LSM-Tree Key Value Store Literature Review

Ian Kelk
iak415@g.harvard.edu
Harvard CS265, Spring 2023

ABSTRACT

This literature review examines ten papers addressing performance issues in Log-Structured Merge-trees (LSM-trees), a widely used data structure in key-value stores such as RocksDB, LevelDB, and Cassandra. The papers are categorized into three main groups: (1) reducing tail latencies, (2) optimizing compaction, and (3) enhancing memory and storage management. We seek to answer the central question: how can we address the critical performance bottlenecks, such as read and write amplification, tail latency, write stalls, and compaction optimization in LSM-tree based key-value stores to improve their efficiency, scalability, and consistency for modern data-intensive applications?

1 INTRODUCTION

Log-Structured Merge-trees (LSM-trees are a popular data structure used in modern key-value stores, such as RocksDB, LevelDB, Cassandra, and HBase. LSM-trees are designed to efficiently handle write-intensive workloads by using a tiered storage hierarchy and background compaction to manage data. LSM-trees partition data temporally into a series of increasingly larger levels, where data enters at the top level and is sort-merged at lower levels as more data arrives. In-memory structures like Bloom filters and fence pointers help filter queries to avoid unnecessary disk I/O. Despite their advantages, LSM-trees can exhibit performance issues, such as high tail latencies, write stalls (situations where write operations are temporarily halted or slowed down to allow the system to catch up with the ongoing compaction process), read amplification, and write amplification, which can degrade overall system performance and user experience. In response, researchers have proposed various solutions to address these issues in LSM-tree based key-value stores (LSM KVs).

LSM-trees have seen various improvements to optimize performance, including better memory allocation for in-memory components, strategic data reorganization through compactions and splits, concurrency control for concurrent queries, support for time-travel queries using timestamps, balancing CPU and I/O costs through compression techniques and data movement, adaptive indexing and layouts for optimizing performance based on workload patterns, and incorporating self-designed system ideas for tailored storage design in different workloads and hardware environments [4].

In this literature review, we discuss ten recent papers that focus on improving different aspects of LSM-tree performance. We can group these papers into three main categories:

(1) reducing tail latencies
(2) optimizing compaction
(3) enhancing memory and storage management

We present a logical order for discussing the papers, focusing on the commonalities within each group and the unique contributions of each paper. Our central question: how can we address the critical performance bottlenecks, such as read and write amplification, tail latency, write stalls, and compaction optimization, in LSM KVs to improve their efficiency, scalability, and consistency for modern data-intensive applications?

1.1 Paper for reducing tail latencies

High tail latencies are often caused by interference between client writes, flushes, and compactions. To address this issue, [1] proposes SILK+, an I/O scheduler that coordinates client load with internal operations to reduce high tail latencies while maintaining good throughput.

1.2 Papers for optimizing compaction

We cover four papers in this review that focus on improving LSM-tree compaction to reduce write stalls, write amplification, and read amplification with different techniques. BoLT [5] minimizes `fsync()` call frequency to reduce write amplification and write stalls. UniKV [11] unifies hash indexing and LSM-tree design to improve read, write, and scan performance. [7] focuses on minimizing write stalls by optimizing LSM merge schedulers and [9] introduces an on-disk compaction buffer to minimize cache invalidations and improve performance under mixed read/write workloads.

1.3 Papers for enhancing memory and storage management

We discuss five papers which propose novel memory and storage management techniques to improve LSM-tree performance. ElasticBF [6] leverages data hotness to boost read performance in large key-value stores. Accordion [2] re-applies LSM design principles to memory management, addressing issues with frequent compactions and fragmented memory layout. Chucky [3] replaces Bloom filters with a succinct Cuckoo filter to improve memory bandwidth, memory footprint, and false positive rate scalability. Lethe [8] efficiently handles delete operations in LSM-based key-value storage engines. MatrixKV [10] exploits non-volatile memory to reduce write stalls and write amplification.

2 REDUCING TAIL LATENCIES

2.1 Motivation and Challenges

Tail latency refers to the higher end of the latency distribution in a system, usually measured at specific percentiles such as the 95th, 99th, or 99.9th percentile. It represents the worst-case or longer-than-average response times experienced by a small fraction of requests or operations in a system. In the context of key-value stores and databases, tail latency can be an important performance metric, as it helps to identify the delays or bottlenecks that may impact user experience or system reliability, especially for latency-sensitive applications.

High tail latencies in LSM KVs, such as RocksDB, LevelDB, and Cassandra, can negatively impact the performance of latency-critical applications and lead to poor user experience. The root cause of these high tail latencies is the interference between client writes, flushes, and compactions. Furthermore, addressing high tail latencies is difficult due to inherent LSM KV design, bursty and variable workloads, and limitations of existing optimization techniques, which focus primarily on improving throughput. As a result, there is a need for an effective solution that balances the conflicting requirements of LSM KVs while managing the interference and coordinating client load with internal operations.

2.2 Proposed Solution

SILK+ Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads [1] proposes an I/O scheduler for LSM KVs that effectively manages interference and coordinates client load with internal operations to reduce high tail latencies while maintaining good throughput. It emphasizes the importance of addressing high tail latencies to enhance the performance of latency-critical applications, improve user experience, optimize resource utilization, ensure scalability, and maintain competitiveness in the market.

The I/O scheduler in SILK employs three key techniques:

- *Dynamic bandwidth allocation*: The I/O scheduler adjusts the bandwidth allocation between client and internal operations depending on the current load. It opportunistically allocates more bandwidth to internal operations during periods of low client load, allowing the system to perform maintenance tasks more efficiently.
- *Prioritizing critical internal operations*: The scheduler gives preference to internal operations that may block client operations. By prioritizing flushes and compactions at the lower levels of the tree, the system can prevent stalls that lead to increased latencies for client operations.
- *Preempting less critical internal operations*: The I/O scheduler allows preemption of less critical internal operations to ensure that resources are allocated to higher-priority tasks, further reducing the likelihood of latency spikes.

These techniques work together to coordinate the client load and internal operations, reducing interference and tail latency while maintaining high throughput. The proposed I/O scheduler is implemented in SILK, derived from RocksDB, and demonstrates significant improvements in tail latency without negatively impacting other performance metrics or workloads.

2.3 Limitations and assumptions

The SILK+ solution for preventing latency spikes in LSM KVs has a few limitations and assumptions. (1) Its applicability to other LSM KVs is uncertain due to the lack of evidence supporting easy adaptation. (2) The paper primarily focuses on write-heavy workloads, neglecting a comprehensive analysis of tail latency under different workload types and varying conditions. (3) Scalability and performance trade-offs associated with the proposed I/O scheduler are not thoroughly discussed, and the additional complexity introduced may result in increased overhead. (4) The assumption that latency spikes are mainly caused by bursty client loads and internal operation interference may not be universally applicable. (5) Lastly, the evaluation methodology relies on synthetic benchmarks and a single production workload, limiting the generalizability of the findings to other scenarios and applications.

3 OPTIMIZING COMPACTION

3.1 Motivation for optimizing compaction

As LSM KVs are widely adopted in modern key-value stores and NoSQL systems, improving their performance directly benefits the users and systems that rely on them. Reducing write amplification, write stall, and data barrier overhead, as highlighted in BoLT [5], can lead to better overall performance and efficiency of key-value stores. This ensures faster and more reliable data processing, which in turn can significantly benefit the applications and systems dependent on the key-value stores in areas like data analytics, real-time processing, and more.

Persistent key-value storage is a critical storage paradigm in modern data-intensive applications, including data deduplication, web search, e-commerce, social networking, and photo stores. As UniKV [11] demonstrates, improving these systems by unifying hashing indexing and LSM-tree design can have a significant impact on the read, write, and scan performance in these applications.

LSM KVs suffer from write stalls and large performance variations due to inherent mismatches between fast in-memory writes and slow background I/O operations [7]. By minimizing write stalls and stabilizing performance, this could lead to enhanced end-user experiences and reliability of systems that depend on LSM-trees.

In addition to these factors, optimizing compaction is vital for improving the performance of LSM-trees in mixed read/write workloads [9]. Existing LSM-tree based solutions tend to suffer from high miss rates and performance degradation due to cache invalidations from compactions. By reducing the interference on buffer caching from compactions, query performance can be improved under mixed read/write workloads while retaining the advantages of LSM-trees for write-intensive workloads. This optimization can contribute to enhancing the performance of data management systems under mixed workloads.

3.2 Challenges of optimizing compaction

Compactions in LSM-trees lead to issues like write amplification, write stall, and data barrier overhead [5] . Write amplification arises from compaction operations rewriting the same key-value pairs multiple times between files, hurting write throughput and causing storage wear issues. Write stall occurs when compaction fails to keep up with incoming write requests, blocking subsequent requests and further impacting write throughput. Data barrier overhead results from heavy reliance on `fsync()/fdatasync()` calls for file consistency, hurting the concurrency level of I/O requests and under-utilizing storage bandwidth.

LSM KVs struggle to efficiently handle both random accesses and range queries while retaining their benefits for write-intensive workloads [9]. Compactions not only lead to cache invalidations and high miss rates on the DB buffer cache but also cause performance degradation in mixed read/write workloads.

Designing a system that combines characteristics of hash indexing and LSM-trees is difficult due to their differences in trade-offs [11]. While hash indexing supports high read and write performance, it does not support efficient scans and has scalability concerns. Conversely, LSM KVs are more efficient in terms of supporting writes, scans, and scalability, but they suffer from high compaction and multi-level access overheads.

Finding ways to minimize write stalls resulting from the inherent mismatch between fast in-memory writes and slow background I/O operations in LSM-trees is challenging [7]. Accurately determining the maximum sustainable write rate, scheduling LSM I/O operations to minimize write stalls, and evaluating write stalls for various LSM-tree designs are all complex aspects to address in order to develop solutions that can achieve high write throughput with low performance variance.

3.3 Problems with existing solutions for optimizing compaction

Current solutions have attempted to address some of these issues, but have not been able to find an optimal balance between SSTables sizes, data barrier overheads, and other performance concerns like latency, read amplification, and scalability. There is a need for innovative approaches that can tackle these challenges while still retaining the benefits of LSM-trees for write-intensive workloads.

These existing solutions to optimize compaction in LSM-trees do not work optimally as they fail to address various interconnected issues. BoLT [5] addresses some problems, but existing key-value stores like LevelDB and RocksDB still suffer from high write amplification, write stall, data barrier overhead, and metadata caching overhead. These issues reduce write throughput, increase latency, and affect the overall performance of key-value stores.

In UniKV [11], LSM KVs experience difficulties optimizing both read and write performance without sacrificing scan performance due to large read and write amplifications. Existing optimizations make trade-offs and cannot combine hash indexing and LSM-tree efficiently, which would improve reads, writes, and scans simultaneously.

Performance stability in LSM KVs remains a problem as they suffer from write stalls and large performance variances caused by the mismatch between their fast in-memory writes and slow background I/O operations [7]. Existing solutions fail to accurately measure the maximum sustainable write rate or effectively schedule LSM I/O operations to minimize write stalls.

Finally, with mixed read/write workloads in LSbM-tree [9], existing LSM-tree solutions experience low throughput and long latency due to interference caused by compactions on buffer caching. Proposed solutions, such as Key-Value store cache, dedicated compaction servers, and Stepped-Merge algorithm, have limitations that prevent them from efficiently handling both random and range queries while retaining LSM-tree merits for write-intensive workloads.

3.4 Core intuitions for optimizing compaction presented in the papers

BoLT: Barrier-optimized LSM-Tree [5] minimizes the number of calls to `fsync()/fdatasync()` barriers while taking advantage of fine-grained SSTables in key-value stores. BoLT achieves this by decoupling SSTables from physical files, which allows multiple SSTables to be stored in a single physical file, thereby reducing the file consistency overhead. It consists of four key elements: (i) compaction file, (ii) logical SSTables, (iii) group compaction, and (iv) settled compaction. These elements work together to improve write performance, reduce write amplification and write stalls, and avoid the negative impacts of large SSTable sizes on read performance.

UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing [11] unifies the key design ideas of hash indexing and LSM-tree in a single system to simultaneously improve the performance in reads, writes, and scans for large LSM KVs. UniKV leverages data locality to differentiate the indexing management of key-value pairs.

By using hash indexing to accelerate single-key access on a small fraction of frequently accessed (i.e., hot) key-value pairs, while for the large fraction of infrequently accessed (i.e., cold) key-value pairs, the original LSM-tree-based design is still followed to provide high scan performance. Additionally, dynamic range partitioning is proposed to support very large LSM KVs and provide good scalability. By unifying hash indexing and the LSM-tree in a single system with dynamic range partitioning, UniKV aims to achieve high performance in reads, writes, and scans in large key-value stores.

On Performance Stability in LSM-based Storage Systems [7] minimizes write stalls in LSM-trees by carefully evaluating and optimizing LSM merge schedulers given an I/O bandwidth budget. The paper proposes a simple yet effective two-phase experimental approach to evaluate write stalls for various LSM-tree designs. The approach consists of a testing phase, where the maximum write throughput is measured, and a running phase, where the data arrival rate is set close to the measured maximum write throughput to evaluate its write stall behavior based on write latencies. By identifying and exploring design choices for LSM merge schedulers, the paper presents a new merge scheduler to address the challenges of accurately measuring the maximum sustainable write rate and scheduling LSM I/O operations to minimize write stalls at runtime. With proper tuning and configuration, LSM-trees can achieve both high write throughput and small performance variance.

A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads [9], adds an on-disk compaction buffer to the LSM-tree to minimize frequent cache invalidations caused by compactions. This compaction buffer directly maps to the buffer cache and maintains the frequently visited data in the underlying LSM-tree but is updated at a much lower rate than the compaction rate. The LSbM-tree (Log-Structured buffered-Merge tree) directs queries to the compaction buffer for frequently visited data that will be hit in the buffer cache and to the underlying LSM-tree for others, including long-range queries. By using a small amount of disk space as a compaction buffer, the LSbM-tree achieves high and stable performance for queries by serving frequently accessed data with effective buffer caching while retaining all the merits of LSM-tree for write-intensive workloads.

3.5 Limitations and Assumptions

Some limitations and assumptions of the four papers focused on optimizing compaction can be grouped as follows:

- *Compatibility*: Both BoLT [5] and LSbM-tree [9] do not discuss compatibility with other key-value store systems or LSM-tree implementations, which may limit their applicability.
- *Workload Assumptions*: BoLT [5] assumes fine-grained SSTables are beneficial, while UniKV [11] assumes high data access skewness and locality. These assumptions may not hold true for all workloads.
- *Memory and Metadata Caching Overhead*: Both BoLT [5] and UniKV [11] do not thoroughly address issues related to memory overhead and metadata caching, which may impact scalability and performance.
- *Performance Trade-offs and Evaluation*: All four papers have limitations in their performance evaluations and do not comprehensively consider varying workloads, system configurations, or real-world scenarios.
- *Compaction Strategies*: Both BoLT [5] and LSbM-tree [9] propose compaction strategies without providing enough details on their impact on overall system performance and management.
- *Scalability and Robustness*: BoLT [5] and LSbM-tree [9] do not discuss scalability and robustness under varying workloads and system configurations, potentially limiting their applicability in different settings.
- *Implementation Complexity*: UniKV's [11] implementation introduces higher complexity compared to traditional key-value store designs, which may impact maintainability, ease of use, and potential for future optimizations.
- *Write Throughput and Stalls*: Performance Stability [7] makes assumptions about write throughput and stalls, which may not be accurate or generalizable across different LSM-tree designs and workloads.

4 ENHANCING MEMORY AND STORAGE MANAGEMENT

4.1 Motivation for enhancing memory and storage management

Enhancing memory and storage management in LSM KVs is of vital importance, as it directly impacts read and write performance, latency, disk wear, and overall efficiency. Modern day key-value stores must contend with several interconnected issues, including read amplification, false positives in Bloom filters, sensitivity to compaction rate and extent, inefficient memory management, latency of delete operations, violation of privacy regulations, write stalls, and performance fluctuations.

The advent of SSDs has significantly reduced the performance gap between storage and memory devices, making the cost of memory access more noticeable when compared to storage access. Moreover, as data size increases, the overheads produced when querying and constructing Bloom filters lead to deteriorating performance. Key-value stores often have to handle streaming systems, which require efficient deletion of data outside a designated window. This also necessitates compliance with privacy regulations like GDPR and CCPA that require organizations to delete user data within a set timeframe. The inability to provide guarantees for delete persistence latency can result in privacy breaches.

4.2 Challenges for enhancing memory and storage management

The challenges faced when enhancing memory and storage management in LSM KVs involve various issues. One such issue rooted in the nature of LSM-trees is severe read amplification, caused by the need to check multiple SSTables and resulting in extra I/Os [6]. Handling read amplification often involves using Bloom filters, but these filters bring along their own set of complications: large memory overhead and a propensity for false positives [6]. Furthermore, access skewness exists in key-value stores, making it necessary to dynamically adjust settings for optimal performance, a challenging feat due to varying access unevenness levels and the dynamic nature of data hotness [6].

Another difficulty arises from LSM store's sensitivity to the rate and extent of compactions [2]. This sensitivity impacts both read and write performance and increases disk wear. Adding to the problems in LSM stores is inefficient memory management, which manifests as cache performance issues and garbage collection overhead as the memory store size increases [2]. The advent of SSDs has changed the storage media landscape, causing a reduction in the performance gap between storage and memory devices, effectively transforming memory access costs into something no longer negligible [3].

Compounding these challenges are the trade-offs between access and construction costs of Bloom filters, which affect both read and write performance in LSM-trees [3]. Write amplification and write stalls in LSM KVs lead to performance fluctuations, long-tail latencies, and degraded user experiences [10]. Deletes in LSM engines cause additional difficulties, increasing space amplification and consequently adversely affecting both read performance and write amplification [8]. Furthermore, delete persistence latency in LSM engines is unbounded, preventing guarantees for persistent deletion and leading to potential privacy breaches due to data retention [8].

4.3 Problems with existing solutions for enhancing memory and storage management

Existing solutions face numerous challenges, particularly due to their inability to effectively address the inherent complexities of read amplification, write amplification, and write stalls, among others. The presented papers reveal the following shortcomings in existing techniques:

- Current methods for reducing read amplification focus on the use of Bloom filters; however, they do not effectively leverage access skewness present in key-value stores. They also suffer from false positives, leading to unnecessary I/O operations and memory overhead [6].
- Approaches for mitigating compactions in LSM-trees do not address the core problems of sensitivity to compactions and

inefficient memory management, which cause performance issues and negatively impact garbage collection overhead [2].
- As the performance gap between storage and memory devices narrows (particularly with SSDs), Bloom filters become a bottleneck in LSM-trees, making current solutions inadequate in addressing the challenges of workload skew and the trade-off between access and construction costs of Bloom filters [3].
- Existing LSM engines struggle to handle delete operations efficiently, leading to increased space amplification, read cost, write amplification, and privacy concerns due to unbounded delete persistence latency [8].
- Current LSM KVs primarily focus on reducing write amplification rather than addressing write stalls. The all-to-all compaction in the L0-L1 levels of the LSM-tree consumes CPU cycles and SSD bandwidth, causing write stalls, performance fluctuations, and long-tail latencies [10].

Proposed solutions such as ElasticBF, Accordion, Chucky, Lethe, and MatrixKV aim to address these challenges by targeting different aspects of memory and storage management in LSM-trees, striving to improve performance and user experience.

4.4 Core intuitions for enhancing memory and storage management in the papers

ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores [6] is a fine-grained heterogeneous Bloom filter management scheme with dynamic adjustment according to data hotness (a measure of how frequently the key-value pair is accessed). By assigning multiple small-sized Bloom filters to each small group of key-value pairs when building SSTables, it allows for dynamic allocation of Bloom filters based on the hotness of the key-value pairs. This reduces the overall false positive rate during the whole execution process of applications while still limiting the volume of all Bloom filters, leading to improved read performance in large key-value stores.

Accordion: Better Memory Organization for LSM Key-Value Stores [2] reapplies the LSM design principles to memory management in LSM KVs. Accordion proposes organizing the memory store into two parts: (1) a small dynamic segment that absorbs writes, and (2) a sequence of static segments created from previous dynamic segments. This partitioning allows the memory store to benefit from in-memory compactions, which eliminate redundant data before it is written to disk. The benefits of Accordion include:

- Fewer compactions, which reduces the write volume and resulting disk wear.
- More keys in RAM, improving read latency.
- Reduced garbage collection overhead, as it dramatically decreases the size of the dynamic segment.
- Cache friendliness, as the flat nature of immutable indices improves locality of reference and boosts hardware cache efficiency.

Accordion proactively handles disk compaction even before data reaches the disk and addresses garbage collection overhead by directly improving the memory store structure and management. The approach increases memory utilization and reduces fragmentation, garbage collection costs, and the disk write volume, resulting in significantly improved system performance and reduced disk wear.

Chucky: A Succinct Cuckoo Filter for LSM-Tree [3] proposes a new LSM-tree filter design that replaces Bloom filters with a Cuckoo filter variant that maps each data entry to both a fingerprint and an auxiliary Level ID (LID). Chucky compresses LIDs using techniques from information theory, such as Huffman coding, to keep the FPR low and stable as data grows.

Cuckoo filters and Bloom filters are both probabilistic data structures that are used to test whether an element is a member of a set. The trade-off is that Cuckoo filters are generally more space-efficient and support item deletion, but may have a small chance of false negatives (where Bloom filters have no chance of false negatives). Chucky points out that LIDs in an LSM-tree are highly compressible due to the exponential growth of the LSM-tree. Chucky proposes that since most entries reside at larger levels, the distribution of LIDs within the Cuckoo filter is heavily skewed. Because of this, LIDs of larger levels can be encoded with fewer bits than those of smaller levels to minimize the average LID size. By saving bits from LID encoding, the saved bits can be assigned to the fingerprints to keep them large and maintain a low false positive rate (FPR). This design simultaneously scales memory bandwidth, memory footprint, and FPR while providing more efficient and robust performance in various contexts, including storage media, workload skew, LSM-tree tuning, and data size.

Lethe: A Tunable Delete-Aware LSM Engine proposes a new LSM KV which efficiently supports deletes without compromising the benefits of LSM-trees. Lethe introduces two new LSM design components: FADE and KiWi.

- FADE (Fast Deletion) is a new family of compaction strategies that prioritize files for compaction based on the number of invalidated entries contained, the age of the oldest tombstone, and the range overlap with other files. This helps in deciding when to trigger a compaction on which files, to purge invalid entries within a threshold.
- KiWi (Key Weaving Storage Layout) is a new continuum of physical layouts that allow for tunable secondary range deletes without causing latency spikes, by introducing the notion of delete tiles. KiWi augments the design of each file with several delete tiles, each containing several data pages. A delete tile is sorted on the secondary (delete) key, while each data page remains internally sorted on the sort key. KiWi facilitates secondary range deletes by dropping entire pages from the delete tiles, with a constant factor increase in false positives.

Lethe is the first LSM engine to offer efficient deletes while improving read performance, supporting user-defined delete latency thresholds, and enabling practical secondary range deletes.

MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM [10] exploits non-volatile memory (NVM) to address two limitations of LSM KVs: write stalls and write amplification. The proposed

solution aims to perform smaller and cheaper L0-L1 compactions to reduce write stalls while reducing the depth of LSM-trees to mitigate write amplification.

MatrixKV utilizes four novel techniques to achieve these goals:

- *Matrix container*: Relocating and managing the L0 level in NVM with the proposed matrix container, consisting of a receiver and a compactor.
- *Column compaction*: A fine-grained compaction between L0 and L1, compacting small key ranges at a time, which reduces write stalls by processing a limited amount of data and promptly freeing up the column in NVM for the receiver to accept data flushed from DRAM.
- *Reducing LSM-tree depth*: MatrixKV increases the size of each LSM-tree level to reduce the number of levels, reducing write amplification and delivering higher throughput.
- *Cross-row hint search*: MatrixKV introduces this technique for the matrix container to maintain adequate read performance by logically sorting all keys in the container, thus accelerating search processes.

4.5 Limitations and Assumptions

Some limitations and assumptions of the five papers focused on enhancing memory and storage management can be grouped as follows:

- *Assumption of specific access patterns or data characteristics*: ElasticBF [6], Accordion [2], and Lethe [8] assume specific access patterns or data characteristics, which might not be true for all key-value stores or use cases, potentially affecting the effectiveness of their proposed solutions.
- *Complexity*: All the proposed solutions for enhancing memory and storage management introduce various techniques and components that increase the complexity of implementation, maintenance, and optimization. This might affect the ease of adopting these solutions in production systems.
- *Scalability and applicability*: ElasticBF [6], Accordion [2], Chucky [3], and MatrixKV [10] have limitations in terms of scalability to larger memory stores, diverse workloads, or different storage configurations. This might affect the generalizability of their proposed solutions.
- *Parameter tuning and reliance on heuristics*: Accordion [2] and Chucky [3] rely on parameter tuning and heuristics, which can significantly impact the performance of their algorithms. Finding optimal values or accurately capturing compaction decisions might be challenging in practice.
- *Overhead and performance trade-offs*: ElasticBF [6], Accordion [2], Chucky [3], and Lethe [8] introduce potential overheads or trade-offs in their solutions, which might affect overall system performance. These trade-offs include CPU, memory, storage overheads, garbage collection, false positives, and increased read or write costs.
- *Dependency on specific technologies*: MatrixKV's [10] solution depends on the presence of NVM and assumes consistent performance gains with NVM. This might limit its applicability in systems without NVM or where NVM technology is not yet mature or widely adopted.

5 FUTURE WORK

As the use of LSM KVs continues to evolve and expand to a variety of applications, identifying potential areas for future work is essential to fostering innovation and maintaining the relevance of the technology. Here are a few ideas:

- Exploring new I/O scheduling and resource allocation techniques to address previously unexplored performance bottlenecks and limitations in LSM KVs, offering further optimizations in tail latency, write stalls, and read/write amplification.
- Investigating the applicability and adoption of the proposed solutions (SILK+, BoLT, UniKV, Performance Stability, LSbM-tree, ElasticBF, Accordion, Chucky, Lethe, and MatrixKV) in other LSM KV implementations or under diverse workloads and settings to determine their adaptability and generalizability.
- Conducting more comprehensive performance evaluations under varying workloads, system configurations, and real-world scenarios to better understand the trade-offs and impact of the proposed enhancements in memory and storage management, compaction optimization, and tail latency reduction.
- Analyzing the impact of advances in non-volatile memory and storage technologies on LSM KV performance, adapting existing solutions or proposing novel approaches to fully leverage the potential of these technologies for improved performance and efficiency.
- Investigating the impact of hardware advancements, such as new storage-class memory (SCM) technologies and persistent memory, on LSM KV designs, compaction schemes, and memory/storage management techniques.

6 CONCLUSION

The research papers discussed in this review highlight the ongoing efforts to address performance issues in LSM KVs. By implementing novel approaches like SILK+, BoLT, UniKV, Performance Stability, LSbM-tree, ElasticBF, Accordion, Chucky, Lethe, and MatrixKV, researchers are striving to enable more efficient, cost-effective, and consistent performance in LSM-trees while maintaining or improving performance.

We've compared the different methods and approaches presented in these papers, emphasizing their core intuitions and highlighting their particular challenges. Although many solutions attempt to tackle the performance bottlenecks in LSM-trees, none of them can universally and completely address all issues: there is no perfect design and no data structure minimizes all performance trade-offs [4]. Each paper discussed here provides a unique perspective on specific aspects, whether it be reducing read and write amplification, mitigating tail latency, optimizing compactions, or enhancing memory and storage management.

REFERENCES

[1] 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 (2020), 1 – 27.

[2] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.* 11 (2018), 1863–1875.

[3] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[4] Stratos Idreos and Mark D. Callaghan. 2020. Key-Value Storage Engines. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[5] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. 2020. BoLT: Barrier-optimized LSM-Tree. *Proceedings of the 21st International Middleware Conference* (2020).

[6] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *USENIX Annual Technical Conference*.

[7] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-based Storage Systems. *ArXiv* abs/1906.09667 (2019).

[8] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[9] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. 2018. A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads. *ACM Transactions on Storage (TOS)* 14 (2018), 1 – 26.

[10] Ting Yao, Yiwen Zhang, Ji guang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX Annual Technical Conference*.

[11] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), 313–324.