DEDUPKV: A Space-Efficient and High-Performance Key-Value Store via Fine-Grained Deduplication

Safdar Jamil
Department of Computer Science and Engineering
Sogang University
Seoul, Seoul, Republic of Korea
safdar@sogang.ac.kr

Awais Khan
Oak Ridge National Lab
Oak Ridge, TN, USA, TENNESSEE, USA
khana@ornl.gov

Xubin He
Temple University
Berwyn, Philidelphia, USA
xubin.he@temple.edu

Youngjae Kim*
Department of Computer Science and Engineering
Sogang University
Seoul, Seoul, Republic of Korea
youkim@sogang.ac.kr

Abstract

Log-Structured Merge Tree (LSM-tree) based key-value stores excel in write-intensive environments but suffer from data duplication, consuming up to 49% of storage space in LSM-tree-based key-value store deployments. Traditional solutions like compression and coarse-grained file system-level deduplication introduce overhead or have limited effectiveness. In this study, we propose DedupKV, a fine-grained deduplication framework tailored for LSM-tree, maximizing data reduction efficiency while minimizing write stalls and read overheads. DedupKV features three key innovations: (1) FLUSH-*integrated inline deduplication*, which removes duplicates during memory-to-storage writes; (2) *WAL file-based offline deduplication*, repurposing write-ahead logs to avoid double writes; and (3) *elastic execution*, dynamically balancing inline and offline deduplication based on memory pressure and workload intensity. Additionally, dynamic granularity management reduces deduplication metadata overhead. We implemented these four ideas in RocksDB for the first time and conducted experiments in a Linux environment. Our evaluation shows that WAL file-based offline deduplication and DedupKV outperform BlobDB by 33% and 23%, respectively, in write-heavy workloads, while reducing write amplification by 1.2×, 2×, and 1.6× for real KV datasets.

*Y. Kim is the corresponding author.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730424

Keywords

Log-Structured Merge Tree, Deduplication, Key-Value Store

ACM Reference Format:
Safdar Jamil, Awais Khan, Xubin He, and Youngjae Kim. 2025. DEDUPKV: A Space-Efficient and High-Performance Key-Value Store via Fine-Grained Deduplication. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3730424

1 Introduction

Log Structured Merge Tree (LSM-tree)-based Key-Value Stores (LSM-KVS) – exemplified by BigTable [7], Cassandra [22], LevelDB [24], and RocksDB [18] – have become the de facto storage engine for high-throughput applications in cloud environments and large-scale systems at companies like Facebook [21] and ByteDance [34]. Their excellent write performance has cemented their role as the backbone of modern data storage. However, the very design that enables high write throughput also introduces a critical challenge: *significant write and space amplification.* This is mainly due to the frequent compaction operations inherent in maintaining the LSM-tree structure, which becomes even more problematic when facing highly redundant user-generated data [1, 9, 32, 37, 39–42, 47, 48] – our analysis of representative workloads (see Table 1 in Section 2.3) reveals duplication ratios as high as 49%.

Techniques such as block-level compression and file system-level deduplication have attempted to mitigate redundancy [5, 9, 10, 27, 32, 36, 40, 41]. However, these techniques demonstrate limited benefits or are tailored for specific use-cases. Compression, constrained by a block-level scope, is unable

to achieve maximum space savings even for extreme redundancy datasets [45]. Meanwhile, file system-level deduplication grapples with a granularity mismatch between the deduplication chunk sizes (typically 4 KB to 128 KB) and the variable sizes of the values of key-value pairs [5, 9, 32, 40, 41]. Our experiments with ZFS deduplication (see Figure 2 in Section 3.2) confirm that – even after tuning the chunk sizes – these approaches incur significant overhead due to processing all incoming files (including Write-Ahead Log files), metadata management, and Copy-on-Write penalties.

In this study, we address the mismatch between key value sizes and the size of deduplication in the file system by proposing a solution that operates in the Key Value Store (KVS) layer. Incorporating key-value semantics directly into file systems would be impractical, as file systems inherently rely on fixed-sized blocks. However, deduplication with LSM-KVS has never been attempted, which makes its design inherently complex. Therefore, we identify two key challenges.

- **Balancing Inline vs. Offline Deduplication**: While inline deduplication is known to effectively reduce write amplification by only writing the unique data, it intensifies the existing LSM-KVS write stall problem by adding deduplication steps to the critical write path. In contrast, traditional offline deduplication suffers from the *"double-write problem"* [26, 50].
- **Reducing Deduplication Metadata Overhead**: Minimizing the substantial deduplication metadata overhead of fine-grained key-value-level deduplication is crucial – while smaller granularity of deduplication can greatly increase efficiency, it also increases memory usage and metadata access costs.

To address these challenges, we systematically investigate and propose a novel design enabling deduplication for LSM-KVS, called DedupKV. To our knowledge, DedupKV is the first *comprehensive* approach to integrate deduplication into LSM-KVS. By rethinking deduplication at the LSM-KVS layer, DedupKV significantly reduces write amplification and enhances storage efficiency, laying the groundwork for more scalable and cost-effective key value stores.

DedupKV specifically addresses the question of *when and how* to detect duplicates – whether inline or offline – through three interwoven components. First, **FLUSH-Integrated Inline Deduplication** incorporates deduplication into the FLUSH operation, harnessing underutilized background threads to remove duplicates before they reach storage. Although this reduces the level of write amplification, it can prolong the FLUSH phase under high memory pressure. Second, **WAL File-based Offline Deduplication** shifts duplicate detection into existing Write-Ahead Log (WAL) files, eliminating the "double-write" issue often seen in traditional offline deduplication approaches and alleviating FLUSH-induced write stalls. Finally, DedupKV uses **Elastic Execution** to dynamically toggle or overlap these two approaches, based on memory utilization and write load. This adaptability leverages the advantages of inline deduplication when memory is abundant, while using offline deduplication to prevent extended write stalls under heavier write demands.

In addition to deciding between inline and offline deduplication, DedupKV also addresses the second challenge of mitigating the high metadata overhead inherent in fine-grained deduplication. To balance space efficiency and duplicate detection, DedupKV adopts a hybrid framework called **Dynamic Granularity Deduplication (DGD)**, which selects different techniques based on the size of the value. For a value larger than a certain threshold, it uses a hash-based exact deduplication scheme; for smaller values, it applies lightweight *LZ4 compression*. This adaptive approach keeps duplicate lookup and deduplication metadata overhead low while achieving substantial reduction in write and space amplification. Furthermore, a *tiered metadata storage system* offloads cold data to LSM-tree, reducing the footprint in memory without sacrificing performance. By holistically integrating both inline/offline deduplication decisions and a size-aware, hybrid deduplication scheme, DedupKV delivers a scalable, efficient deduplication solution tailored for modern write-intensive workloads.

We implemented DedupKV on BlobDB [19] and conducted extensive evaluation using real key-value datasets, real-world key-value traces, and the YCSB benchmark [15]. Our results show that, in isolation, FLUSH-integrated inline deduplication can lead up to a 2× performance drop on certain datasets (e.g., Mails workload); however, when integrated with DedupKV, our design recovers this loss and achieves a 23% improvement over the baseline BlobDB under write-only scenarios. Moreover, DedupKV consistently reduces write amplification in proportion to the redundancy of the dataset and maintains robust performance across diverse access patterns, including range scans.

In summary, our key contributions are as follows:

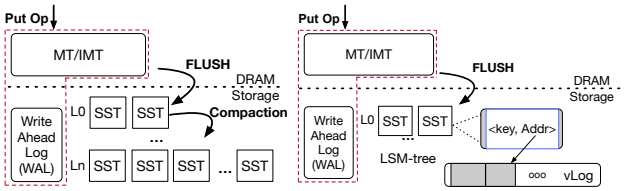
- We experimentally demonstrate that conventional file system-level deduplication is unsuitable for key-value stores due to granularity mismatches, with duplication ratios reaching nearly 49% in typical workloads.
- We propose DedupKV, a novel deduplication-enabled design for LSM-tree-based key-value stores that integrates flush-integrated inline deduplication, WAL file-based offline deduplication, and an elastic execution to allow concurrent execution of inline and offline deduplication to maintain performance characteristics and dramatically reduce write amplification.
- We implemented DedupKV on BlobDB and evaluated its impact on write performance and write amplification.

WAL file-based offline deduplication and DedupKV improve write performance by 33% and 23%, respectively, in write-heavy workloads, while reducing write amplification by 1.2× and 2×.

2 Background and Related Work

2.1 LSM-Tree

An LSM-tree consists of memory and storage components optimized for write-heavy workloads. The memory layer includes a MemTable (MT) and Immutable MemTable (IMT) to buffer incoming writes, while the storage layer comprises Sorted String Table (SSTables) and Write-Ahead Log (WAL) files, as shown in Figure 1.



(a) Traditional LSM-tree (b) Key-value separation

Figure 1: A high-level overview of two LSM-tree designs in state-of-the-art RocksDB.

The FLUSH operation moves data from IMT to SSTables, ensuring persistence. LSM-trees follow two primary designs: (i) traditional LSM-trees, where keys and values are stored together in SSTables, and (ii) KV-separated LSM-trees, where values are stored in a separate value-log (vLog) to reduce write amplification. Figure 1(a) illustrates RocksDB's traditional design while Figure 1(b) depicts the KV-separated approach. The latter, adopted by WiscKey [35], BlobDB [19], and LavaStore [43], minimizes LSM-tree compaction overhead. This study focuses on KV-separated LSM-trees.

Lifespan of WAL files: LSM-tree rely on WAL files as a primary mechanism to ensure crash consistency for in-memory data. In an LSM-tree, every write operation is first recorded in a WAL file before being applied to the MemTable. This ensures that, in the event of a system failure, the database can recover by replaying these log entries. The typical lifecycle of a WAL file is as follows:

- A new WAL file is created each time a new MemTable is created.
- The WAL file continuously logs incoming write requests until the MemTable or the WAL file itself reaches a predefined size threshold.
- Upon reaching this threshold, the MemTable is marked as immutable (IMT), and the corresponding WAL file is closed.
- During the FLUSH operation, the IMT is written to persistent storage as SSTables.
- Once the FLUSH completes and the data is safely stored, both the IMT and its associated WAL files are removed from the system.

Notably, unlike other components that require periodic garbage collection, WAL files are simply deleted after their contents have been flushed to disk.

2.2 Storage Space Optimization Techniques

As datasets grow and NVMe capacity remains limited, KVS must adopt storage optimizations like compression and deduplication to improve space efficiency.

Compression reduces data size using encoding techniques before writing to storage, minimizing storage footprint and write amplification. It is widely adopted in KVS, with BlobDB and RocksDB supporting algorithms such as Snappy [23], ZSTD [14], LZ4 [12], and LZ4HC [13]. These operate at the block or file level during compaction, compressing individual SSTable blocks. However, compression remains localized, failing to eliminate cross-block and cross-file redundancies, leading to suboptimal storage efficiency [45].

Deduplication, on the other hand, eliminates redundant data by storing only unique chunks, making it essential in archival and primary storage, including file systems [2, 6, 31, 32], databases [38, 45], and distributed storage [25, 29, 30]. Deduplication typically consists of (1) chunking, (2) fingerprinting, (3) duplicate detection, and (4) metadata population for unique chunks. It can be classified as inline (performed during I/O [42]) or offline (executed post-write [32]), with offline deduplication avoiding performance degradation in the critical I/O path.

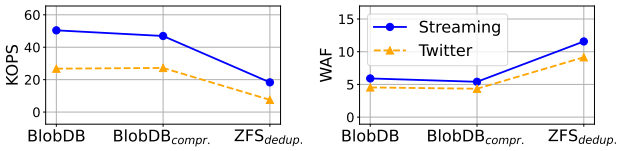
Despite its success in file systems, *deduplication remains underexplored in LSM-based KVS*. The primary challenge is that KV pairs in LSM-KVS are much smaller than typical deduplication chunk sizes (4 KB - 128 KB), making conventional deduplication techniques difficult to apply efficiently.

2.3 Key-Value Dataset Characteristics

KV datasets exhibit a wide range of characteristics, including variable key sizes, diverse value sizes, and differing access patterns across applications [8, 20, 33]. As demonstrated by Cao et al. [5], typical key sizes at Facebook range from 16 to 128 bytes, while value sizes span from tens of bytes to several kilobytes.

Building on this analysis, we expand our scope to include datasets from three distinct applications:

- **Twitter:** Here, KVS serve as the default store engine, managing billions of KV pairs (tweets) with highly variable content sizes.
- **Streaming Platform:** KV pairs are extracted from an in-house platform that uses RocksDB as intermediate storage for application state.



(a) Performance (b) WAF

Figure 2: Performance and write amplification in BlobDB.

- **Enron Mails:** This dataset exemplifies message-based applications (akin to Facebook Messenger), where each email is treated as a distinct KV pair.

Table 1 summarizes key metrics for these datasets, including the total number of KV pairs, value size distribution, and overall dataset size. Notably, the value sizes in the Twitter and Enron Mails datasets – ranging from tens of bytes to several kilobytes – are consistent with the observations in [5].

To evaluate storage efficiency, we conducted a deduplication analysis using an in-house tool with fixed-sized chunking and SHA1-based hashing [17]. Our findings are summarized as follows:

- **Chunk-Size Alignment:** When the deduplication chunk size closely matches the typical value size, significant duplication is detected.
- **File System-level Limitations:** Increasing the chunk size to the conventional file system level (4 KB) leads to a marked drop in duplicate detection. This suggests that file system-level deduplication – commonly used in LSM-KVS alongside compression algorithms – may not be effective for KV datasets, where mismatches between value sizes and deduplication chunk sizes create performance bottlenecks.

Section 3 further discusses these inefficiencies and their impact on data reduction effectiveness.

3 Preliminary Study and Motivation

In this section, we conduct an experimental study to evaluate the effectiveness of KVS-level compression and file system-level deduplication for reducing write amplification in LSM-KVS (see Table 1). Our results demonstrate that file system-level deduplication not only fails to reduce write amplification but also degrades performance, while KVS-level compression provides only minimal write traffic reduction.

3.1 KV Store-level Compression

Compression is widely adopted data reduction technique in KVSs. BlobDB, for instance, supports multiple compression algorithms (e.g., Snappy) at the block level for SSTables and blob files. We evaluated Snappy's impact on write amplification factor (WAF) and throughput using the Streaming and Twitter datasets. Figure 2(a) compares baseline BlobDB with compressed BlobDB$_{compr.}$, with ZFS deduplication disabled to isolate compression effects. Our results show that compression has minimal impact on throughput and only reduces WAF by less than 5%, despite both datasets containing 40–50% duplicate data (Figure 2(b)). This limited effectiveness stems from additional WAL writes, which remain uncompressed, contributing significantly to write amplification.

3.2 File System-level Deduplication

In addition to compression, KVSs often rely on file system-level deduplication. We evaluated ZFS's inline deduplication by configuring its deduplication chunk size to 4 KB (instead of the default 128 KB) to better align with the block size used in BlobDB's SSTable and blob files. Figure 2(a) shows that enabling deduplication (BlobDB$_{dedup}$) causes a significant performance drop compared to the baseline. This degradation is primarily due to the overhead of inline deduplication, which processes every incoming file – including WAL files.

Furthermore, Figure 2(b) indicates a marked increase in WAF with deduplication enabled. This adverse effect results from two main factors:

- ZFS's Copy-on-Write design, which inherently increases WAF.
- The frequent updates to ZFS's deduplication metadata table, leading to additional writes.

Additionally, ZFS fails to detect duplicates due to granularity mismatch – treating 4 KB blocks as atomic units, even when only parts contain unique data. This results in inefficient duplicate identification, especially when multiple KV pairs are packed into a single block. Instead of reducing write traffic, deduplication inflates WAF, as shown in Figure 2(b). *These findings highlight the need for a fine-grained deduplication approach tailored to KVS workloads, addressing both write amplification and metadata overhead more effectively.*

3.3 Exploring Deduplication in LSM-KVSs

Based on our experiments, we make the following key observations:

Observation 1. Both KVS-level compression and file system-level deduplication fail to effectively reduce write traffic, even though the dataset exhibits significant redundancy.

Observation 2. File system-level deduplication introduces substantial performance degradation, undermining the KVS's throughput requirements.

Observation 3. A granularity mismatch between the typical value sizes and the deduplication chunk size leads to suboptimal storage efficiency, thereby limiting the effectiveness of write traffic minimization.

Table 1: Analysis of KV dataset from three applications.

Application	Total # of KV pairs	Distribution of value size (Bytes)			Dataset Size	Deduplication Percentage by Chunk Size			
		Min.	Max.	Avg.		64 B	128 B	256 B	4 KB
Twitter [16]	393141781	3	2271	91	36 GB	39.62	34.54	32.27	0.19
Streaming	76373940	52	1212	132	9.8 GB	49.35	0.31	0	0
Enron Mails [11]	517401	639	5128	2746	1.6 GB	27.53	21.92	19.56	1.93

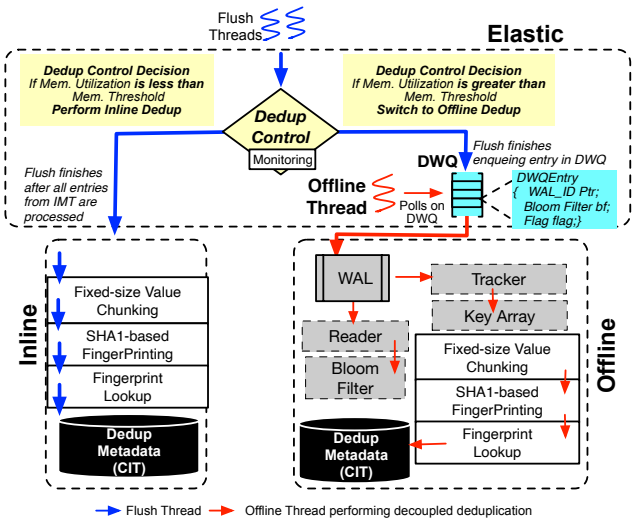


Figure 3: The design overview of DedupKV.

These observations highlight the need to integrate deduplication directly within LSM-KVS, enabling efficient duplicate elimination while overcoming the limitations of compression and file system-level deduplication. However, integrating deduplication into LSM-KVS presents three key challenges:

- **Performance Impact:** Performing inline deduplication during foreground operations (e.g., PUT operations and WAL writes) is infeasible due to the latency sensitivity of LSM-KVS, as shown in Figure 2(a). Therefore, deduplication must be designed to operate in the background to avoid degrading system performance.
- **Incomplete Write Traffic Reduction:** Without inline deduplication, write amplification may not be fully minimized. This necessitates an approach that selectively applies deduplication while maintaining a balance between write traffic reduction and system throughput.
- **Deduplication Metadata Overhead:** Fine-grained deduplication introduces substantial metadata overhead, leading to high memory consumption and frequent random disk lookups. Since each unique value requires a metadata entry, maintaining an efficient index structure is critical to avoid excessive memory usage and performance bottlenecks. Additionally, ensuring fast lookups and updates without compromising deduplication effectiveness remains a key challenge.

4 Design of DedupKV

4.1 Design Goals

The primary design objectives for DedupKV are as follows:

- **Minimal Impact on Foreground I/Os:** DedupKV ensures that deduplication operates efficiently within LSM-KVS, imposing minimal overhead on foreground processes while maintaining fast application writes.
- **Reducing Write Traffic to Storage:** By storing only unique values, DedupKV significantly minimizes write amplification, improving storage efficiency and prolonging device lifespan.
- **Optimizing Metadata Overhead:** To mitigate the high memory and disk lookup costs of fine-grained deduplication, DedupKV employs tiered metadata management, maintaining frequently accessed metadata in memory while offloading cold metadata to storage.

4.2 Design Overview

DedupKV integrates inline and offline deduplication within a KV-separated LSM, optimizing storage efficiency while maintaining performance, as illustrated in Figure 3. The design consists of three synergistic components, each adaptable to different workload demands: FLUSH-Integrated Inline Deduplication: By identifying and eliminating redundant values during the FLUSH operation, DedupKV reduces write amplification before data reaches persistent storage, leveraging background processes for efficiency. WAL File-based Offline Deduplication: To minimize the overhead of inline deduplication, DedupKV uses WAL files as a deduplication source, ensuring that only unique values are written to storage without affecting foreground I/O. Elastic Execution: DedupKV dynamically adjusts resource allocation between inline and offline deduplication, maintaining a balance between write amplification reduction and system throughput under varying workloads. Dynamic Granularity Deduplication (DGD): To balance deduplication efficiency and metadata overhead, DedupKV dynamically selects between exact deduplication for larger values and LZ4 compression for smaller values, reducing storage footprint while optimizing lookup performance. This hybrid approach ensures efficient deduplication while preserving high write performance and low metadata overhead, making DedupKV adaptable to diverse KV store workloads.

4.3 FLUSH-integrated Inline Deduplication

In LSM-KVS, the FLUSH operation moves KV pairs from memory to storage using background threads. Inline deduplication integrates deduplication into this process, eliminating redundant values before they reach storage, thereby reducing write amplification while minimizing impact on foreground PUT operations (Figure 4). Only unique values are stored in the Unique Value Log (UVL) files, which optimizes storage efficiency.

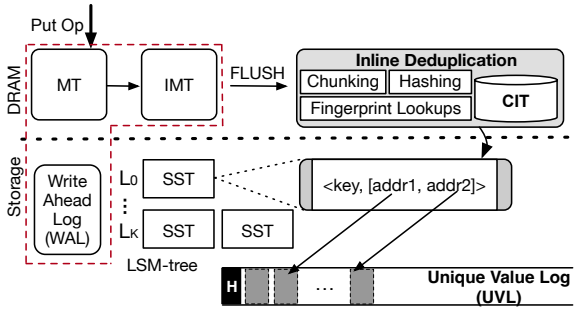


Figure 4: I/O service flow in DedupKV (Inline).

To detect duplicates, inline deduplication utilizes Dynamic Granularity Deduplication (DGD), which dynamically selects the deduplication strategy based on value size: Larger values undergo SHA1-based exact deduplication to remove duplicates. Smaller values are compressed using LZ4, minimizing storage footprint while controlling metadata overhead. By decoupling metadata management from data storage, inline deduplication preserves the SSTable structure, where SSTables store only keys and UVL pointers.

While inline deduplication runs in the background, in-flight operations, such as strategy selection, hashing, and compression, may slightly extend FLUSH durations. The next section explores conditions where inline deduplication maintains optimal performance while minimizing storage overhead.

Limitations of FLUSH-Integrated Inline Deduplication: While FLUSH-integrated inline deduplication reduces write traffic and optimizes space utilization based on data redundancy, its integration within the FLUSH operation can slow background writes. If FLUSH rates drop, LSM's MemTable and IMTs may fill up faster than they can be flushed, triggering write stalls that block incoming writes [4, 49].

Figure 5 compares BlobDB's standard FLUSH behavior (Figure 5(a)) with inline deduplication (Figure 5(b)). In BlobDB, each IMT is flushed to SSTables and vLog without additional processing, thereby freeing memory immediately. In contrast, inline deduplication introduces extra processing time, keeping IMTs in memory longer, and increasing the risk of hitting the write-stall threshold (WST). Once WST is reached, incoming writes are blocked, reducing application throughput.

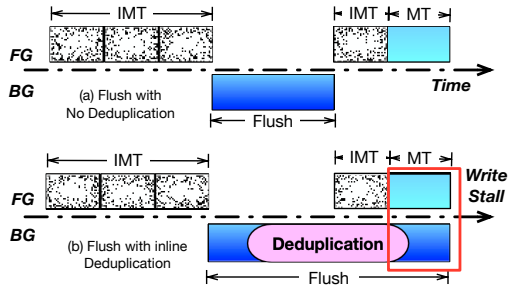
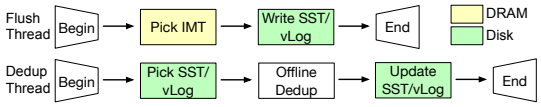


Figure 5: A case of write stall occurrence as a result of performing inline deduplication in Flush operation.

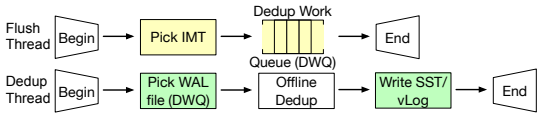
The impact of write stalls depends on workload intensity and redundancy. In high-write workloads with low redundancy, memory can saturate quickly, leading to frequent stalls. In contrast, in workloads with high redundancy, deduplication overhead is minimal compared to storage writes, resulting in fewer stalls. The following sections explore how DedupKV mitigates these limitations through offline deduplication and elastic execution, balancing storage efficiency with system performance.

4.4 WAL file-based Offline Deduplication

When deduplication is integrated into the FLUSH operation (Section 4.3), it can degrade performance under high write I/O traffic. Offline deduplication addresses this by detecting and removing duplicates post-write. However, conventional approaches suffer from a double-write problem: data is first written in its original form, then deduplication invalidates redundant data, and finally, garbage collection rewrites valid data into new storage components—effectively causing each unique value to be written twice (Figure 6(a)) [26, 50].



(a) Traditional offline deduplication approach.



(b) WAL file-based offline deduplication approach.

Figure 6: An illustration of traditional vs offline deduplication approach in LSM-tree.

Mitigating the Double-Write Problem with WAL Files: DedupKV repurposes WAL files for offline deduplication, eliminating the double-write overhead. Since WAL files already log incoming writes for failure recovery, leveraging them for deduplication ensures that only unique KV pairs are written to persistent storage, avoiding redundant rewrites (Figure 6(b)). This approach also offloads deduplication to a background thread, reducing memory pressure and minimizing interference with foreground I/Os.

Each WAL file contains metadata such as CRC checksums, KV sizes, operation types (PUT/DELETE), and payloads. Because the WAL is append-only, duplicate KV pairs may exist within a single file. To efficiently process these records, offline deduplication employs four modules:

- *Reader:* Extracts KV payloads from WAL records.
- *Tracker:* Identifies whether a KV pair has already been processed.
- *Key Array:* Maintains references to processed entries within the same WAL file.
- *Deduplication:* Runs the same deduplication logic as inline deduplication to remove duplicates.

Offline deduplication traverses WAL files from tail to head, prioritizing the most recent records and avoiding unnecessary reprocessing. Instead of immediately writing data to storage, the FLUSH operation enqueues deduplication requests in the Deduplication Work Queue (DWQ). A dedicated background thread then processes the WAL file segments through the offline deduplication modules.

This approach preserves the benefits of offline deduplication – minimal impact on foreground writes – while ensuring that each unique value is written only once, effectively eliminating the double-write problem of traditional methods. Once a WAL file is processed, it is deleted, eliminating the need for garbage collection.

Limitations of WAL File-based Offline Deduplication: Offline deduplication eliminates double-writes and matches inline deduplication in reducing storage traffic. However, its reliance on WAL files impacts read performance. *Get* and *Scan* operations require additional logic to check if data remain in the WAL and handle re-directions. While point lookups are optimized via an index, range queries (*Scan*) remain unoptimized, requiring future improvements. Additionally, the append-only WAL structure necessitates linear traversal for extracting valid KV pairs, adding overhead to read paths. Despite these challenges, offline deduplication minimizes write amplification without extending FLUSH duration or blocking foreground writes, making it a key complement to inline deduplication in DedupKV.

4.5 Elastic Execution

Although WAL file-based offline deduplication reduces write stalls, it can increase read latency by redirecting *Get* operations to WAL files (Figure 8(b)) and lead to WAL file proliferation. To address this, DedupKV dynamically balances inline and offline deduplication. It defaults to inline mode, but under memory pressure, both modes run in parallel—inline deduplication processes data from the IMT, while offline deduplication handles WAL files—ensuring efficient deduplication with minimal impact on reads and writes.

Threshold-Based Deduplication Control. The core challenge is deciding which data should undergo inline deduplication and which should be offloaded to offline deduplication. To address this, DedupKV uses a configurable threshold-based mechanism called *Dedup. Control*. At the start of the execution, a static threshold is chosen to reflect a percentage of the total memory budget for the LSM-tree's memory components. During each FLUSH, the utilization is checked. If memory usage is *below* the threshold, the FLUSH operation proceeds with inline deduplication. Otherwise, the FLUSH operation simply enqueues a job for offline deduplication in the Deduplication Work Queue (DWQ) and immediately frees the IMT. This alleviates memory pressure and avoids blocking incoming writes.

Although concurrent flushes may each detect high memory usage and enqueue multiple WAL file entries to the DWQ, this design deliberately prioritizes preventing foreground write stalls, accepting the potential trade-off of having many WAL files queued for offline processing.

While the Dedup. Control offers a simple and effective way to manage memory pressure by switching between inline and offline deduplication, it assumes memory usage as the primary signal for decision-making. However, in practice, the performance implications of deduplication may vary depending on workload characteristics. For instance, workloads with high write intensity may benefit from prioritizing to reduce write amplification, whereas read-heavy workloads might favor faster flushes to preserve read latency. This observation opens the possibility of enhancing the control policy by incorporating workload-awareness – adjusting deduplication strategies dynamically based on runtime access patterns. Such an adaptive mechanism could further improve both performance and space efficiency under various operating conditions.

Deduplication Work Queue (DWQ). The DWQ is a lightweight, in-memory FIFO queue where each entry describes an offline deduplication task, consisting of:

- **WAL File Number:** Identifies the WAL file that offline deduplication should process.
- **Bloom Filter:** Created during the FLUSH by iterating over all keys in the IMT. This helps minimize read overhead by quickly ruling out KV pairs not present in a particular WAL file. We adopt the same Bloom Filter implementation as BlobDB SSTables, ensuring similar false-positive rates.
- **State Flag:** Tracks the stage of the offline deduplication job, with possible states *Inactive*, *Active*, or *Complete*.

Figure 7 illustrates the life cycle of a DWQ entry as it transitions through these states. The offline deduplication processes *Inactive* entries in the background, marking them *Active* once deduplication begins, and finally setting them *Complete* when the KV pairs have been written to storage.

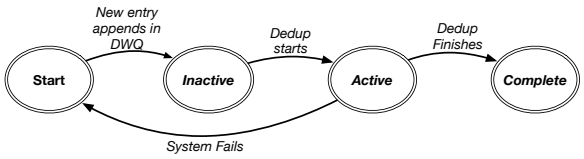


Figure 7: State diagram for the entries in DWQ.

At that point, the DWQ entry and the corresponding WAL file are removed. Note that, there is no garbage collection required for WAL files.

Lightweight monitor. DedupKV relies on a simple monitoring mechanism to track allocation and deallocation of LSM-tree memory components. Because each memory operation updates a shared counter atomically, the overhead is negligible. By reacting to memory pressure before it triggers a write stall, this approach preserves high throughput for foreground writes.

Concurrent Execution and Trade-offs. Enabling both inline and offline deduplication simultaneously ensures efficient write performance under variable workloads. Inline deduplication removes duplicates in real time as long as memory usage remains within acceptable limits, while offline deduplication asynchronously handles overflow jobs from the DWQ. Although this can temporarily increase the volume of unprocessed data in WAL files, it avoids prolonged FLUSH operations that would otherwise stall foreground writes. Details on how DedupKV handles read requests and potential redirection overhead – particularly when multiple WAL files are queued – are discussed in Section 4.7.

Overall, DedupKV provide an adaptive way to balance inline and offline deduplication strategies, preventing write stalls while minimizing the performance penalties associated with managing a growing number of WAL files.

4.6 Dynamic Granularity Deduplication

Drawing inspiration from Xu et. al [45]'s design for online deduplication for databases, DedupKV employs a tiered, adaptive metadata management strategy to overcome the high overhead typical associated with fine-grained deduplication. Our approach – tiered Dynamic Granularity Deduplication (DGD) – integrates multiple deduplication techniques based on the size of the incoming value, thereby balancing space efficiency with effective duplicate detection.

For values exceeding a configurable threshold, DedupKV uses a hash-based exact deduplication method that leverages an in-memory "hot" metadata index. This index, maintained as a compact unordered map, is shown as a Chunk Index Table (CIT) in Figure 3 keyed by a 20 byte SHA1-based [17] fingerprint, and tracks reference counts and other lightweight metadata, ensuring that duplicate lookup operations remain fast. To prevent the in-memory index from growing unbounded, an LRU (least-recently used) eviction policy is applied; once the number of hot metadata entries exceeds a predefined limit, less frequently accessed entries are offloaded to the persistent store.

Conversely, for smaller values – where fine-grained deduplication can lead to prohibitive metadata overhead – DedupKV applies lightweight LZ4 compression [12] before indexing. This size-aware hybrid strategy not only reduces the volume of metadata that must be stored and managed in memory, but also significantly reduces write amplification and overall storage footprint.

Complementing the adaptive deduplication mechanism is our tiered metadata storage system. Inspired by the dual-tier caching techniques in Xu et al.'s work [45], our system partitions metadata into "hot" and "cold" tiers. The hot metadata index resides in memory for immediate access during deduplication operations, while cold metadata is offloaded to LSM-tree of key-value separation design. This division allows DedupKV to maintain low in-memory overhead without sacrificing the ability to quickly reference historical deduplication data, thereby ensuring high performance even under write-intensive workloads.

4.7 I/O Service Flow

In this section, we present the I/O flow in DedupKV, including Put, FLUSH, Get, Scan, and Compaction.

Put: The Put operation in both FLUSH-based inline and WAL-based offline adheres to the path of traditional LSM-tree, where a KV pair is inserted into the MT and then committed to the WAL file. Once the MT becomes full, it is converted to an IMT, which is later processed by background FLUSH operation.

FLUSH: The FLUSH operation in DedupKV follows two paths corresponding to both inline and offline deduplication. In inline deduplication, the FLUSH operation performs inline deduplication, as discussed earlier in Section 4.3. On the other hand, in offline deduplication, the FLUSH operation generates an entry for the DWQ, as stated in Section 4.4. Additionally, with elastic execution, the FLUSH operation functions according to the directives of Dedup Control, dynamically adjusting based on system resources.

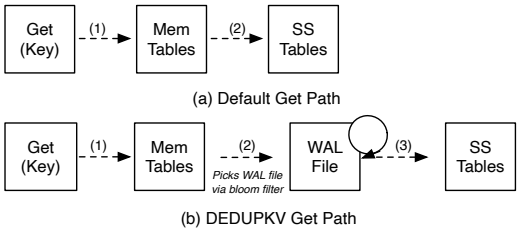


Figure 8: The Get operation in BlobDB and DedupKV. The value corresponding to the key is retrieved from vLog in both cases.

Get: Similar to FLUSH operation, the Get operation in DedupKV follows two different paths for FLUSH-based inline

and WAL-based offline deduplication. In inline deduplication, the Get operation adheres to the traditional flow LSM-tree, as shown in Figure 8(a). However, in offline deduplication, due to the re-purposing of WAL files and the prompt release of the memory components, the Get operation requires modifications, as depicted in Figure 8(b). The Get operation begins by searching for the key in memory components. If the key is found, the corresponding KV pair is returned. If not, the Get operation checks the Bloom Filter in the DWQ before searching the SSTables. If any Bloom Filter returns true, the Get operation sequentially traverses the corresponding WAL file from the tail and returns the first encountered KV pair. If none of the Bloom Filter return true, the search proceeds to the SSTables, following the standard Get operation. However, Bloom Filters are susceptible to false-positive and every false-positive contributes towards read latency. To ensure the ratio of false-positive to be equivalent to baseline BlobDB's SSTables, DedupKV adopts the same Bloom Filter structure in DWQ.

Scan: Although, the baseline BlobDB is not optimized for range scan due to its KV separation approach. DedupKV also follows the similar trade-off with range scan. However, if there are no pending WAL files in DedupKV, then the Scan operation adheres to the path of baseline BlobDB.

Compaction and Garbage Collection: Compaction is performed on the SSTables, which involves sorting and merging key-value pairs. In our approach, the SSTable design remains unchanged, and, therefore, so does the compaction operation. However, BlobDB integrates garbage collection within compaction. Therefore, DedupKV requires accessing the deduplication metadata in cases where the values are updated or removed. For example, when a value is deleted, the compaction operation accesses that value, gets the SHA1 fingerprint, and queries the deduplication metadata table (CIT) to get the latest reference count of the value. If the reference count becomes zero, then the compaction operation marks the value invalid. When the ratio of invalid values within a single UVL file exceeds a certain threshold, the valid values are written to a new UVL file, and their corresponding UVL file number and offsets are updated in the CIT. Note that the garbage collection is only performed on UVL. There is no need for garbage collection for WAL files as they are simply deleted once processed by the offline deduplication.

4.8 Consistency and Failure Analysis

Failure Domains: The key components of LSM-tree, including MT and IMT, are stored in memory. Additionally, DedupKV keeps its deduplication metadata table, such as CIT, DWQ, and the key array of the offline deduplication, in memory. Any unexpected shutdowns or failures may result in data loss. However, LSM-tree relies on WAL files for the fault tolerance of the MT and IMT. In the event of a failure, the WAL file is replayed to recover the data. Similarly, the DWQ and key array can be reconstructed from the WAL files after a failure.

Failure Consistency: In DedupKV, the CIT is considered a critical point for failure consistency, as its reconstruction can significantly extend recovery time and prevent LSM-tree from processing I/O operations. The most basic methods for CIT reconstruction involve traversing each value in the UVL file and rebuilding the CIT. However, this process requires scanning the LSM-tree to identify how many keys reference each value, making it a complex and time-intensive operation.

To reduce recovery time, we implement periodic checkpointing of the CIT onto persistent storage. Upon recovery, the most recent checkpoint is loaded. However, updates made after the last checkpoint and before the failure may be lost, requiring partial CIT reconstruction. We address this with a three-step reconciliation approach: (i) read the tail chunks from the UVL, (ii) access the fingerprints stored along the value, and (iii) cross-check the fingerprint with the CIT from the latest checkpoint. If the fingerprint matches, the CIT is current and the recovery process is complete. If a fingerprint is missing, indicating potential data loss, the missing fingerprint is added to the CIT, and the LSM-tree is traversed to update the reference count. This reconciliation continues until all missing fingerprints are restored, ensuring complete recovery.

5 Evaluation

5.1 Experiment Setup

Implementation: We implemented DedupKV on top of Facebook's BlobDB v8.6.0, retaining key components such as the memory and WAL files. The background thread pool is divided into `FLUSH` (responsible for inline deduplication as well), offline (responsible for WAL-based offline deduplication), and compaction threads. The `FLUSH` thread adjusts its operation based on the incoming request rate, monitored through BlobDB's telemetry data, which tracks the memory utilization of current memory components (MT/IMT). When utilization exceeds DedupKV's defined threshold, the `FLUSH` thread enqueues tasks into the DWQ; otherwise, it performs inline deduplication on each KV pair from the selected IMT. The offline thread processes WAL-based deduplication by polling tasks from the DWQ. To reduce CPU cycles, we considered using semaphores, allowing the offline thread to sleep when the DWQ is empty. For deduplication metadata, we used in-memory hash table, avoiding random disk I/O lookups for duplicate detection when deduplication metadata is stored in memory [41, 46]. Solutions from [28, 44, 46] can complement our approach and be integrated directly.

We conducted comprehensive experiments on a machine that runs Ubuntu 20.04 with Linux Kernel 5.10. The testbed is

equipped with 4 Intel Xeon(R) E5-4640 v2 CPUs @ 2.20 GHz with 10 physical cores per CPU node, 80 MiB last level cache, and 256 GiB DDR3 DRAM. The machine is equipped with 1 TB Samsung 970 EVO SSD.

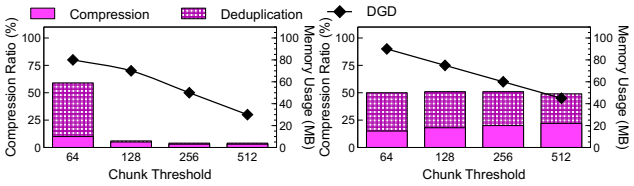
As this is the first comprehensive study of deduplication in LSM-tree-based KV stores, there are no directly comparable state-of-the-art solutions. Therefore, we evaluate the following systems:

- **BlobDB**: The baseline LSM-tree with KV-separation design implemented by Facebook on RocksDB.
- **DKV (In)**: It performs inline deduplication via the FLUSH thread, as discussed in Section 4.3.
- **DKV (Off)**: It performs offline deduplication using WAL files, as described in Section 4.4.
- **DedupKV**: This is the complete DedupKV system that incorporates the elastic execution framework to allow dynamic switching and concurrent execution of inline and offline deduplication based on incoming request arrival rates, as described in Section 4.5.

We conducted a series of experiments to evaluate the performance and overhead of DedupKV under various access patterns and scenarios. Each experiment was repeated at least three times to ensure consistency.

- **Evaluation with Duplicate Dataset** (Section 5.2): Using the real dataset from Table 1, we evaluated DedupKV's performance on KV datasets with duplicate values. We also performed experiments using synthetics datasets with varying duplicate ratios.
- **Evaluation with Real-world Traces** (Section 5.3): We evaluated DedupKV with real-word KV store traces from Nutanix Inc. [3, 4]. We acknowledge that the original data on which these traces were captured are proprietary and are not publicly available. Consequently, we employed the best available public datasets that closely approximate the characteristics of the original data, Streaming, and Twitter datasets.
- **Evaluation with YCSB Benchmark** (Section 5.4): We used the YCSB benchmark [15] to measure performance across different access patterns common in KV stores.

For experiments using real KV dataset and real-world KV traces, we implemented an in-house multi-threaded application. The setup involved 8 application threads to perform operations on the LSM-tree KV stores, alongside 8 background threads dedicated for internal task, including deduplication. We allocated a 1 GB block cache to store frequently accessed SSTables and Blob/UVL files. Each memory component (MT/IMT) was allocated 128 MB, with a maximum of 4 components allowed before triggering a memory pressured-induced write stall. Additionally, we analyzed the memory utilization threshold and for most of the experiments set it to 80% of total memory allocated to LSM-tree's memory components.



(a) Streaming Dataset (b) Twitter Dataset

Figure 9: Compression ratio and index memory usage with varying chunk threshold. The Index Overhead represents the deduplication metadata index overhead.

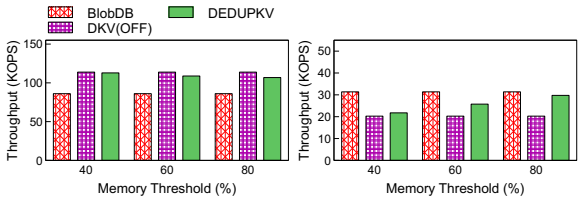
5.2 Evaluation with Duplicates

In this section, we evaluate the performance and write amplification (WAF) of the compared system using real KV datasets (Table 1). Additionally, we provide a microscopic analysis of the dynamic switching between inline and offline, and assess deduplication efficiency, which we define as the proportion of work completed by each deduplication module.

Performance Analysis with real KV datasets: In this section, we first set the basis of our two major thresholds: *Chunk Threshold* – to decide whether to perform compression or exact deduplication on a value – and *Memory Utilization Threshold* – to dynamically switch between inline and offline deduplication. With these experiments, we analyze the compression ratio and memory overhead of the deduplication index (CIT) and pick the optimal configuration parameters for the rest of the experiments. Later, we analyze the performance and WAF of real KV datasets.

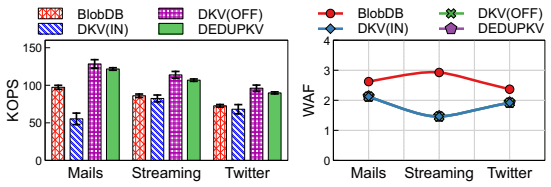
Compression Ratio and Index Memory: Figure 9 evaluates the impact of varying chunk thresholds on the compression ratio and the deduplication index memory overhead. The chunk threshold serves as a decision boundary for applying LZ4 compression versus exact deduplication, directly influencing both storage efficiency and metadata footprint. Figure 9(a) presents results from the Stream KV dataset (refer to Table 1). A clear trend emerges: as the chunk threshold increases, deduplication effectiveness declines sharply. This degradation is attributed to a granularity mismatch, where larger thresholds restrict the scope of exact deduplication, reducing its ability to eliminate redundant data. Furthermore, higher thresholds limit the effectiveness of compression, leading to suboptimal storage efficiency, as discussed in Section 3. A similar pattern is observed in Figure 9(b) for the Twitter KV dataset.

Conversely, a higher chunk threshold significantly reduces deduplication metadata overhead. Since fewer values undergo exact deduplication, the amount of deduplication metadata generated is minimized, resulting in a lower in-memory footprint. In the extreme case, when all values undergo exact



(a) Write-only Workload (b) Mixed Workload

Figure 10: Memory utilization threshold analysis of DedupKV in comparison to baseline BlobDB and DKV (Off). (a) is the write-only workload while (b) is a mixed workload with 50% writes and 50% reads with Streaming dataset.



(a) Performance (b) WAF

Figure 11: Performance and WAF analysis of real datasets. The error bars in (a) present the standard deviation.

deduplication, the deduplication metadata index consumes 2.84 GB of memory for Streaming KV dataset, emphasizing the overhead associated with fine-grained deduplication. To mitigate this issue, dynamic granularity deduplication (DGD) efficiently manages the deduplication metadata through tiered metadata storage. Specifically, only hot metadata – comprising SHA1 fingerprints and reference counts – is retained in memory, while cold metadata is offloaded to the LSM-tree of BlobDB. This hierarchical approach ensures that memory consumption remains controlled while maintaining deduplication effectiveness.

Memory-Utilization Threshold Analysis: Figure 10 illustrates how varying the memory-utilization threshold in DedupKV influences system throughput under different workload patterns. In the write-only scenario (Figure 10(a)), imposing a strict threshold yields performance comparable to DKV (Off) and outperforms the baseline BlobDB by 1.3×. However, relaxing the threshold slightly reduces DedupKV's performance, leading to a 6% drop relative to DKV (Off). In contrast, the mixed workload (Figure 10(b)), comprising 50% writes and 50% reads, shows that with a strict threshold, both DKV (Off) and DedupKV under-perform against BlobDB due to redirection overhead in servicing Get operations via WAL files. As the threshold is relaxed to 80% memory utilization, DedupKV regains performance and closely matches BlobDB. Overall, these observations underscore the effectiveness of threshold-based dynamic switching in DedupKV, which balances write amplification gains with read latency considerations.

Performance and WAF Analysis: Figure 11(a) shows the performance of compared systems on real datasets under a write-only workload, using 8 threads for bursty Put operations. Overall, DKV (Off) and DedupKV maintain a consistently high throughput, outperforming other systems. However, DKV (In) suffers a significant throughput drop, especially with the Mails dataset, where performance decreases by 2× due to fine-grained chunking and SHA1 hash computations that prolong FLUSH operations, causing write stalls (see Figure 5). In contrast, DKV (In) shows improvement with Streaming and Twitter datasets, benefiting from aligned chunk and value sizes, and higher redundancy, which reduces I/O operations and improves performance.

Surprisingly, DKV (Off) and DedupKV achieve throughput rates 33% and 23% higher than BlobDB, respectively. DKV (Off)'s superior performance over BlobDB and DedupKV is due to its WAL-based offline deduplication, which promptly releases the memory and reduces interference between the foreground and background I/Os, a problem seen in DKV (In). DedupKV lags behind DKV (Off) because it dynamically switches between inline and offline, with inline being its primary mode. Additionally, DedupKV's FLUSH operation handles the removal of LSM-KV's memory components and DWQ entry generation. This characteristic can delay the memory release when deduplication operation is being carried out on a selected IMT and thus, limits its overall performance compared to DKV (Off).

WAF Analysis with real KV datasets: Figure 11(b) presents the WAF analysis of the compared systems under a write-only workload. WAD is defined as the total storage write volume divided by the dataset size, including writes to WAL files during Put operations. The results show that deduplication-enabled LSM-KV stores reduce WAF depending on dataset redundancy. Notably, all variants of DedupKV, including DKV (In) and DKV (Off), significantly lower WAF by writing only unique values to the UVL files, effectively eliminating duplicates.

Dynamic switching and deduplication efficiency: This experiment evaluates the dynamic switching and deduplication efficiency of DKV (In) and DKV (Off) when employed under elastic execution of DedupKV over a 20 GB subset of Twitter dataset. The elastic execution mechanism intelligently alternates between DKV (In) and DKV (Off) based on a memory utilization threshold, directly affecting overall performance. The results, presented in Figure 12, show that DedupKV primarily operates in DKV (In) mode, where memory usage gradually increases over time. Upon reaching a predefined threshold, the elastic execution shifts to DKV (Off), indicated by the circular marker in the figure. This

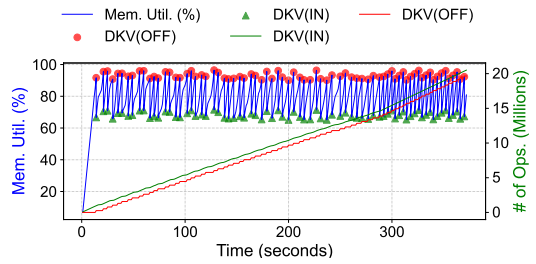


Figure 12: Dynamic switching and deduplication efficiency of inline and offline based on the memory threshold over a subset of Twitter dataset.

transition leads to sharp drops in memory usage, highlighting DKV (Off)'s ability to significantly reduce memory consumption. These transitions occur consistently throughout the runtime, demonstrating the system's ability to efficiently avoid memory pressure-induced write stalls.

Additionally, Figure 12 provides insights into the deduplication efficiency, measured by the number operations performed by each deduplication module. The continuous increase in operations, even during transitions between DKV (In) and DKV (Off), indicates that both components are capable of handling substantial workloads without disruption. The large cumulative number of operations in DKV (In) compared to DKV (Off) suggests that DKV (In) is responsible for the majority of real-time deduplication tasks. In contrast, the stepwise pattern of operations in DKV (Off) indicates periodic bursts of activity, confirming that elastic execution is effectively offloading operations when memory threshold are reached.

5.3 Evaluation with Real-world Traces

We evaluate the performance of the compared systems using real-world traces from Nutanix Inc., as presented in SILK and TRIAD [3, 4]. Nutanix employs KV stores as their primary storage engine, and these traces were collected from their production environment. The traces are write-dominant, with a 57:41:2 write:read:scan ratios (scans covers tens of keys). As stated, the original data on which these traces were captured is not publicly available, therefore, we employed the best available public datasets that closely approximate the characteristics of the KV datasets, Streaming and Twitter datasets. For these experiments, we initially loaded 50% of the dataset before executing the traces. We employed 8 application threads for the operations: 4 threads handled writes, while 2 threads each were dedicated to reads and scans.

Figure 13 presents the throughput and basic I/O latency analysis of the compared systems. In Figure 13(a), we observe the overall trace execution performance. Both DKV (In) and DedupKV perform nearly on par with the baseline BlobDB, showing only a 5% performance degradation. Notably, DKV (Off) consistently demonstrates the worst performance, primarily due to the high overhead of traversing WAL file during read operations, resulting in a significant performance drop to an average of 13%. In contrast, DKV (In) and DedupKV maintain close performance to the baseline because DKV (In) leaves the read and scan operation paths unchanged, while DedupKV's elastic execution strategy limits the number of WAL files, minimizing the negative impact on overall performance.

Latency Analysis: Until now, we have focused on evaluating the throughput of the compared systems using real KV datasets and real-world KV traces. Now, it is important to examine latency, another critical metric, especially given that previous studies [3, 4, 49] have highlighted writes stalls, a primary cause for high latency, as a significant concern for users. Figure 13(b) illustrates the latency distribution for each operation using box-and-whisker plots for the Twitter dataset (we excluded the Streaming dataset due to similar patterns). This analysis focuses on foreground operations; write latency corresponds to the Put operation, while read and scan latencies represent the Get and Scan operations, respectively, performed by application threads.

From Figure 13(b), we observe that DKV (In) exhibits the highest average and tail latencies for the Put operation, primarily due to write stalls induced by memory pressure. Nevertheless, it maintains average latencies for the Get and Scan operations comparable to the baseline, as it preserves the read and scan paths without modifications. Conversely, DKV (Off) achieves the best average and tail latencies for the Put operation, attributed to its prompt memory release and effective offloading of KV pair persistence to WAL-based offline deduplication. However, DKV (Off) incurs the worst Get latencies due to the combination of immediate memory release and the costly search operations required in the append-only WAL files. Additionally, as DKV (Off) does not support the Scan operation, these values were excluded from the evaluation.

Furthermore, Figure 13(b) demonstrates that DedupKV achieves improved average and tail latencies for the Put operation compared to the baseline BlobDB and DKV (In). This improvement is attributed to its elastic execution, which combines the strengths of both DKV (In) and DKV (Off). DedupKV allows the concurrent execution of DKV (In) and DKV (Off), with distinct threads operating on different datasets, improving overall deduplication efficiency (Figure 12). This concurrency enables DedupKV to maintain Get and Scan latencies close to those of BlobDB and DKV (In). However, the tail latencies for Get and Scan operation in DedupKV are slightly higher than those of BlobDB and DKV (In) due to modifications in the read path that account for WAL files processed by DKV (Off). It is also noteworthy that DedupKV

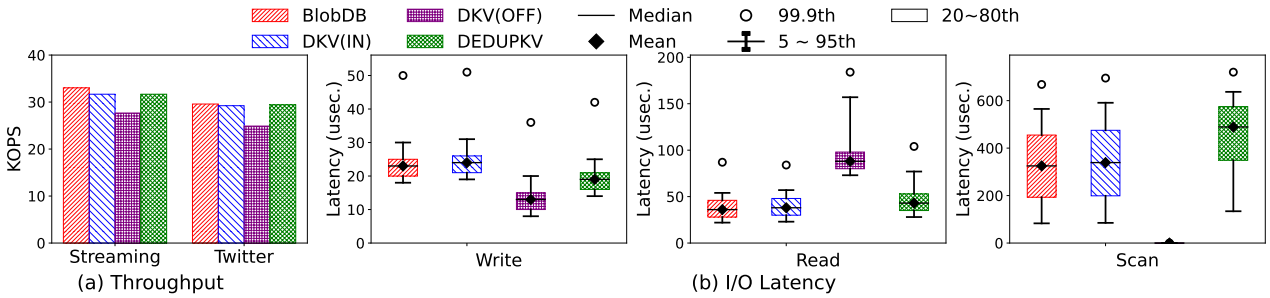


Figure 13: Basic I/O performance on Real-world traces. (b) illustrates the latency distributions using box-whisker plots. For example, boxes indicate the 20th to 80the percentiles, whiskers indicate the 5th to 95th percentiles, and circle fliers indicate 99.9th percentiles.

mitigates this overhead in comparison to DKV (Off) by limiting the number of WAL files through a memory threshold-based dynamic switching.

Table 2 presents the average latency of key DedupKV components for the Streaming and Twitter datasets. LZ4 compression is the fastest operation, with latencies of 0.56 µs (Streaming) and 0.72 µs (Twitter). SHA1 fingerprinting is significantly more expensive, particularly in the Twitter dataset (3.06 µs) compared to 1.06 µs in Streaming, highlighting dataset-dependent hashing overhead. Duplicate lookup incurs 8 µs for Streaming and 16 µs for Twitter, while Bloom filter (BF) lookup remains efficient at 5 µs for both datasets. BF creation introduces a fixed overhead of 175 µs, making it the most expensive operation. LSM-tree and Blob file write I/O show the highest storage latencies, particularly for Twitter (6.63 µs) versus 4.25 µs for Streaming. These results emphasize that LZ4 compression and lookups are lightweight, whereas SHA1 hashing, BF creation, and storage I/O dominate DedupKV's overall latency.

5.4 Evaluation with Standard Benchmark

In this section, we utilized the YCSB benchmark as a standard benchmark to evaluate the compared systems under varying access patterns and workloads. Unlike BlobDB's default db_bench, which generates 99% duplicate values, YCSB inherently produces unique value without offering control over duplicate value generation. This setup not only provides a more diverse range of access patterns but also serves as a stress test for simulating workloads that contain no duplicate values. Table 3 presents details of workload executed through YCSB benchmark.

Figure 14 presents the performance results across all workloads. For these experiments, we initially loaded *100 GB* of data with *512 bytes of KV pairs*, with the performance of this load phase labeled as "Load" in the figure. In the run phase of YCSB, we executed *25 Million* request for each workload, with Zipfian request distribution. It is evident that DKV (In) exhibits the worst performance in Load phase (2×), meanwhile it exhibits poor performance in workloads with significant portion of update operations, such as workload A and F. This performance degradation is attributed to increased interference between foreground and background operations. In addition to inline deduplication in FLUSH operation, the compaction process contributes to this slowdown, particularly in workloads A and F. During compaction, DKV (In) accesses and updates the CIT when a value becomes invalid in UVL, and this additional redirection results in further performance degradation. However, for workloads dominated by read and scan operations, DKV (In) maintains performance levels comparable to the baseline BlobDB.

Additionally, DKV (Off) demonstrates superior performance in Load phase, benefiting from its efficient handling of write operations. However, the overhead of WAL file traversal during read operations becomes evident in workloads

Table 2: Average latency analysis of different components of DedupKV in Microseconds.

KV dataset	Streaming	Twitter
LZ4 Compression	0.56	0.72
SHA1 Fingerprint	1.06	3.06
Duplicate Lookup	8	16
BF Lookup	5	5
BF Creation	175	175
LSM-Tree Write I/O	4.25	6.63
Blob File Write I/O	4.25	6.63

Table 3: YCSB Benchmark and workload description.

Workload	Description
Load	write-only: 100% writes
A	write-intensive: 50% updates, 50% reads
B	read-intensive: 5% updates, 95% reads
C	read-only: 100% reads
D	read-latest: 5% updates, 95% reads
E	scan-intensive: 5% updates, 95% scans; average scan length 50 elements
F	50% read-modify-write, 50% reads

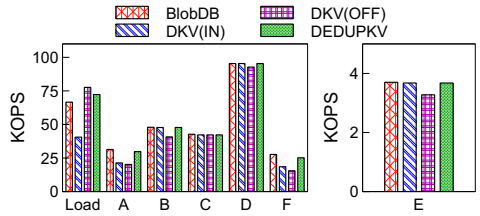


Figure 14: Performance analysis of YCSB Benchmark with different workloads.

such as A, B, D, and F, which involve update and read-modify-write (RMW) access patterns. In these workloads, each Put operation is preceded by either verifying the existence of a KV pair or retrieving the entire KV pair. These steps necessitate querying the Bloom Filters and, in the worst-case scenario, traversing the WAL files, resulting in increased latency and affecting overall performance. Although, DKV (Off) does not support range scan operations, in this experiment, the scan operations were performed on preloaded data, allowing DKV (Off) to perform comparably to the baseline BlobDB and DKV (In).

Moreover, DedupKV consistently demonstrates performance comparable to the BlobDB and outperforms both DKV (In) and DKV (Off) in several workloads, including A, C, D, and F. This advantage is attributed to DedupKV's elastic execution, which effectively limits the number of WAL files, thereby mitigating the negative impact on update and RMW access patterns. Additionally, while the compaction process still incurs some overhead due to the redirection needed for CIT updates, this overhead does not propagate in a way that causes significant interference between foreground and background operations. As a result, DedupKV maintains its performance advantage in diverse workloads.

6 Conclusion

In this study, we explore the integration of deduplication into LSM-tree-based key-value stores to address inefficiencies caused by the mismatch between key-value pair sizes and deduplication block sizes. We proposed three deduplication approaches: Inline, Offline, and Hybrid (DedupKV)—each designed to optimize write amplification and storage utilization. Through extensive evaluation with real-world datasets and benchmarks, we demonstrated that WAL-based offline and DedupKV significantly outperform BlobDB in write-heavy workloads, reducing write amplification by up to 2×. These findings highlight the potential for direct deduplication within key-value stores as an effective strategy to improve performance and storage efficiency.

Acknowledgments

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-00564249 and RS-2024-00416666). This research also used resources of the Oak Ridge Leadership Computing Facility, located at the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

References

[1] Mohammadamin Ajdari, Pyeongsu Park, Dongup Kwon, Joonsung Kim, and Jangwoo Kim. 2017. A scalable HW-based inline deduplication for SSD arrays. *IEEE Computer Architecture Letters* 17, 1 (2017), 47–50.

[2] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2022. DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 281–296.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 363–375.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766.

[5] 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 18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223.

[6] Zhichao Cao, Hao Wen, Xiongzi Ge, Jingwei Ma, Jim Diehl, and David HC Du. 2019. TDDFS: A tier-aware data deduplication-based file system. *ACM Transactions on Storage (TOS)* 15, 1 (2019), 1–26.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv* abs/1512.01274 (2015).

[9] Wande Chen, Zhenke Chen, Dingding Li, Hai Liu, and Yong Tang. 2021. Low-overhead inline deduplication for persistent memory. *Transactions on Emerging Telecommunications Technologies* 32, 8 (2021), e4079.

[10] Xubin Chen, Ning Zheng, Shukun Xu, Yifan Qiao, Yang Liu, Jiangpeng Li, and Tong Zhang. 2021. KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression. In *Proceedings of the 17th International Workshop on Data Management on New Hardware* (Virtual Event, China) *(DAMON '21)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages.

[11] William Cohen. 2004. Enron Email Dataset. https://www.cs.cmu.edu/~enron/. Accessed: 2024-10-15.

[12] Yann Collet. 2011. LZ4 Compression. https://github.com/lz4/lz4. Accessed: 2024-10-16.

[13] Yann Collet. 2011. LZ4HC Compression. https://github.com/lz4/lz4. LZ4 high-compression mode, Accessed: 2024-10-16.

[14] Yann Collet. 2016. Zstandard Compression. https://github.com/facebook/zstd. Accessed: 2024-10-16.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. YCSB: Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB. Accessed: 2024-10-15.

[16] Walter Daelemans, Mike Kestemont, Enrique Manjavacas, Martin Potthast, Francisco Rangel, Paolo Rosso, Günther Specht, Efstathios Stamatatos, Benno Stein, Michael Tschuggnall, Matti Wiegmann, and Eva Zangerle. 2019. Overview of PAN 2019: Bots and Gender Profiling, Celebrity Profiling, Cross-Domain Authorship Attribution and Style Change Detection. In *Experimental IR Meets Multilinguality, Multimodality, and Interaction: 10th International Conference of the CLEF Association, CLEF 2019, Lugano, Switzerland, September 9–12, 2019, Proceedings* (Lugano, Switzerland). Springer-Verlag, Berlin, Heidelberg, 402–416.

[17] D. Eastlake and P. Jones. 2001. RFC3174: US Secure Hash Algorithm 1 (SHA1).

[18] Inc. Facebook. 2013. RocksDB. https://rocksdb.org. Version 6.29, Accessed: 2024-10-15.

[19] Inc. Facebook. 2018. BlobDB. https://rocksdb.org. Version 6.29, Accessed: 2024-10-15.

[20] Inc. Facebook. 2018. Cassandra on RocksDB at Instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram/. Accessed: 2024-10-15.

[21] Inc. Facebook. 2024. Facebook. https://www.facebook.com. Accessed: 2024-10-19.

[22] Apache Software Foundation. 2008. Apache Cassandra: A Highly Scalable Distributed Database. https://cassandra.apache.org/. Accessed: 2024-10-15.

[23] Google. 2011. Snappy Compression. https://github.com/google/snappy. Accessed: 2024-10-16.

[24] Inc. Google. 2011. LevelDB. https://github.com/google/leveldb. Accessed: 2024-10-15.

[25] Prince Hamandawana, Awais Khan, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. 2020. Crocus: Enabling Computing Resource Orchestration for Inline Cluster-Wide Deduplication on Scalable Storage Systems. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1740–1753.

[26] Safdar Jamil, Joseph Ro, Joo-Young Hwang, and Youngjae Kim. 2024. Efficient Data Placement in Deduplication Enabled ZenFS via CRC-Based Prediction. *IEEE Access* (2024).

[27] Charles Jaranilla and Jongmoo Choi. 2023. Requirements and Trade-Offs of Compression Techniques in Key–Value Stores: A Survey. *Electronics* 12, 20 (2023).

[28] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the 2018 {USENIX} Annual Technical Conference (USENIX ATC 18)*. 993–1005.

[29] Awais Khan, Prince Hamandawana, and Youngjae Kim. 2020. A Content Fingerprint-Based Cluster-Wide Inline Deduplication for Shared-Nothing Storage Systems. *IEEE Access* 8 (2020), 209163–209180.

[30] Awais Khan, Chang-Gyu Lee, Prince Hamandawana, Sungyong Park, and Youngjae Kim. 2018. A Robust Fault-Tolerant and Scalable Cluster-Wide Deduplication for Shared-Nothing Storage Systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Milwaukee, Wisconsin, USA, 87–93.

[31] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. 2012. Deduplication in SSDs: Model and quantitative analysis. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, San Diego, CA, USA, 1–12.

[32] Hyungjoon Kwon, Yonghyeon Cho, Awais Khan, Yeohyeon Park, and Youngjae Kim. 2022. DENOVA: Deduplication Extended NOVA File System. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, VIRTUALLY, 1360–1371.

[33] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14.

[34] ByteDance Ltd. 2024. ByteDance. https://www.bytedance.com. Accessed: 2024-10-19.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1, Article 5 (March 2017), 28 pages.

[36] Linsen Ma, Rui Xie, and Tong Zhang. 2023. ZipKV: In-Memory Key-Value Store with Built-In Data Compression. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (Orlando, FL, USA) *(ISMM 2023)*. Association for Computing Machinery, New York, NY, USA, 150–162.

[37] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. 2012. A study on data deduplication in HPC storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. ACM/IEEE, Salt Lake City, Utah, USA, 1–11.

[38] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 2022. DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 247–264.

[39] João Paulo and José Pereira. 2014. A Survey and Classification of Storage Deduplication Systems. *ACM Comput. Surv.* 47, 1, Article 11 (June 2014), 30 pages.

[40] Jiansheng Qiu, Yanqi Pan, Wen Xia, Xiaojia Huang, Wenjun Wu, Xiangyu Zou, Shiyi Li, and Yu Hua. 2023. Light-Dedup: A Light-weight Inline Deduplication Framework for Non-Volatile Memory File Systems. In *In Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 101–116.

[41] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. 2012. iDedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 24.

[42] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, Yechao Yang, and Mingdi Xue. 2017. NV-Dedup: High-performance inline deduplication for non-volatile memory. *IEEE Trans. Comput.* 67, 5 (2017), 658–671.

[43] Hao Wang, Jiaxin Ou, Ming Zhao, Sheng Qiu, Yizheng Jiao, Yi Wang, Qizhong Mao, Zhengyu Yang, Yang Liu, Jianshun Zhang, et al. 2024. LavaStore: ByteDance's Purpose-built, High-performance, Cost-effective Local Storage Engine for Cloud Services. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3799 – 3812.

[44] Guangping Xu, Bo Tang, Hongli Lu, Quan Yu, and Chi Wan Sung. 2019. LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Santa Clara, CA, USA,, 299–310.

[45] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. 2017. Online Deduplication for Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1355–1368.

[46] Qirui Yang, Runyu Jin, and Ming Zhao. 2019. SmartDedup: optimizing deduplication for resource-constrained devices. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 633–646.

[47] Jianwei Yin, Yan Tang, Shuiguang Deng, Ying Li, and Albert Y Zomaya. 2017. D^3: A dynamic dual-phase deduplication framework for distributed primary storage. *IEEE Trans. Comput.* 67, 2 (2017), 193–207.

[48] Hongliang Yu, Xu Zhang, Wei Huang, and Weimin Zheng. 2017. PDFS: Partially Dedupped File System for Primary Workloads. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2017), 863–876.

[49] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 65–80.

[50] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R Butt. 2020. DupHunter: Flexible High-Performance Deduplication for Docker Registries. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 769–783.