practitioners that would enable to choose the appropriate buffer implementation for an LSM-engine given a workload and some performance goal.

Contributions. The contributions of our work is as follows.

(1) **Workload-Aware Performance Modeling**: We develop a benchmark[1] for the buffer implementation in LSM-engines for a diverse set of key-value workloads, showcasing the impact of buffer implementation and workload on performance.
(2) **The Memory vs. Performance Tradeoff**: In addition to the raw performance numbers on throughput for inserts and queries, and operational latency, we measure the memory footprint of every buffer implementation and discuss how different implementations navigate the memory vs. performance tradeoff.
(3) **Answer *What-If* Questions**: We consolidate the key takeaways from the experiments in form of a handbook that, given a workload and target performance, would allow us to avoid the objectively worst design choices to implement the memory buffer and answer complex *what-if* questions, such as:
   (a) What is the appropriate buffer design for an ingestion-heavy workload?
   (b) How to avoid write stalls when processing a workload with interleaved in point queries and inserts?
   (c) How to set the bucket count or the prefix length for hashing in a *hash skip-list* buffer?

2 BENCHMARK DESIGN

In this section, we discuss construction of the LSM-buffer benchmark and the evaluation methodology.

2.1 Buffer Implementation

The memory buffer in LSM-trees serve as a staging area for writes (inserts, updates, and deletes) before they are flushed to disk. Batching inserts in the main memory allows for amortizing the cost of writes. Retaining the ingested entries in the buffer also allows future point queries on the recently ingested entries to be returned directly from the buffer, without any disk access. This can significantly improve the throughput for point queries on "hot" data. Below, we discuss the design and configuration of four memory buffer implementations, namely *vector*, *skip-list*, *hash skip-list* and *hash linked-list*, that we use for the benchmark.

Vector. We experiment with two *vector* implementations of the memory buffer. The first is a *dynamic vector* that is implemented as an automatically re-sizing array. A *dynamic vector* ensures that the data in the buffer is stored in contiguous memory locations. Inserts to such a *vector* is appended to the array if there is available space. Otherwise, the array size is doubled through re-sizing before appending the new entry. While appending inserts allows for high throughput for writes, the cost of point queries becomes prohibitively high as they may need to scan the entire buffer. Production systems, using a *dynamic vector* implementation for the buffer, thus, sort the buffer every time there is a point query in the workload. The optimistic outlook here is that for future point queries that precede the next insert, the target entry can be searched using binary search. A *dynamical vector* suffers from high latency spikes resulting from (i) re-sizing of buffer or (ii) sorting the buffer contents during a point query. Alternatively, a *fixed-sized vector* requires allocating the designated memory space upfront, but is better suitable for latency-sensitive applications as it offers predictable performance with no latency spikes. The *vector* implementation has little memory overhead as it needs to maintain minimal metadata.

Skip-List. The default buffer implementation in many systems, such as RocksDB [13] and LevelDB [16], is *skip-list*. A *skip-list* is implemented as a hierarchical structure of linked-lists, where intermediate levels act as an index for the last level which stores the data in sorted order. *Skip-lists* offer logarithmic $O(\log n)$ complexity for inserts, updates, deletes, as well as point queries. However, the hierarchical structure of *skip-lists* requires maintaining additional key-pointer pairs for indexing purposes, and this adds to the metadata overhead of the data structure. For example, a *skip-list* with four levels, requires a memory of roughly 1.33× of the data size.

Hash Skip-List. A memory buffer implemented as a *hash skip-list* combines a hash table with *skip-lists* within each bucket. The keys to be inserted are partitioned on their prefix (of a predetermined length) for bucket allocation, facilitating targeted insertions and point queries through a combination of hashing and binary search. This structure offers superior performance for ingestion, point queries, as well as, range queries with very small selectivity, as the operations are typically limited to only a specific hash-bucket. Range queries with large selectivity, however, may span across multiple hash-buckets and, therefore, may have a significantly higher latency. The design of the *hash skip-list* involves configuring the number of buckets (H) and specifying a prefix extractor, typically the first X characters of the key. The role of the prefix extractor is to isolate the prefix from the key and hash it to the appropriate bucket. Each bucket, in turn, is a *skip-list* that contains the key-value pairs. The entire key (including the prefix) is stored in the *skip-list* as we need to facilitate range queries. The point queries are realized by simply hashing the prefix to locate the target bucket and then by searching through *skip-list*. The memory overhead of the *hash*

[1] Our code is available at https://github.com/SSD-Brandeis/LSMMemoryProfiling.

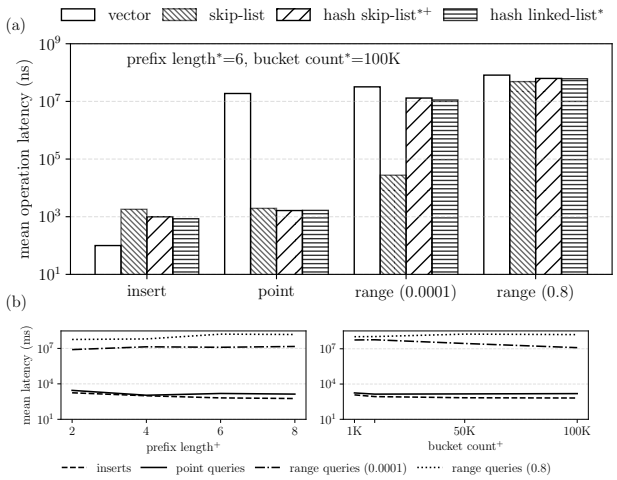

Fig. 2: Different buffer implementations offer varying performances tradeoffs. (a) Vector is not good for point and range queries but best for insert-only workloads. (b) The performance of the hash-based buffer implementations depends on prefix length and number of buckets.

skip-list includes the additional space needed to maintain the hash table and the pointers to each *skip-list*. Due to additional memory overhead, for same memory budget, a *hash skip-list* can buffer fewer key values compared to *skip-list* and *vector* before it is saturated.

Hash Linked-List. Similarly to the *hash skip-list*, the *hash linked-list* also uses a hash table framework. However, within each hash-bucket the entries are stored as a sorted singly *linked-list*. This design is particularly tailored for efficiently handling range queries on specific key prefixes. It also offers a similar ingestion performance as the *hash skip-list* with less memory overhead. For point queries, the *hash linked-list* relies on linear search within each *linked-list*, which leads to a higher query latency. The *hash linked-list* has a lower memory overhead compared to the *hash skip-list*, primarily because it eschews the layered structure inherent to *skip-lists*.

2.2 Evaluation Metric

The choice of write buffer and its implementation have critical implications on the performance of LSM-engines. Below, we discuss the key metrics along which we compare the buffer implementations.

- **Memory footprint** quantifies the overall memory required by each buffer implementation to store a given number of entries, including all corresponding metadata.
- **Reads latency** specifically captures the point and range query performance for entries in the buffer. The goal is to capture the impact of the buffer implementation on in-memory reads.
- **Writes latency** measures how fast data is written to the write buffer. This does not include flush of compaction overheads.
- **Flush** and **compaction counts** refer to the frequency of flushing the buffer content to the disk and the frequency of compacting the data on disk to create fewer but longer sorted runs, respectively.

The different workload characteristics may have different resource requirements in terms of memory, CPU, and I/O. The write buffer type also contributes to the performance of the system. Fig. 2(a) shows a comparison of different buffer implementations for four types of workloads (i) insert only, (ii) insert with point queries, (iii) insert with small-range queries, (iv) insert with long-range queries. We observe that *vector* performs best for an insert-only workload, but can become a pain-point for the point and range queries. Similarly, a buffer implemented as a *hash skip-list* and the *hash linked-list* realizes inserts and point queries better than a *vector* and a *skip-list*, but the performance is heavily dependent on the prefix length and the number of buckets. Fig. 2(b) shows how the mean query latency for a *hash skip-list* buffer changes with the prefix lengths and the number of hash buckets. Our goal is to provide a detailed comparative analysis of how various write buffer designs affect the overall performance of LSM-based systems.

3 EXPERIMENTAL EVALUATION

To ensure a comprehensive evaluation, we have selected three classes of workloads, each designed to test the write buffers under different scenarios: (a) inserts with interleaved point queries, (b) inserts with interleaved range queries (small selectivity), and (c) inserts with interleaved range queries (large selectivity).

3.1 Experimental Setup

We run the experiments on a virtual machine equipped with 2 Intel Gold 6126 vCPUs at 2.60GHz, 192GB of DDR4 RAM, a 22MB L3 cache, and 240GB of SSD storage, running Ubuntu 20.04 LTS. The implementation is done on RocksDB (v9.0.0) which is a widely used, open source LSM-based storage engine [15]. To capture the raw performance, we configure RocksDB to use only one thread for compactions and one thread for flushes. We also disable the write-ahead log (WAL) to ensure that the performance is not affected by the WAL. Note that enabling WAL would not change the trends for any of the experiments. Unless otherwise mentioned, the size of a key-value pair is 64B and the size of each disk page is 4KB. The size of the buffer is 16MB. The tree has a size ratio of 4, i.e., each level on disk is 4× larger than the previous level. N is the total number of records in the workload. The generate the experimental workload using the KVBench generator [29].

3.2 Memory Footprint

Different buffer implementations may have a widely varying memory footprint when storing the same amount of data. In this set of experiments, we reveal the trade-off between the memory footprint and the storage capacity of different write buffer implementations.

Setup: We experiment by inserting 1.5M entries into the memory buffer for every implementation and measure the memory footprint. For *hash skip-list* and *hash linked-list*, the bucket count (H) and the prefix length (X) are varied from 10 to 1000 and 2 to 10, respectively.

Hybrid Buffer Implementations have Low Memory Utilization. Fig. 3(a) and (b) demonstrates that the *vector* exhibits the lowest memory footprint due to its contiguous storage and indexing is done implicitly without additional metadata overhead. A *skip-list* requires more memory than a *vector* as its hierarchical structure requires maintaining within and across the levels. A *hash skip-list*, on the other hand, has a higher memory footprint than a *skip-list* because of the additional overhead of maintaining a hash

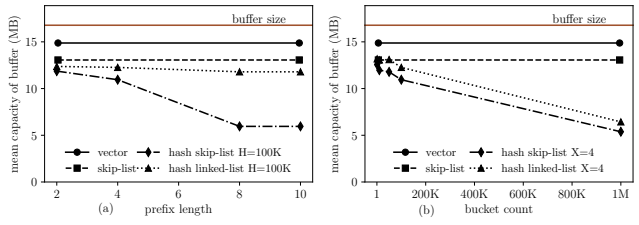

Fig. 3: Increase in (a) prefix length or (b) bucket count reduces the number of entries that the hash-based hybrid memory buffer can store, due to the overhead of pointers.

table and any necessary pointers. The memory footprint of *skip-list* and *hash linked-list* for prefix length of 2 (Fig. 3(a)) and bucket count of 1 (Fig. 3(b)) are close to that of *skip-list*. However, as the prefix length or bucket count increases, the metadata overhead for the two hybrid hash data structures shoot up, which means the buffers can store fewer entries before they are full.

3.3 Read Performance

Point queries in LSM-engines return the latest entries with a matching key, if found. The buffer implementation and tuning affects the point query performance in memory significantly. For instance, a *vector* buffer has a time complexity of $O(n)$, where n is the number of entries in the buffer, for point queries as the unsorted buffer needs to be scanned linearly. An alternative is to sort the buffer and perform a binary search. This reduces the search time to $O(\log n)$, but adds an overhead of $O(n \log n)$ for sorting. A *skip-list* serves point queries with a time complexity of $O(\log n)$ by leveraging its hierarchical structure. *Hash skip-list* and *hash skip-list* offers the best performance for point queries if the prefix length (X) and the number of buckets (H) are tuned appropriately. At the two extremes: (i) setting the prefix length too small (say, to 0) results in all keys being stored in a single bucket; (ii) increasing the prefix length to the maximum length of the keys while maintaining only one bucket leads to the same problem. Ideally, the number of buckets should 256^X (assuming character-based prefix) as this would mean keys with the same X-bit prefix are mapped to the same bucket. In fact, the hybrid hash data structures can achieve the theoretical lower bound for the point query time complexity (i.e., $O(1)$) when X is set to the maximum key length, we have 256^X buckets in memory. However, this does not scale in practice as the memory requirement increases exponentially with the prefix length (e.g., 65K for $X = 2$ and about 16M for $X = 3$.

Setup: For this set of experiments, we insert 140K entries of length 64B, with 200 point queries randomly interleaved, and we measure the time taken to execute each query. In Fig. 4, the first row of plots compares the point query latency of a *skip-list* buffer implementation (the default choice in RocksDB [13]) and the other buffer implementations. The second row of plots shows how *prefix length* and *bucket count* affects the point query performance for a *hash skip-list* and a *skip-list*. The point queries begin after 75% of the inserts have been completed. The queries are uniformly interleaved with the remaining operations.

Point Queries are Prohibitively Costly Without Indexes. Fig. 4(a) shows that the implicit indexes of a *vector* do not benefit point queries. In fact, for interleaved queries, the point query cost can be up to 12× higher than that of a *skip-list*. This is because, the read path in RocksDB requires sorting the entries in the buffer before performing a binary search. We also observe that as the buffer becomes fuller, the cost of sorting and searching increases linearly. The spikes in the plot reflect the cost of sorting the buffer before the binary search. The sort function of the standard library of C++ uses introspective sorting by doing a selection of algorithms dynamically to optimize performance. It begins with *quicksort*, switches to *heapsort* when the recursion depth exceeds a level based on the number of elements being sorted, and finally, switches to *insertion sort* when the number of elements is below a threshold. Fig. 4(b) and (c) demonstrate that *hash skip-list* (for $X = 6$ and $H = 100$K) exhibits the best performance for point queries, while the *hash linked-list* performs better than the *vector* and the *skip-list*. Fig. 4(d)-(f) illustrate that a bad choice of prefix length and bucket count can lead to suboptimal performance for point queries for hash-based hybrid buffer implementations.

Range queries in LSM-trees return the latest versions of all keys within a given range. During a range query, all entries in the buffer within the query range must be fetched, merged with the data from disk to eliminate any stale data, and returned as part of the query result. Due to the lack of inherent ordering, a *vector* buffer must scan through all entries to locate the qualifying entries. This adds a significant overhead to the range query cost. In contrast, data structures with explicit indexes, such as *skip-list* and the hybrid hash-based data structures can efficiently filter out the non-qualifying entries reducing the cost of range queries.

In particular, *hash skip-list* and *hash linked-list* are designed to optimize range queries by leveraging the $O(1)$ time complexity of a hash table to locate the correct hash bucket. This performance comes with a restriction of the entire range having the same prefix. The range query path is as follows: (i) locate the correct hash bucket, (ii) perform a seek operation in the bucket to locate the target or a key greater than target, and (iii) scan the bucket and return the keys that are within the range. If a range query spans over multiple prefixes, a *hash skip-list* or a *hash linked-list* will perform a copy and sort operations to find all the keys within the range. This requires iterating over all qualifying buckets and perform a seek operation to locate the keys. This may happen if the prefix is large enough (let's say 3, the prefix sample space will be 256^3) and the number of buckets is less than that (say, 10K). Now each bucket will have keys from $256^3/10K \approx 1667$ prefixes on average. The performance depends on the number of buckets qualifying for a given range and the number of keys it needs to seek over to construct the result set.

Setup: We run the experiments with different selectivity and record the time taken to execute each range query. We hand-picked values for X and H based on our observations from the point queries experiments. We observed two extreme cases (i) a small prefix with a large bucket count and (ii) a long prefix with a small bucket count. In both cases, the performance is suboptimal as in both cases, multiple prefixes will be mapped to the same bucket, increasing the cost of search within every bucket.

Hash-Based Hybrid Buffers Dominate the Range Query Space. Fig. 5 shows the comparison of range queries performance for different buffer implementations for selectivity 0.0001, 0.02, and 0.4 in

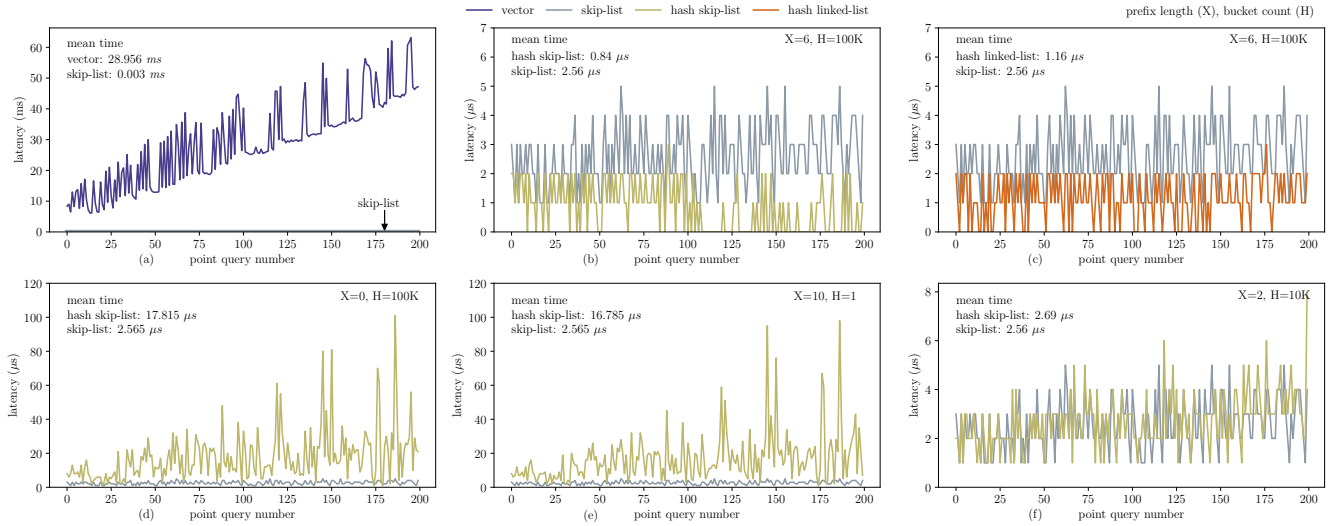

Fig. 4: A *vector* implementation exhibits the poorest performance with respect to the ingestion, while the *hash skip-list* (for *X*=6 and *H*=100K) demonstrates better performance for point queries.

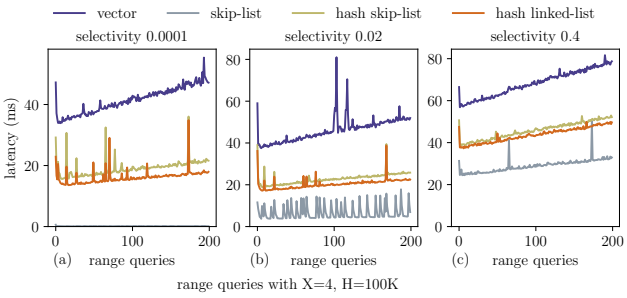

Fig. 5: The sorting of buffer is the main slowdown for the *vector* implementation. The *skip-list* outperforms it for both short and long range queries. The *hash skip-list* and *hash linked-list* also shows better performance than *vector*.

Fig. 5(a), (b), and (c), respectively. The *vector* implementation shows the poorest performance for both short and long-range queries as regardless of the selectivity, it has to scan the entire buffer. This explains the peaks and the associated high latency observed in Fig. 5(a). For each range query, the vector first sorts the data in memory and then performs a binary search to locate the starting key. For *hash skip-list* and *hash linked-list*, the increase in the prefix length with the same number of buckets for range queries with low selectivity leads to poor performance if the start and end of the range has different prefixes. This is because the filtering overhead dominates the cost of range queries and we end up processing a large amount of data to filter out the qualifying few. For the *skip-list*, the peaks arise from accessing different sets of keys during a range query. For *hash skip-list* and *linked-list*, the spikes occur when RocksDB creates a temporary *skip-list* view or snapshot of the data within the memory buffer before executing a range query.

3.4 Ingestion Performance

Setup: To measure the ingestion performance of the different memory buffer implementations, we run a set of experiments in which we insert 140K entries of 64B, and record the time taken to realize each insert. Note that to quantify the impact of the buffer implementation on inserts, we only measure the time taken to insert an entry into the buffer. The analysis of writing the data from memory to disk through flush and compaction in discussed in Section 3.5.

A Vector Buffer Offers the Best Ingestion Performance. Fig. 1(b) illustrates the comparison of write performance across the different write buffer implementations. The ingestion process for a *vector* write buffer is analogous to appending bytes to an array. A typical vector implementation dynamically allocates memory, doubling the allocation when the existing space is exhausted. This can lead to poor performance due to the copying of the buffer content every time the memory is reallocated. The pre-allocation of memory serves as a remedy for this inefficiency. As depicted in Fig. 1 (b), the dynamic memory allocation in a *vector* leads to latency spikes during ingestion. In contrast, Fig. 1 shows the enhanced performance for a pre-allocated *vector*, where the memory of $N \times (E+10) \approx 11$MB was pre-allocated to the buffer. The inherent sorting nature of *skip-list* also slows down write operations as it requires $O(\log n)$ time to locate the correct insertion location for every new key whereas, a *vector* simply append the new key to the end of the array and shows a superior write performance over a *skip-list*.

Both the *hash skip-list* and the *hash linked-list* exhibit improved write performance compared to the *skip-list*. These structures capitalize on the $O(1)$ time complexity afforded by a hash table to identify the correct hash bucket. This is followed by $O(\log \frac{P \cdot B}{H})$ for searching a key within the *hash skip-list* and on average $O(\frac{P \cdot B}{H})$ in the *hash linked-list*. The improved performance (assuming uniform data distribution) can be achieved with a careful choice of X and H. The results of our experiments, as plotted in Fig. 1 do not show

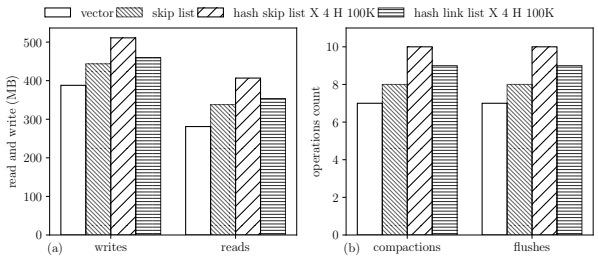

Fig. 6: (a) The write and read amplification is very high when using *hash skip-list* memory buffer, whereas the *vector* buffer has the least write and read amplification. (b) The total number of compactions and flushes is higher for *hash skip-list* memory buffer.

the insert latency. However, the latency for inserting 140K records indicates that the *skip-list* is 5× and 7× slower than the *hash skip-list* and the *hash linked-list*, respectively.

3.5 Flush & Compaction Performance

Buffer Implemetations with Explicit Indexes Flush and Compact Frequently. The rate of flushes in LSM-tree can increase the read and write amplifications, which can impact the overall performance of the system. For every flush operation, the buffer is written to the disk as a new file. The system has two choices: either 1) to create a new file and write to the first level on disk, i.e., to Level 0, or 2) read the existing (overlapping) files from Level 0 and then write the merged file back to Level 0. The first choice may lead to a larger number of smaller files at Level 0, which will increase the number of file handles that need to be managed by the system. The second choice will increase the read and write amplifications as the system needs to read the existing files, merge them with the buffer and then write the merged file(s) back to the level.

A *hash skip-list* or *hash linked-list* buffer with a large number of buckets can get full very quickly because of the hash table and pointers overhead. This leads to frequent flushes which will end up in one of the above discussed scenarios. To measure the number of flushes and compactions, we ran our experiments with unique 1M inserts and recorded the number of flushes, compactions and total number of bytes read and written to the disk. Fig. 6 shows the comparison of flushes and compactions for different write buffer implementations. The *hash skip-list* due to its large memory requirement shows almost 0.5× more flushes and 0.7× more writes. The *vector* implementation shows the least number of flushes and compactions among all the buffer implementations.

4 DISCUSSION

Choice of Buffer Implementation. Our analysis reveal that the design and implementation of memory buffer in LSM-based storage engines have significant implications on the overall performance of the system. Our results demonstrate that a *vector* buffer enhances the ingestion performance by up to 10× compared to other implementations, such as *skip-list*, *hash skip-list*, and *hash linked-list* (refer to Fig. 2). For workloads characterized by a mix of point queries and inserts, it is advised that one avoids using a *vector* buffer. This configuration tends to induce write stalls due to the necessity of sorting and binary searching within the entire buffer, compounded by the overheads of data relocation inherent to *vector* buffers. Instead, employing a *skip-list* or hash-based buffer can mitigate these issues by enhancing responsiveness and reducing overall latency. Moreover, the adoption of a pre-allocated *vector* buffer can significantly diminish performance variability, enhancing peak performance by up to two orders of magnitude for specific configurations as shown in Fig. 1(b). These observations underscore the importance of buffer configuration in tuning LSM-based storage engines for enhanced performance.

Selection of Bucket Count and Prefix Length for Hash-Based Hybrid Buffers. Setting bucket count and prefix length for hashing in a *hash skip-list* buffer requires a careful consideration to optimize performance. Factors such as the prefix length and bucket count must be tailored to the specific characteristics of the workload. A shorter prefix length may increase the risk of bucket overflows, whereas a longer prefix could lead to sparse bucket utilization and higher memory overhead. Similarly, a large bucket count can improve performance for point queries as it cuts down the bucket search times, but this has a prohibitively high memory overhead.

Storage Hardware and Memory Availability. Solid state drives (SSDs) have lower write latency and higher throughput compared to hard disk drives (HDDs). LSM-trees are designed to optimize the write performance by sequentially writing data to the memory buffer and periodically flushing it to the disk in form of immutable sorted runs. Compares to classical HDDs which rely on mechanical movement of the disk arm to write data, SSDs offer high parallelism for sequential writes by design, and therefore, offer significantly better throughput for ingestion-heavy workloads. The amount of available memory also plays a critical role in the performance of LSM-based storage engines. More memory allows for larger memory buffers, which can improve the performance of the storage engine by reducing the number of flushes to the disk. The read performance can be improved by higher cache capabilities as it can keep more data in the cache, reducing the number of disk reads. However, this often means the files on disk has a larger size which leads to a larger amount of data movement during compactions.

5 CONCLUSION

LSM-based storage engines perform well for write intensive workloads. The implementation and configuration of the memory buffers has critical implications on the performance of a storage engine. In this paper, we benchmark the performance of LSM-based storage engines with different memory buffer implementations and under different workload characteristics. We point out that an appropriate workload-aware choice and tuning of the memory buffer can improve the overall performance of the LSM-based storage engines by several orders of magnitude.

REFERENCES

[1] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.
[2] Apache. Accumulo. *https://accumulo.apache.org/*, 2023.

[3] Apache. Cassandra. *http://cassandra.apache.org*, 2023.
[4] Apache. HBase. *http://hbase.apache.org/*, 2023.
[5] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461—-466, 2016.
[6] M. Athanassoulis, S. Sarkar, T. I. Papon, Z. Zhu, and D. Staratzis. Building Deletion-Compliant Data Systems. In *IEEE Data Engineering Bulletin*, pages 21–36, 2022.
[7] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 209–223, 2020.
[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.
[9] CockroachDB. CockroachDB. *https://github.com/cockroachdb/cockroach*, 2021.
[10] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.
[11] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.
[12] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 33–49, 2021.
[13] Facebook. MemTable. *https://github.com/facebook/rocksdb/wiki/MemTable*, 2021.
[14] Facebook. MyRocks. *http://myrocks.io/*, 2023.
[15] Facebook. RocksDB. *https://github.com/facebook/rocksdb*, 2024.
[16] Google. LevelDB. *https://github.com/google/leveldb/*, 2021.
[17] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. 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*, pages 651–665, 2019.
[18] S. Idreos and M. Callaghan. Key-Value Storage Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2667–2672, 2020.
[19] C. Luo and M. J. Carey. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proceedings of the VLDB Endowment*, 12(5):531–543, 2019.
[20] C. Luo and M. J. Carey. LSM-based Storage Techniques: A Survey. *The VLDB Journal*, 29(1):393–418, 2020.
[21] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
[22] S. Sarkar and M. Athanassoulis. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2489–2497, 2022.
[23] S. Sarkar, K. Chen, Z. Zhu, and M. Athanassoulis. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2429–2432, 2022.
[24] S. Sarkar, N. Dayan, and M. Athanassoulis. The LSM Design Space and its Read Optimizations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 3578–3684, 2023.
[25] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–908, 2020.
[26] S. Sarkar, T. I. Papon, D. Staratzis, Z. Zhu, and M. Athanassoulis. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems (TODS)*, 48(3):8:1—-8:40, 2023.
[27] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment*, 14(11):2216–2229, 2021.
[28] WiredTiger. Source Code. *https://github.com/wiredtiger/wiredtiger*, 2021.
[29] Z. Zhu, A. Saha, M. Athanassoulis, and S. Sarkar. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest)*, 2024.
[30] Z. Zhu, S. Sarkar, and M. Athanassoulis. Acheron: Persisting Tombstones in LSM Engines. In *Companion of the International Conference on Management of Data (SIGMOD)*, pages 131–134, 2023.