LavaStore: ByteDance's Purpose-built, High-performance, Cost-effective Local Storage Engine for Cloud Services

Hao Wang[*], Yi Wang[*], Jianyang Hu[*], Lixun Cao[*], Lei Zhang[*]
Jiaxin Ou[*], Qizhong Mao[*], Jingwei Zhang[*], Heng Zhang[*], Jian Liu[*]
Ming Zhao[*], Zhengyu Yang[*], Jinrui Liu[*], Hongde Li[*], Guanghui Zhang[*]
Sheng Qiu[*], Yang Liu[*], Jiaqiang Chen[*], Ming Li[*], Fei Liu[*]
Yizheng Jiao[*], Jianshun Zhang[*], Yong Shen[*], Yue Ma[*], Jianjun Chen[*]
ByteDance

ABSTRACT

Persistent key-value (KV) stores are widely used by cloud services at ByteDance as local storage engines, and RocksDB used to be the *de facto* implementation since it can be tailored to a variety of workloads and requirements. In this paper, we provide key insights into local storage engine usage at ByteDance, explain why the combination of highly write-intensive workloads and stringent requirements on cost efficiency and point lookup tail latency may pose challenges to a general-purpose local storage engine such as RocksDB, and present the design and implementation of *LavaStore*, a high-performance cost-effective local storage engine purpose-built to address these challenges.

LavaStore achieves its design goals by selectively customizing a few components of a RocksDB-based, general-purpose local storage engine, including a distinct KV separation design that decouples garbage collection from compaction, a specialized engine type for the commonly recurring Write-Ahead-Logging workload, and a customized user-space append-only filesystem. LavaStore has been deployed to production with hundreds of thousands of running instances, storing more than 100 PB of data and serving billions of requests per second, bringing significant performance improvements and cost reductions to customers over their original local storage engines. For example, a ByteDance proprietary distributed OLTP database service has experienced a reduction in average write and read latency by 61% and 16%, respectively, and a ByteDance proprietary caching service has gained an 87% increase in write throughput with no more than 6% space overhead.

PVLDB Reference Format:
Hao Wang, Jiaxin Ou, Ming Zhao, Sheng Qiu, Yizheng Jiao, Yi Wang, Qizhong Mao, Zhengyu Yang, Yang Liu, Jianshun Zhang, Jianyang Hu, Jingwei Zhang, Jinrui Liu, Jiaqiang Chen, Yong Shen, Lixun Cao, Heng Zhang, Hongde Li, Ming Li, Yue Ma, Lei Zhang, Jian Liu, Guanghui Zhang, Fei Liu, and Jianjun Chen. LavaStore: ByteDance's Purpose-built, High-performance, Cost-effective Local Storage Engine for Cloud Services. PVLDB, 17(12): 3799 - 3812, 2024.
doi:10.14778/3685800.3685807

[*]{hao.wang, oujiaxin, zhaoming.274, sheng.qiu, yizhengjiao, wangyi.ywq, qizhong.mao, zhengyu.yang, yangliu1, zhangjianshun, hujianyang, zhangjingwei.831, liujinrui.yummy, chenjiaqiang.0, shenyong.sy, caolixun, zhangheng.he, lihongde, liming.1018, mayue.fight, zhanglei.michael, liujian.kv, zhangguanghui, fei.liu, jianjun.chen }@bytedance.com

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685807

1 INTRODUCTION

Persistent Key-Value (KV) stores are widely used by cloud services at ByteDance as local storage engines. For example, ByteNDB, a proprietary OnLine Transaction Processing (OLTP) system, stores database page versions in persistent KV stores; ABase, a proprietary distributed NoSQL database, also implements Redis-compliant data structures on top of persistent KV stores.

RocksDB [25] is a Log-Structured Merge (LSM) tree [62] based, high-performance persistent KV store developed for large-scale distributed systems and optimized for Solid State Drives (SSDs). Due to its configurability, RocksDB can be tailored to a variety of workloads and requirements, and became the *de facto* local storage engine implementation for many cloud services at ByteDance.

With the tremendous growth of popular ByteDance applications, however, some cloud services began to encounter performance and cost issues with their RocksDB-based local storage engines. After numerous attempts in tuning RocksDB configuration, we came to the conclusion that the unique workload characteristics, performance requirements, and cost objectives of these cloud services demand a local storage engine design with some distinctly different trade-offs from that of RocksDB. In 2019, ByteDance acquired TerarkDB [9], a RocksDB-based general-purpose KV store with customized indexing and compression algorithms, and set out to develop *LavaStore*, a high-performance and cost-effective local storage engine based on TerarkDB but purpose-built for cloud services at ByteDance. In this paper, we describe the challenges for local storage engines at ByteDance, explain why existing designs in RocksDB fall short, and present the design and implementation of LavaStore that address these challenges.

Write throughput was one of the first major bottlenecks that emerged early on due to the unique workload characteristics at ByteDance. Specifically, ByteDance applications aggressively deployed in-memory caches (e.g., Redis and Memcached) at multiple layers of their architecture to reduce read latency, leaving only a small fraction of read requests for local storage engines to handle. Furthermore, with applications commonly batching write requests for higher throughput, the write workload is dominated by large value writes. Unfortunately, RocksDB's write throughput under such a workload is severely limited by the inherently large write amplification of LSM-tree for large value sizes. Even with `BlobDB`, which is the RocksDB implementation of the commonly used technique of *KV separation* to improve write throughput for large value sizes, the write throughput still falls significantly short of application requirements. In order to address this challenge, we opted for a distinct KV separation design that decouples Garbage Collection (GC) from compaction, which enables more flexible trade-offs among space usage, read performance and write performance. Compared with the design of `BlobDB` in RocksDB, which ties GC to compaction, LavaStore can achieve much better write performance with comparable space usage and read performance.

As applications in ByteDance enjoyed continued exponential growth, the resource consumption by the cloud services, in terms of the volume of data stored, CPU usage, etc., gradually grew to a scale for which cost reduction became a major concern. In order to improve resource efficiency, LavaStore introduced various GC optimizations to its KV separation implementation. In particular, for the commonly recurring Write-Ahead-Logging (WAL) workload, we added a specialized local storage engine, LavaLog, to exploit the fact that these data are mostly written and expired in a First-In-First-Out (FIFO) fashion, and rarely read, in order to achieve near-optimal write amplification and GC overhead. Moreover, some applications would like to enable RocksDB "sync" write to guarantee data durability, but the ensuing throughput and latency penalty is so large that most of them opted for "non-sync" write with increased replication, effectively trading resource efficiency for durability. For such applications, LavaStore employed cross-layer optimization between the KV store and the underlying filesystem to eliminate the "sync write" performance penalty, which helps these cloud services restore resource efficiency while maintaining data durability.

Finally, as cloud services continued to pursue aggressive cost reductions, the previously abundant in-memory caches got more and more scarce, so read performance optimization began to gain importance. With a careful study of application requirements and read performance metrics, we found that point lookup queries are the main pain point. Specifically, due to their interactive nature, many popular ByteDance applications have stringent Service Level Agreements (SLAs) on both the average and tail (e.g., the 99-percentile) latencies. Although the average read latencies of most cloud services were well below SLA, tail latencies were orders of magnitude above SLA. Since resources must be provisioned to meet SLAs on both average and tail latencies, average resource utilization became very low. Reducing the tail latency of point lookup queries thus became the key to improving resource efficiency. To this end, LavaStore introduced a new index type that is very memory efficient with outstanding point lookup performance. Together with a more refined caching strategy, this new index type enables LavaStore to achieve near-optimal read amplification for point lookup queries, thus drastically closing the gap between average and tail latencies for point lookup queries, and leading to substantial cost savings for such cloud services.

As of this writing, LavaStore has been successfully deployed to three widely used cloud services at ByteDance, with more than 100,000 running instances in production, storing more than 100 PB of data and serving an aggregate of over 2 billion Queries Per Second (QPS). Customers of LavaStore have enjoyed great benefits over their original local storage engines. Specifically, ByteNDB has seen its average write latency reduced by 61% and its read latency by 16%. The write QPS of ABase is increased by 87% while keeping the total garbage ratio between 1% and 6%. Flink also reduces its CPU usage by up to 67% after switching from its previous RocksDB-based state backend to one based on LavaStore.

In summary, our key contributions are as follows:

- We provide insights into local storage engine usage by cloud services at ByteDance, and explain why the combination of highly write-intensive workloads and stringent requirements on cost efficiency and point lookup tail latency may pose challenges to a general-purpose local storage engine such as RocksDB;
- We show how these challenges can be effectively addressed by selectively customizing a few components of a RocksDB-based, general-purpose local storage engine. Specifically,
  - We present LavaKV, with a distinct KV separation design that decouples garbage collection from compaction, which enables more flexible trade-offs among space usage, read performance, and write performance than RocksDB;
  - We present LavaLog, a specialized local storage engine for the commonly recurring Write-Ahead-Logging workload, which significantly outperforms RocksDB in terms of write amplification and garbage collection overhead;
  - We present LavaFS, a user-space append-only filesystem, which provides much lower write amplification and synchronous write latency than the in-kernel filesystem Ext4 when used by LavaKV and LavaLog;
- Using both synthetic and production workloads, we validate LavaStore's design and implementation by showing that it successfully meets the performance requirements and cost objectives of cloud services at ByteDance;
- We share the lessons we have learned while developing LavaStore and running it in production at scale.

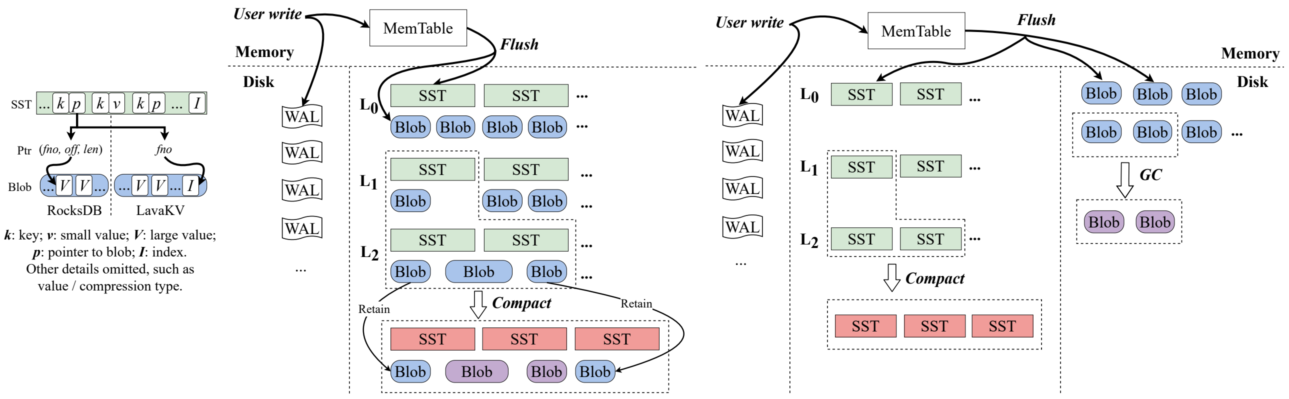
The rest of the paper is organized as follows. Section 2 presents the background and motivation of this work. Section 3 describes the design of LavaStore, with Section 3.2 focusing on improving write performance, Section 3.3 on improving cost-effectiveness, and Section 3.4 on improving read performance. In Section 4, we evaluate LavaStore's performance and cost-effectiveness using both synthetic and production workloads. Section 5 discusses our lessons learned and future work. Section 6 reviews the related work. Finally, Section 7 concludes our work.

2 BACKGROUND AND MOTIVATION

In this section, we first describe three cloud services that represent typical use cases of persistent KV stores as local storage engines at ByteDance, with a focus on their workload characteristics, performance requirements, and cost objectives. We then explain why the existing designs in RocksDB does not adequately address these use cases, which also motivates the design of LavaStore.

2.1 Local Storage Engine Usage at ByteDance

2.1.1 ByteNDB. ByteNDB, short for ByteDance NewSQL Database, is a cloud-native, distributed OLTP database suite engineered for full compatibility with MySQL [26] and PostgreSQL [57]. Moving away from MySQL's traditional non-distributed InnoDB storage engine, ByteNDB's storage layer comprises two key components: LogStore and PageStore [15]. LogStore leverages append-only distributed



(a) SST/blob format **(b) RocksDB (blob GC with SSTable compaction)** **(c) LavaKV (independent SSTable compaction and blob GC)**

Figure 1: Comparison of key-value separation between RocksDB and LavaKV.

blob storage to provide fast redo log persistence with large capacity, while PageStore, through a collection of page servers, handles the persistence of redo logs from LogStore and their application in constructing data pages. Each PageStore, managing 16 KB database page versions and indexing them by unique IDs, places significant demands on local storage engines for large value management.

To ensure high availability, page servers persist redo logs from LogStore using unique Logical Sequence Numbers (LSNs), and use a gossip protocol [6] to retrieve missing logs from peers. This allows for mostly sequential log insertions with occasional out-of-order entries. Immediate durability is critical, ensuring that log appends are persistently stored before user confirmation.

2.1.2 ABase. ABase is ByteDance's distributed KV storage system, supporting services like advertising, e-commerce, recommendation, search, and video platforms. It caches large datasets beyond in-memory capacity, such as Redis [12], using a disk-based approach for handling large KV pairs and terabyte-scale data. ABase ensures high throughput, essential disk-based data preservation, and low cost. Its critical operations are `Set` (inserting/overwriting values) and `Get` (retrieving values).

LavaStore's KV separation significantly boosts write throughput and reduces tail write latency, aligning with ABase's performance expectations. Yet, as ByteDance's online services expand, the challenge of tail read latency emerges, particularly as in-memory caching becomes increasingly expensive. The objective is to reduce disk accesses in Sorted-Sequence/String Tables (SSTables) within LavaStore, ideally limiting it to a single I/O to retrieve a value from the blob file, thereby minimizing tail read latencies.

2.1.3 Apache Flink. Flink, a framework and distributed processing engine for stateful computations on unbounded and bounded data streams [11], is instrumental at ByteDance, powering over 30,000 streaming and 100,000 batch jobs daily across various business units. Flink's demand for high write throughput, focusing on lookup queries with occasional scans, requires efficient resource utilization. Deployed via YARN [77] or Kubernetes [8], Flink configurations often involve multiple storage engine instances per process on shared CPU and memory, sometimes exceeding 100 instances on a single machine. Each engine, running within containers, has a predefined quota for CPU and memory allocation. Given Flink's substantial need for computational resources, optimizing the storage engine to minimize CPU and memory consumption is critical, enabling more resources for computation-heavy tasks.

2.2 Problems with Existing Systems

2.2.1 KV Separation in RocksDB. Adjusting compaction settings, such as increasing flush size or switching to universal compaction, are common methods to mitigate RocksDB's write amplification. However, these approaches often fall short in sufficiently improving tail write latencies. The KV separation technique, introduced by Lu et al. [48], offers a more effective solution by storing large values in separate blob files and reducing their involvement in compactions but increasing space amplification. It decreases compaction frequency and size, thus reducing write amplification and necessitating specialized garbage collection to preserve space efficiency. While it improves write performance through reduced SSTable sizes and potentially better cache efficiency, it may increase disk I/O for accessing separated values, negatively impacting read performance.

RocksDB quickly incorporated the concept of KV separation through `BlobDB` [55]. It stores a triplet of blob file number, value offset, and value length within SSTables (Figure 1a), requiring a space overhead of approximately 24 bytes per key and facilitating efficient read access, as values in a blob file could be directly retrieved using the offset and length without the need for a separate index. However, blob GC requires rewriting all valid values to new files and updating their triplets of blob file number, value offset, and value length in the associated SSTables, changes that can only be made during compaction. This inadvertently increases write and space amplification since retained blob files must wait for future compactions of associated SSTables for reclamation, as shown in Figure 1b. Despite attempts to strike a balance among write, read, and space efficiencies, `BlobDB` did not fulfill our stringent requirements for ultra-low write amplification and minimized tail write latency. In 2019, PingCAP released the initial version of Titan, a RocksDB plugin with a slightly different KV separation design [67, 84]. Titan's strategy is similar to `BlobDB` but uses an

`EventListener` callback to re-insert GC'ed KVs as updates after compactions. Although Titan was more usable and outperformed `BlobDB`, its write performance still fell short of our requirements. For a more in-depth comparison of KV separation between `BlobDB` and Titan, please refer to [81].

In addition to the write performance bottleneck of `BlobDB` and Titan, their instability and unavailability between 2018 and 2019 precluded their direct application in our production settings, compelling us to devise our own KV separation method, which is elaborated upon in Section 3.2. Our initial KV separation approach surpassed both `BlobDB` and Titan in write throughput and reduced tail write latency but incurred greater space use and read amplification. These issues are further discussed in Section 3.3 and Section 3.4, respectively. For the remainder of this paper, unless otherwise specified, RocksDB will refer to RocksDB `BlobDB`.

2.2.2 Ineffective Synchronous Writes. To ensure high reliability, WAL must be used for synchronous writes on every insert. Moreover, specific workloads, such as ByteNDB's log shipping, demand low synchronous write latency with a Write Amplification Factor (WAF) near 1. RocksDB, however, can only achieve a minimum WAF of 2 with synchronous writes and WAL. This limitation led to the development of a new local storage engine, LavaLog, tailored for workloads that are highly sequential and require strong durability and low write latency. Despite this, both RocksDB and the new LavaLog experience significant tail latencies due to `fsync` operations on the standard kernel filesystem Ext4, severely impacting performance. Consequently, users like ByteNDB were forced to disable synchronous writes and rely on additional server replicas to maintain system durability, which significantly increased the cost of operations. To tackle these issues, there's a pressing need for a filesystem capable of delivering outstanding synchronous write performance along with ensuring strong durability.

3 LAVASTORE DESIGN

3.1 System Architecture

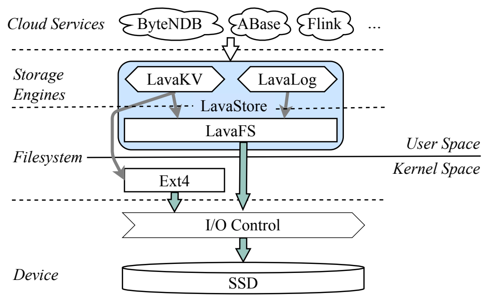


Figure 2: LavaStore Architecture

Considering various design choices and compromises, we created LavaStore at ByteDance— a purpose-built, high-performance, and cost-effective local storage engine. LavaStore stands out for its modular and flexible architecture, divided into an upper engine layer and a lower filesystem layer, as shown in Figure 2.

LavaKV serves as a direct substitute for RocksDB, targeting general write-heavy workloads. It incorporates a refined KV separation strategy, significantly reducing write amplification and tail latency, while delivering comparable or superior read performance to RocksDB. Additionally, LavaKV employs an adaptive GC approach to optimize the balance between space efficiency and write performance, considering the current disk utilization. As indicated by the gray arrows, LavaKV runs on top of either Ext4 or LavaFS.

LavaLog, entirely developed anew, caters specifically to log-style workloads with high sequentiality. It leverages an LSM-tree-based approach for metadata management, enabling a near 1 WAF with support for immediate durability. Distinct from RocksDB's WAL, which is append-only during writes and read-only during recovery, LavaLog also facilitates real-time searches and scans, making it suitable for applications like ByteNDB's log shipping. As indicated by the gray arrow, LavaLog runs on top of LavaFS only for now.

LavaFS, a user-space filesystem, is specially designed to accommodate the append-only write patterns of both LavaKV and LavaLog. By fully managing the I/O pathway within LavaStore, we have implemented enhancements that previously required kernel modifications, thus improving the performance of synchronous writes in ways that are not feasible with kernel filesystems like Ext4.

3.2 Optimizing Write Performance

This section outlines our strategies to enhance LavaStore's write performance, showcasing a distinct KV separation approach that diverges from RocksDB's `BlobDB`, achieving exceptionally low write amplification and reduced tail latency. Additionally, we introduce a specialized user-space, append-only filesystem LavaFS designed specifically for achieving low latency in synchronous writes.

3.2.1 LavaKV's KV Separation. Launched in 2019, LavaKV implemented a KV separation technique aimed at surpassing RocksDB in write throughput, making a deliberate trade-off for space efficiency in its design phase. This approach has led to outstanding write performance, along with competitive read functionality and greater flexibility in configuration. As illustrated in Figure 1c, LavaKV uniquely separates GC processes from compactions, with compactions merely updating information about garbage (e.g., the garbage ratio in blob files), while dedicated tasks execute the actual I/O operations for GC. This separation offers multiple advantages: it allows for even smaller compactions, facilitating quicker merges of SSTables to lower levels in high write-intensive scenarios and mitigating write stalls by speeding up L0 to L1 compactions. Moreover, it provides flexibility in tuning compactions and GC independently, enabling users to optimize for higher disk utilization or prioritize quicker write operations based on current disk usage and the balance of write and read demands.

In practical terms, LavaKV utilizes the same SSTable format for blob files, incorporating a fully sorted index within each blob file, as shown in Figure 1a. Therefore, only the blob file number is needed in the original SSTables, effectively reducing their file sizes. During GC, rewritten valid values in new blob files are tracked in a manifest file alongside old numbers, creating a manageable inheritance tree that's pruned periodically to maintain efficiency. During the GC process, new blob file numbers are documented in a manifest file along with their former numbers,creating a manageable inheritance

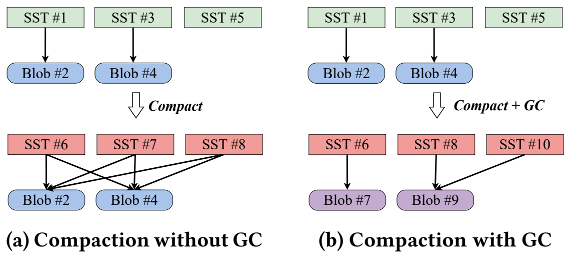


Figure 3: Resolving SSTable and blob dependencies.

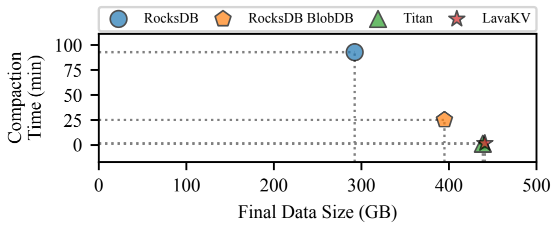


Figure 4: Evaluating the trade-offs in write performance and space utilization between RocksDB, Titan and LavaStore with KV separation (A - `Compact` in Table 1).

tree that's pruned periodically to maintain efficiency. Through this innovative structure, LavaKV has successfully decoupled blob GCs from compactions, attaining a write amplification that is 23% less than that seen in a finely tuned RocksDB.

3.2.2 On-demand Correlated Compaction and GC. Always executing blob GCs separately from SSTable compactions presents a significant challenge, as demonstrated in Figure 3a. For instance, consider a scenario where SSTables 1, 3, and 5 undergo compaction without including the blobs (2 and 4) they reference. This process redistributes keys from SSTable 1 that reference blob 2, and keys from SSTable 3 that reference blob 4, across the three resultant SSTables (6, 7, and 8), increasing the total reference count from 2 to 6 among them. This situation leads to two main issues:

- If blob 2 or 4 is later selected for GC, identifying still-valid keys becomes highly time-consuming as it necessitates reverse searches across more SSTables (6, 7, and 8), potentially taking minutes even hours for large datasets with numerous SSTables and blobs.
- Post-compaction, the keys in SSTables 6, 7, and 8 are sorted, enhancing scan performance due to improved locality. However, since the values in the blobs remain in their original, unsorted order, this negates the locality benefits from the SSTables, degrading the overall scan performance.

To address these challenges, LavaKV optionally performs blob file reconstruction during SSTable compactions. When SSTables are selected for compaction, their associated blobs are also chosen based on criteria like garbage ratio and blob overlapping ratio, aiming to reclaim space and reduce blob-SSTable links while enhancing sequentiality across blobs. During this SSTable compaction, selected blob files are compacted too, creating new blobs as depicted in Figure 3b, which aids in reducing the cost of reverse querying for future compactions and GCs, and improves blob locality to enhance scan performance. This approach is similar to that of `BlobDB`, but with a critical distinction: if a blob in `BlobDB` is not selected during an SSTable compaction, it must wait for the referenced SSTables to be compacted again, which could be a lengthy process. Conversely, in LavaKV, blobs can be garbage collected either independently or jointly with SSTable compaction, offering greater flexibility to adapt to changes in write throughput and space constraints.

3.2.3 Performance Results. Figure 4, detailed in Table 1 in Section 4.1.2, illustrates the trade-offs between write performance and space utilization among vanilla RocksDB, RocksDB `BlobDB`, Titan and LavaKV. Since `db_bench` does not directly measure write and space amplification, these are inferred from the total compaction time (with longer duration indicating higher write amplification) and the final data size post-compaction (where larger size suggests greater space amplification). Vanilla RocksDB shows the longest compaction time but the smallest data size, highlighting its focus on minimizing space use at the expense of write efficiency. In contrast, both Titan and LavaKV opt to use more disk space to achieve lower write amplification (they are almost identical), positioning `BlobDB` in a middle ground. This demonstrates LavaKV's aim for significantly reduced amplification, a goal not attainable by RocksDB.

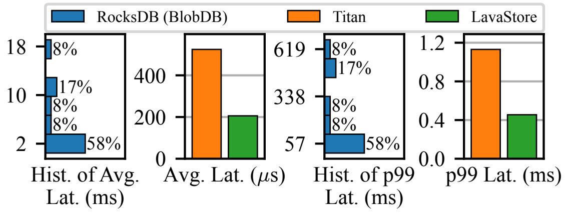


Figure 5: Comparing overwrite performance (B in Table 1).

Furthermore, Figure 5 compares performance in an overwrite-centric workload, also outlined in Table 1 in Section 4.1.2. With `BlobDB`, Titan and LavaStore operating under a uniform write rate limit of 60 MB/s, `BlobDB`'s GC operations during compactions are insufficient for handling large concurrent write traffic, leading to frequent write stalls in about half of the twelve runs. In its best runs, `BlobDB`'s average and tail write latencies are similar to Titan's, but both are significantly higher than LavaStore's. In the majority of runs, `BlobDB`'s latencies are several orders of magnitude higher than those of Titan and LavaStore. Titan performs better than `BlobDB` in most runs, but its reinsertion of KVs after compactions causes higher write amplification, resulting in average and tail latency numbers that are twice as high as those of LavaStore. This underscores the effectiveness of LavaStore's KV separation technique in enhancing write efficiency.

This design outperformed RocksDB and Titan in write efficiency but introduced extra challenges, including the need for precise adjustments in the standalone GC process to maintain write performance and the complex index searches required by LavaKV's blob files, which often led to increased tail read latencies. Addressing

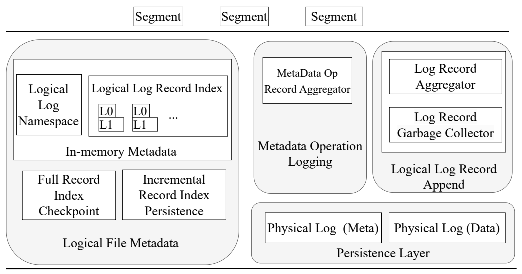


Figure 6: LavaLog Internals

these issues to enhance space utilization and read performance within LavaKV's KV separation framework and the solutions devised will be discussed in Sections 3.3 and 3.4.

3.3 Cost-Effectiveness Improvements

This section details how LavaLog reduces write costs for sequential log-style workloads, outlines the development of blob GC strategies to enhance space efficiency in LavaKV, and discusses the optimizations for synchronous writes in LavaFS.

3.3.1 Efficient Log Management with LavaLog. LavaLog provides a set of APIs for users to create, open, and delete log files (called *logical logs* in LavaLog's context). After a logical log is created and opened, a user can append new records to it by specifying an LSN and a buffer that holds log data. A user can also `trim` all stale data before a given LSN in a log. Additionally, a user can read the data with an LSN, or scan a log after a given LSN.

Design Goals. The design of LavaLog focuses on the following aspects to serve in production. First, LavaLog needs to accommodate log records with both sequential incremental LSNs and occasionally non-sequential LSNs. While most LSNs ingested to LavaLog are in order, some of them might be out of order, e.g., when a recovering replica is doing missing logs catch-up. The ability to handle log records with non-sequential LSNs distinguishes LavaLog from systems like LogDB [46]. Second, LavaLog should prioritize the performance of frequent operations. The most frequently used operations in LavaLog are append, `trim`, delete, and read. Third, LavaLog has to be resource efficient. LavaLog should avoid resource contention with other co-located services on the same machine. In fact, ByteNDB requires LavaLog to keep its memory consumption under 1% of the total log data size it stores. In addition, LavaLog needs to reclaim the space from trimmed records promply.

Overview. LavaLog logs user operations with two shared logs [3], i.e., *metalog* and *datalog*. The metalog logs `create`, `delete`, and `trim` operations, while the datalog records append operations. LavaLog buffers log operations from different users in a queue, while a background task coalesces and flushes the logs to the disk.

LavaLog stores the metalog and datalog in *physical logs*, which are files in filesystem with a configurable capacity (e.g., 256 MB). When a physical log outgrows the capcity, LavaLog seals that file, and opens a new one for writes.

Two-Level Indexing. To locate a user log record in physical logs, LavaLog maintains two mappings. The in-memory *namespace* maps logical log names to physical file IDs, and *metadata* maps an LSN to the location of that log record in a physical file. Because of the size of the metadata, LavaLog organizes it into a two-level index, as illustrated in Figure 7. LavaLog only keeps part of the metadata in memory as direct mapping (*L0*), while *MetaFlush* the rest mappings into a physical log. The indirect mapping (*L1*) maps an end LSN to the physical location of MetaFlush records, each of which encodes a group of mappings of $\langle lsn, log\_data\_location \rangle$.

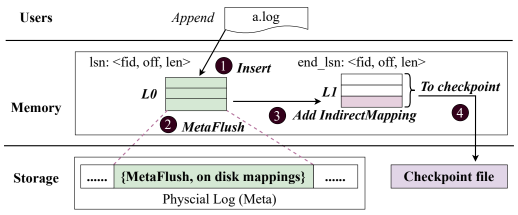


Figure 7: LavaLog Metadata Management

Checkpoint and Garbage Collection. LavaLog utilizes fuzzy checkpoints [70] to minimize user request obstruction. It checkpoints both *L1* and namespace data in the background for quick recovery. This involves merging incremental MetaFlush records from physical logs with a base checkpoint to create a new checkpoint.

LavaLog reclaims space in the datalog by moving valid logical log records from the old log files to the active physical log. When garbage exceeds a threshold, the oldest physical log is targeted for garbage collection. Instead of scanning the entire physical log, LavaLog uses a reverse index at the end of each physical log, mapping [*logical log id, LSN, offset, size*]. This allows quick validation of logical log records and locating valid records via offsets, significantly reducing read amplification.

Performance. Figure 8 compares the performance of immediate durable writes in RocksDB and LavaLog, where LavaLog has much lower average and tail latencies than RocksDB.

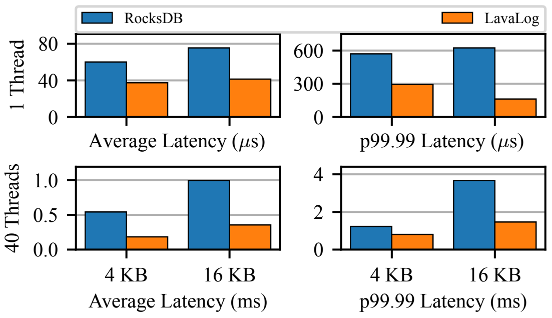


Figure 8: Latencies of writes with immediate persistence for RocksDB and LavaLog (WriteOnly in Section 4.1.3).

3.3.2 Optimizations for Space Utilization for LavaKV. We also introduce techniques to increase space efficiency of LavaKV. Specifically, we have implemented two techniques: calculating a blob file's garbage ratio to identify optimal candidates for space recovery (*Accurate GC*), and increasing the number of concurrent GC tasks through additional threads guided by an adaptive strategy based on disk usage (*Adaptive GC*). When disk usage is low, our approach focuses on minimizing write amplification by conservatively managing GC, delaying actions on files with lower garbage ratios until they meet set thresholds. Conversely, high disk usage prompts an increase in aggressive GC activities, utilizing more threads to maintain SLA standards, with GC thread priority adjusted below compaction tasks to balance operational needs. Figure 9 illustrates the relationship between GC thread count and garbage ratio as disk usage grows. In the optimized version, the number of GC threads begins to rise once disk usage surpasses the 75% threshold, effectively limiting the overall garbage ratio to approximately 13%, a significant improvement from the 20% observed in the prior, non-optimized version. As a result, disk usage stabilized at around 77%, leaving more usable space.

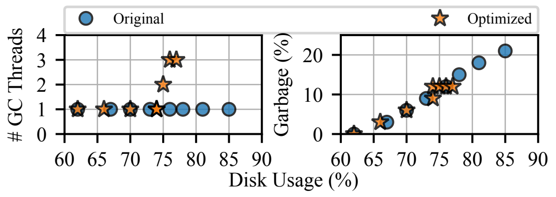


Figure 9: Scatter plot of the number of GC threads and overall garbage ratio versus disk usage (WriteOnly in Section 4.1.3).

Overall, our GC optimizations have improved space efficiency, demonstrated by write-only workloads, for which our GC techniques increase effective garbage ratio by 7% and reduces the number of GC occurrences by 30%. Moreover, peak disk usage is dropped by 8%, which increases the storage capacity provisioned to users by 8%, thus lowering storage costs.

3.3.3 *Optimizations for Synchronous Writes via LavaFS.*

To tackle the write performance issues mentioned earlier, we introduce the optimizations for synchronous writes in LavaFS.

Lightweight Journaling. Ext4 uses the jbd2 mechanism [34] for metadata journaling, where a transaction includes a header block (4 KB), one or more journal blocks (4*n* KB), and the transaction commit block (4 KB), as illustrated in Figure 10a. The `inode` grouping, together with the extra bookkeeping metadata blocks in jbd2 transaction records, introduce substantial write amplification, in particular for applications that frequently issue synchronous writes. LavaFS, on the other hand, employs a more lightweight journaling mechanism and commits metadata changes to disk by flushing them to a dedicated journal file only, without in-place metadata updates.

Synchronous Write Optimization. LavaStore further improves synchronous write performance through a customized `fdatasync` implementation. Although the concept of `fdatasync` itself is not novel, using `fdatasync` in kernel filesystems such as Ext4 for append-only writes behaves the same as using `fsync`, due to the need to synchronize file metadata to update file size after each append. In contrast, the `fdatasync` implementation in LavaFS does not flush file metadata for file size changes, unless there is also a change in file extents, e.g., when a new extent is allocated. Figure 10 compares the `fdatasync` implementations of Ext4 and LavaFS, suggesting a WAF of 8 and ~1, respectively, for 4 KB synchronous writes. Figure 11 shows the actual WAF measurement of ~6.7 and ~1, respectively, using 4 KB writes up to a 256 MB file. Figure 11 also shows that the `fsync` WAF of Ext4 is the same as `fdatasync`, and almost doubles for LavaFS due to the additional metadata synchronization in this case.

It is worth noting that with `fdatasync`, users can no longer rely on the `inode` information to retrieve the accurate file size (e.g., via `stat`), and must identify the boundary between valid and invalid data through other means. This is less of an issue, however, for the targeted applications that LavaFS supports. For instance, in case of recovery during `OpenDB`, LavaKV scans the extents of its WAL to validate record checksums. If the validation fails for a record, both the failed record and any subsequent records will be discarded.

Although it is possible to achieve effects similar to the `fdatasync` optimization entirely within LavaKV, e.g., by utilizing sparse files initialized to a large size, we note that decoupling the low-level, storage-related optimizations into a user-space filesystem has additional benefits. For one thing, these optimizations can be easily re-used to improve performance and cost for any other systems using LavaFS. For another, having a user-space filesystem makes it easier to adopt technologies such as SPDK and ZNS SSDs for the future evolution of LavaStore.

LavaFS Performance. We compare the performance of sequential write and random read, the two access patterns cared by LavaLog and LavaKV, with FIO tests on LavaFS and Ext4. The sequential write tests include two cases. As shown in Figure 12, the first case uses one thread to create 400 files and write 256 MB worth of data to each file consecutively, and each write is followed by an `fsync` to simulate the writes for a WAL. The second case uses 20 threads and each thread does the same work as the first case, to simulate SSTable writes in an LSM-tree KV store (i.e., writes that do not require immediate durability). The only difference is that each write is not followed by an `fsync`. The read tests use one thread or 20 threads to randomly read data from 400 pre-created files, and the total size of read data for each thread is 100 GB. For all tests, we vary the I/O size from 4 KB to 512 KB. To eliminate the effect of kernel page cache, both Ext4 and LavaFS use direct I/O.

In the case of sequential write operations, LavaFS outperforms Ext4 for single-threaded synchronous writes due to its much lower WAF. E.g., for 4 KB I/O, LavaFS has a WAF of 2 when using `fsync`, while Ext4 has a WAF of ~6.7, as shown in Figure 11. Additionally, for multi-threaded non-synchronous writes, LavaFS also exhibits superior performance compared to Ext4 thanks to its lower overhead due to kernel bypassing.

Regarding random read operations, both in single-threaded and multi-threaded tests, LavaFS shows a slight advantage over Ext4, particularly for small I/O sizes. However, in most scenarios, their performance remains comparable.

Figure 13 shows the write and read performance of LavaKV on LavaFS and Ext4 with `db_bench`. In both workloads, LavaKV performs better on LavaFS than Ext4 with lower average and tail latencies. Compared with Ext4, LavaFS has in-memory metadata.

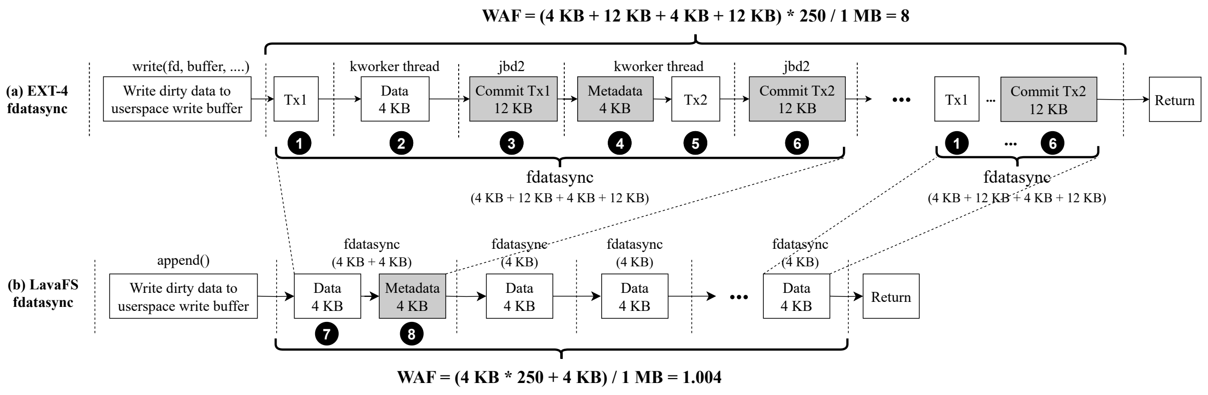


Figure 10: 1MB appending `fdatasync` Ext4 v.s. LavaFS.
(a) Ext4 `fdatasync` regresses to `fsync` with WAF = 8; (b) LavaFS `fdatasync` with WAF ≈ 1.
Sync write operations. ① and ⑤: Add `inode` updates to journal transactions (Tx1 and Tx2); ② and ⑦: Write dirty data to disk; ③ and ⑥: Commit transaction, 12 KB (4 KB transaction header + 4 KB log block + 4 KB commit block); ④ Update Ext4 on-disk `inode` metadata; and ⑧ Write `inode` update transaction log to disk.

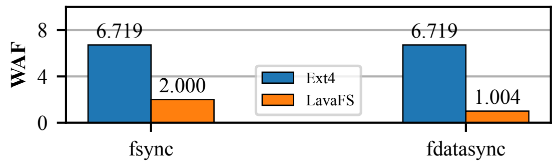


Figure 11: Actual WAF comparison of `fsync` and `fdatasync` between Ext4 and LavaFS for 4 KB writes up to a 256 MB file.

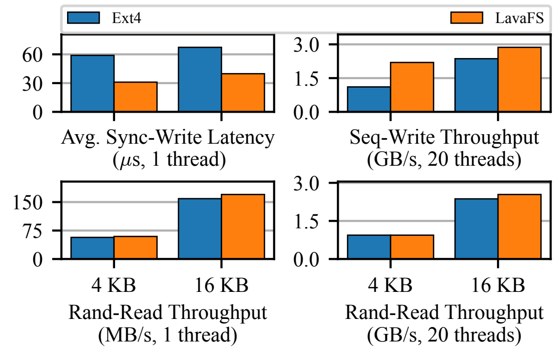


Figure 12: FIO performance comparison for Ext4 and LavaFS.

Therefore, its read and write paths do not require metadata lookup and thus reduce average read and write latency. The better write performance also stems from the asynchronous journal compaction and `fdatasync`-based co-design.

3.4 Read Performance Optimizations

Our aim was to reduce the I/O cost for a `Get` query to 1, meaning that the index and data blocks from SSTables, as well as the blob index, should be accessible directly from the block cache. Thanks

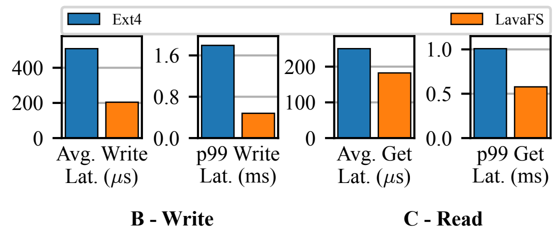


Figure 13: Write and read performance comparison of LavaKV on different filesystems (B and C in Table 1).

to KV separation leading to reduced SSTable sizes and an efficient compaction strategy, entire SSTables should easily fit into the block cache. Nonetheless, the fully sorted index in LavaKV's blob files is too extensive and redundant, mirroring SSTable indexes and proving useful only during GCs.

To integrate the blob index into the block cache efficiently, we utilized a memory-efficient Crit-bit Tree (CBT) [39] index. The CBT index, using just 2.3 bytes per key, incurs a negligible space overhead of only 1.6‰ according to our `db_bench` test. This is significantly more efficient than the space overhead reported by RocksDB for a similar-purpose Data Block Hash Index [54] (~4.6%), making the CBT index only $\sim\frac{1}{287}$ of this size. It facilitates efficient blob file lookups, enabling the quick retrieval of desired values. As a result, caching only needs to accommodate the CBT index. In contrast to RocksDB's approach, which uses a 24-byte triplet, LavaKV's method employs only 11 bytes on average (including an 8-byte blob file number common to both RocksDB and LavaKV), resulting in a better cache hit ratio and improved read performance.

In pursuit of optimizing block cache efficiency, we implemented an innovative cache system known as the *unified cache*, which operates atop any existing caches, including RocksDB's built-in clock and LRU caches. This system uniquely allocates a specific segment for indexing blocks, enabling dynamic size adjustment

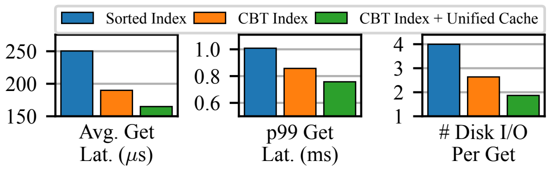


Figure 14: Comparing Get performance using the CBT index and unified cache (C in Table 1) with LavaKV (I/O size 4 KB).

between index block and data block caches to optimize storage. To guarantee that each `Get` request results in no more than one disk I/O, indexing blocks are given priority in caching due to their higher importance. This prioritization is facilitated by the introduction of the CBT index for blob files. The integration of the CBT index alongside the unified cache system has led to a notable reduction in both average and tail `Get` latencies by approximately 20% to 30%, as well as a reduction in the average disk I/O for a `Get` operation from 4 to below 2, even in scenarios that favor RocksDB over LavaStore, enhancing overall efficiency and performance in data retrieval (as shown in Figure 14). It is noteworthy that this unified cache can function independently of KV separation, although the performance gains are not as pronounced as when used in conjunction with KV separation and the CBT index.

4 PERFORMANCE EVALUATION

This section begins with a comparison of end-to-end I/O performance between upstream RocksDB and LavaStore using two standard benchmarks, `db_bench` and `SysBench`, in Section 4.1. We then assess the performance of LavaStore in our real-world production environment, with ABase and Flink, in Section 4.2.

4.1 Standard Benchmarks

4.1.1 Hardware Environment.

We use machines with identical hardware configurations to evaluate the performance of RocksDB, Titan and LavaStore. Each test machine features an Intel Xeon Silver 4314 CPU with 64 cores, 377 GB of DRAM, and 5 Intel SSDPF2KX038TZ NVMe SSDs, each with a capacity of 3.84 TB. The test machines all run a 64-bit Debian 10 operating system with Linux kernel version 5.15.103. Both `db_bench` and `SysBench`, which will be discussed in 4.1.2 and 4.1.3 respectively, use these same hardware equipment.

4.1.2 Performance Evaluation using `db_bench`.

Configurations. We employ `db_bench`, the benchmarking tool commonly used for assessing the performance of RocksDB and its derivatives, to evaluate and compare the performance of RocksDB version `8.5.3` and Titan version `7.1` (the latest versions as of this writing) with LavaStore. For all our tests, we utilize the original `benchmark.sh` script from RocksDB version `8.5.3`, as referenced in the RocksDB Wiki [56]. Our evaluation compares the performance about three types of operations: write operation, read operation, and short/long-range scans.

We conduct tests across 5 distinct workloads based on the original `benchmark.sh` script with default parameters, as detailed in Tables 1 and 2. It is worth noting that the `USE_O_DIRECT` flag is set to enable direct I/O for read, flush, and compaction in RocksDB, thereby avoiding the use of the system's page cache, to ensure a fair comparison with LavaStore, which operates in direct I/O mode. This is because in our production environment, a larger block cache is prioritized over the page cache. In all workloads except A, the `MB_WRITE_PER_SEC` parameter is limited to 60 MB/s to restrict the write rate and prevent write stalls, with a maximum running time of 30 minutes. In Table 2, the SSTable cache is a `SecondaryCache` in RocksDB and a block cache in Titan, storing SSTable blocks. Conversely, in LavaStore, it stores both SSTable blocks and the blob CBT indexes. The blob cache stores blocks of blob files in RocksDB and Titan, whereas in LavaStore, it only stores the values of blobs. Averages are based on twelve runs for RocksDB, and six runs for Titan and LavaStore. RocksDB required more runs due to unstable write performance, while the other two showed consistent results.

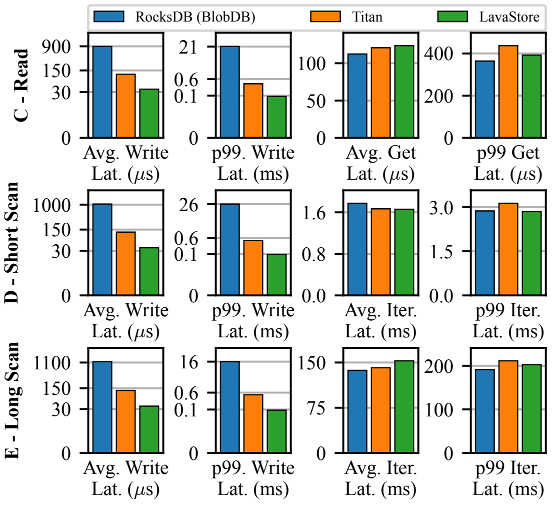


Figure 15: Read / Scan while Writing Workloads Comparison (Y-axes of the first two columns are log-scaled)

Performance Results. We have the following three observations:

(1) *Write Performance*: Similar to workload B, LavaStore outperforms RocksDB and Titan in workloads C, D, and E. LavaStore's average and 99th percentile write latencies are 67% to 80% lower than Titan's and several orders of magnitude lower than RocksDB's. Note that RocksDB's numbers are averages from twelve runs, but actual values fluctuate significantly, similar to Figure 5. The write performance gain demonstrates that we have successfully achieved our design goal of handling highly write-intensive workloads.

(2) *Read Performance of Lookups*: In the given settings, all data except for the values in blobs can be fully cached in the SSTable cache. As a result, retrieving a value typically requires only one disk read in a blob file. Consequently, the average `Get` latencies for RocksDB, Titan, and LavaStore are similar, with RocksDB being marginally faster. However, LavaStore supports cache warming by proactively caching

Table 1: Summary of db_bench workloads.

	Name	benchmark.sh Workload	Description
A	Load	`bulkload`	1 thread doing batched writes without syncing and WAL (`fillrandom` workload), followed by manual compaction (`compact` workload) for fast preconditioning.
B	Write	`overwrite`	16 threads doing random `Put`'s.
C	Read	`readwhilewriting`	16 threads doing random `Get`s, while 1 thread doing random `Put`s.
D **E**	Short Scan Long Scan	`fwdrangewhilewriting`	16 threads doing random iterator `Seek`s, followed by 10 or 1000 iterator `Next`s, while 1 thread doing random `Put`s.

Table 2: RocksDB, Titan and LavaStore key parameters in db_bench. Other parameters are kept default.

Parameter	Value
Key / Value Length	36 / 16,000 bytes
Compression	50% ratio; `ZSTD`
Total Operations	68,565,205
Total Raw KV Size	1 TB (500 GB compressed)
KV Separation Threshold	512 (`min_blob_size`)
Total Cache Size	1 GB (1:500 cache-to-total data size ratio)
SSTable Cache	800 MB, compressed, LRU
Blob Cache	224 MB, uncompressed, LRU

KVs during compactions, which results in a slightly better 99th percentile latency compared to Titan.

(3) *Read Performance of Range Scans*: LavaStore performs comparably or even slightly better than RocksDB and Titan in short-range scans but is less efficient in long-range scans, as illustrated in the third row of Figure 15. This is because our KV separation design, unlike RocksDB, decouples GC and compaction to achieve higher write throughput. However, this leads to less frequent compactions (Titan is similar), resulting in poorer locality for range queries compared to RocksDB. We note, however, that cloud services at ByteDance rarely rely on range queries, so this does not adversely impact LavaStore's performance in production.

4.1.3 *Performance Evaluation using* `SysBench`.

While `db_bench` can evaluate the I/O performance of single local storage engines, it is also important to conduct a broader evaluation of LavaStore in a cluster environment. Thus, we used `SysBench` [38] to benchmark a ByteNDB cluster comprising 3 nodes, each equipped with 5 NVMe SSD drives. These drives supported either 5 RocksDB instances on Ext4 or 5 LavaStore instances. Our setup handled 6 datasets, each containing 250 tables with 25 million rows of 200 bytes each, as detailed in Section 4.1.1.

The `SysBench` test has two workloads: `WriteOnly` workload employs the `oltp_write_only` script to write all 25 million rows to all tables; and `ReadWrite` utilizes the `oltp_read_write` script to both read and write to all rows in all tables. This setup allowed us to showcase the performance gains achieved with LavaStore, including the purpose-built LavaLog, comparing outcomes from persistent volumes and ByteNDB clients across these distinct storage solutions in a single, comprehensive paragraph.

Table 3: Comparison of `WriteOnly` and `ReadWrite` on ByteNDB using `SysBench`.

	WriteOnly		ReadWrite	
	RocksDB	LavaStore	RocksDB	LavaStore
`WriteLog` (ms)	3.6	1.4	1.4	0.7
`ReadPage` (ms)	N/A	N/A	1.2	1.0
DB Lat. (ms)	3.8	3.6	62.3	56.5
DB TPS (×1000)	200.5	212.5	12.3	13.6
DB QPS (×1000)	1202.8	1275.3	246.6	271.6
E2E WAF	7.3	5.6	35.94	20.25

Table 3 presents the performance metrics for two essential internal operations, `WriteLog` and `ReadPage`, comparing RocksDB and LavaLog specifically in log-style workloads. Additionally, it includes common end-to-end performance indicators such as throughput, latency, and overall WAF for a comprehensive assessment.

The table reveals that, compared to RocksDB, LavaLog decreases `WriteLog` latency by 61% in the `WriteOnly` workload and 48% in the `ReadWrite` workload, showcasing LavaLog's efficient design that targets reducing write amplification from at least 2 to nearly 1. Additionally, the `ReadPage` operation witnesses a 16% reduction in average latency during the `ReadWrite` test. This improvement highlights LavaLog's effective read path implementation, especially when compared to reading SSTables in such specific workloads.

From a comprehensive end-to-end viewpoint, the overall average latency of combined operations sees a reduction of 6% in the `WriteOnly` workload and 10% in the `ReadWrite` workload. Thus, the database's overall throughput, in terms of Transactions Per Second (TPS) and QPS, experiences enhancements of 6% and 10%, respectively. This more pronounced improvement in the `ReadWrite` workload aligns with observations from similar `db_bench` workloads, such as C—E, indicating that LavaStore delivers superior write performance, particularly under heavy read pressure. Furthermore, the end-to-end WAF for ByteNDB shows a substantial decrease of 24% in `WriteOnly` and 44% in `ReadWrite` tests, further highlighting the impact of our optimization efforts on write efficiency.

4.2 Production Performance Results

In Figure 16, LavaStore demonstrates marked enhancements for Abase in production, which maintains ~100 GB to ~1 TB data per instance on average, with a read:write ratio of 3.6:1 and an average value size of ~1 to ~2 KB. The efficient KV separation of LavaStore leads to an 87% increase in overall QPS for write (`Set`) operations, nearly doubling the performance compared to RocksDB.

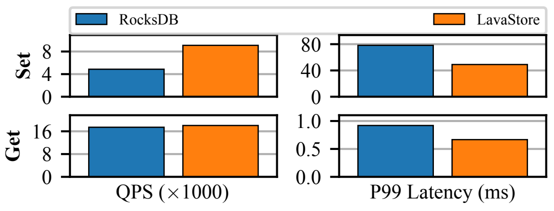


Figure 16: RocksDB v.s. LavaStore on ABase.

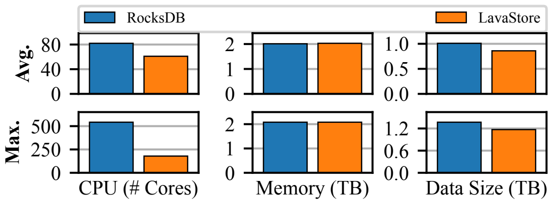


Figure 17: RocksDB v.s. LavaStore on Flink.

Furthermore, tail write latency sees a 38% reduction, thanks to synchronous write optimizations jointly engineered in LavaKV and LavaFS. Regarding read (`Get`) performance, LavaStore achieves a comparable QPS to RocksDB, with a slight 3% improvement. But more importantly, the CBT index and the unified cache significantly reduce tail read latency by 28%, underscoring the combined efficacy of these technologies in supporting lookup-centric workloads.

Figure 17 compares LavaStore and RocksDB for a Flink online joiner of an advertising service in production, which features a multi-instance setup with ~6 GB data per instance and ~3000 instances per server, with a read:write:delete ratio of 1:1:1 and an average value size of ~70 KB. Compared with RocksDB, LavaStore enjoys a 26% and 67% reduction in average and peak CPU usage, respectively. This is primarily due to LavaStore's more efficient KV separation, which significantly reduces both the frequency and the size of CPU-heavy compactions. While average memory usage of LavaStore sees a minor increase of 1%, peak memory usage remains unchanged. Furthermore, both average and peak data sizes are reduced by 15% using LavaStore, highlighting the effectiveness of our optimized GC strategies in enhancing space efficiency.

5 DISCUSSIONS

5.1 Lessons Learned

Specializing vs. Generalization: While common configurations like RocksDB suffice for many scenarios, specific cases highlight performance issues, such as high write amplification and significant tail latency. Databases that demand high write throughput and immediate disk flush, while maintaining read performance and capacity efficiency, benefit from specialized engines designed for particular I/O streams. These tailored solutions address performance challenges more effectively than a one-size-fits-all approach.

Building on Existing Foundations: Though crafting specialized software can offer performance gains, simplify the software stack, and enhance code quality, it introduces significant challenges. These include the substantial effort required for development from the ground up, as opposed to modifying existing systems. This approach also bypasses the benefits derived from mature testing frameworks, well-established benchmarks, and a plethora of utility tools inherent in existing solutions, which are crucial for maintaining high stability standards, especially for infrastructure software like local storage engines where stability is often paramount.

Avoid Over-engineering: Efficient resolutions often emerge from uncomplicated and nimble approaches. For example, although KV separation effectively lowers write amplification during compaction, it inherently requires a balance between read performance and space usage. Our experience has shown that rather than resorting to intricate models, employing a straightforward strategy based on disk usage to dynamically regulate GCs can adeptly manage this balance, thereby enhancing overall performance.

Emphasizing Modular Design: Maintaining a comprehensive, yet modular design is crucial. This approach facilitates adaptability to emerging challenges. As priorities shifted from write throughput and space utilization to optimizing for lookup tail latency, we were compelled to reevaluate certain design and implementation aspects of LavaKV's KV separation strategy. Modularity is thus indispensable for facilitating such evolutionary development processes.

5.2 Future Directions

Evolving Local Storage Engine Optimization: Future enhancements may involve categorizing query requests into point lookup queries and range queries, each handled by distinct algorithms. This differentiation allows for optimizing sequential chunks, thereby improving write amplification and Total Cost of Ownership (TCO).

***Refining LavaLog's GC:*.** The current FIFO-based GC strategy in LavaLog shows limitations with out-of-order `Trim` requests from applications. We plan to develop and incorporate new GC strategies that intelligently adapt to varying `Trim` patterns, moving beyond a sole reliance on FIFO.

Adapting to New Storage Technologies: Data streams with a sequential-intensive nature stand to benefit from append-optimized filesystems and storage devices. A potential avenue for exploration is the integration of append-friendly technologies like ZNS [7] SSDs, explicitly designed to cater to log-style data streams.

***Optimize Cross-layer GC:*.** Our efforts have aimed at unifying GC processes at the KV and filesystem layers. However, considering the SSD still operates its own GC, which may introduce additional write amplification and impact performance and device lifespan, future work will focus on enhancing coordination across all layers, especially leveraging advancements in ZNS and SPDK technologies.

6 RELATED WORK

Significant research has aimed at improving write, read, and space efficiencies in LSM-tree-based KV stores like RocksDB. Key studies in [50, 71, 72] examine the trade-offs among these aspects, providing a thorough overview of applicable techniques. Our discussion narrows down to related work in the following four specific categories, focusing on the optimization of LSM-tree-based systems.

Improving Write Performance. KV separation has emerged as a prevalent technique for boosting write performance within database systems. Initially introduced by WiscKey [48] and later adopted by platforms such as [13, 21, 43, 55, 84], this approach has been effective in reducing write amplification to some extent. Despite its effectiveness, these implementations fall short of achieving the ultra-low levels of write amplification desired in some applications. This shortfall is largely due to the necessity of integrating blob GCs with compactions, which leads to a compromise: either the write amplification remains higher than ideal or there is an increase in space amplification. Consequently, while KV separation marks a significant advancement in minimizing write amplification, the challenge of optimizing it alongside space efficiency persists, highlighting the need for further innovations in this area. Various alternative approaches focus on the improvement of compaction methodologies, as explored in [18, 20, 23, 33, 52, 63, 78]. These strategies often require a compromise between space or processing time to boost write efficiency. Although such methods might not reduce write amplification as significantly as KV separation does, they serve as valuable adjuncts to KV separation. By integrating these compaction strategies with KV separation techniques, it is possible to achieve a more refined equilibrium among write speed, read performance, and efficient use of storage space.

Improving Read Performance. In LSM architectures, where queries may necessitate accessing multiple SSTables, filters like [19, 22, 28, 44, 69, 75, 86] are frequently employed to limit the SSTables accessed. These mechanisms generally exchange space, memory, or CPU resources to diminish read amplification from SSTable access for lookup queries. However, they fall short in alleviating the read costs associated with accessing blob files in KV separated LSM trees. Other widely used techniques involve optimizing block cache data structures to enhance data locality and minimize resource contention caused by concurrency control mechanisms to improve upon or replace the commonly used LRU cache [10, 25, 27, 32, 68, 79], which are orthogonal to our unified cache. Additionally, specialized hash indexes and partitioning strategies are employed for workloads with stringent performance demands for lookup queries [2, 5, 16, 41, 47, 82]. We are also investigating techniques that could improve range scan performance, areas where LavaStore ties or even lags behind RocksDB. This includes exploring technologies such as [37, 51, 58, 76, 80], which leverage filters to boost short range query efficiency. Additionally, REMIX [85] offers a redesigned index structure that presents a globally sorted view of KV data across multiple table files, aiming to enhance the efficiency of range queries. FenceKV [74] introduces a key-range GC strategy to reduce GC overhead and ensure sequential access for range queries in KV separated LSM trees. Despite the potential of these advancements, they have not been the main focus of our efforts due to their marginal applicability to the core workloads of our direct users, who seldom require long range scans.

Handling Special Workloads. Local storage engines are faced with a diverse range of unique workloads that require specialized management. For instance, several studies tackle the challenges posed by sequential, log-style workloads, including WAL in RocksDB [14, 35, 36, 45, 46, 73]. However, these methods either incur a WAF of at least 2 due to writes to WALs and SSTables, or they necessitate strictly sequential data. As a result, they either underperform or fail to fulfill the durability criteria of LavaLog. Others examine Write-Once-Read-Many (WORM) workloads essential for storage engines [1, 29, 30, 64]. Additionally, there are proposals for data structures, algorithms, and implementations tailored to multi-dimensional data like time series and spatial-temporal data, which are crucial for analytic applications [49, 53, 60, 65]. Beyond these specific workload types, [17, 18, 31, 40, 59, 83] are dedicated to adjusting system parameters to optimize performance for a range of changing and diverse workloads. These efforts are complementary to our own and could potentially be leveraged to broaden the customer base of LavaStore.

Cross-layer Storage Stack Optimization. Cross-layer storage stack optimization is essential for reducing redundant and uncoordinated overheads, such as compaction and GC, thereby enhancing disk efficiency and durability, as highlighted in a recent survey [24]. For instance, RStore [42] leverages modern storage technologies to achieve low tail latency in multi-core KV stores. exF2FS [61] introduces transaction support for log-structured filesystems, showing notable performance and scalability improvements. ScaleStore [87] offers a fast and cost-effective storage engine utilizing DRAM, NVMe, and RDMA. Viper [4] combines a DRAM-based hash index with a PMem-aware storage layout to capitalize on DRAM's rapid random-write speed and PMem's efficient sequential-write capability. Research by [43] explores differentiated KV storage management to balance I/O performance. SpanDB [14] employs high-speed SSDs selectively, allowing data placement across different SSD tiers based on TCO and SLA. [66] delves into LSM management within computational storage environments. Our approach, tailored to specific needs, shows that our append-friendly LavaFS improves write performance in local storage engines.

7 CONCLUSION

This paper presents LavaStore, a high-performance cost-effective local storage engine purpose-built for cloud services at ByteDance. We demonstrate how to effectively handle a highly write-intensive workload with stringent requirements on resource efficiency and point lookup tail latency, by strategically customizing components of a RocksDB-based general purpose local storage engine. Although our implementation is based on RocksDB, the design techniques employed by LavaStore could be generally applicable to other LSM-tree-based persistent KV stores to handle workloads with similar characteristics and requirements.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. We are grateful to our shepherd for the extensive help in revising the paper. The CBT index is implemented by Guoshun Ma utilizing some components from the original version of TerarkDB. We would like to thank all those who have contributed to the design and development of LavaStore, in particular Changlong Chen, Yunxiao Du, Kuankuan Guo, Peng Lei, Jianchuan Li, Yuanjin Lin, Yangming Liu, Yu Miao, Chi Zhang and Maosen Zhang.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 707–722.

[2] Nikolas Askitis and Ranjan Sinha. 2007. HAT-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*. Citeseer, 97–105.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed data structures over a shared log. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 325–340.

[4] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid PMem-DRAM key-value store. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1544–1556.

[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 521–534.

[6] Ken Birman. 2007. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review* 41, 5 (2007), 8–13.

[7] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 689–703.

[8] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running*. O'Reilly Media, Inc.

[9] ByteDance. 2019. TerarkDB. Retrieved January 18, 2024 from https://github.com/bytedance/terarkdb

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

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[12] Josiah Carlson. 2013. *Redis in action*. Simon and Schuster.

[13] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[14] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.

[15] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: ByteDance's HTAP system with high data freshness and strong data consistency. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3411–3424. https://doi.org/10.14778/3554821.3554832

[16] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[18] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.

[19] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[20] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[21] Dgraph. 2019. BadgerDB. Retrieved January 18, 2024 from https://dgraph.io/docs/badger/

[22] Peter C Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *arXiv preprint arXiv:2103.02515* (2021).

[23] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. 2022. TriangleKV: Reducing write stalls and write amplification in LSM-tree based KV stores with triangle container in NVM. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4339–4352.

[24] Krijn Doekemeijer and Animesh Trivedi. 2022. Key-Value Stores on Flash Storage Devices: A Survey. *arXiv preprint arXiv:2205.07975* (2022).

[25] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[26] Paul DuBois. 2013. *MySQL*. Addison-Wesley.

[27] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[28] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[29] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), 3–1.

[30] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.

[31] Yichen Jia and Feng Chen. 2020. Kill two birds with one stone: Auto-tuning RocksDB for high bandwidth and low latency. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 652–664.

[32] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.

[33] Peiquan Jin, Jianchuan Li, and Hai Long. 2021. DLC: A New Compaction Scheme for LSM-tree with High Stability and Low Latency.. In *EDBT*. 547–557.

[34] The kernel development community. 2023. Journal (jbd2). Retrieved January 18, 2024 from https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html

[35] Jongbin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2019. Border-Collie: a wait-free, read-optimal algorithm for database logging on multicore hardware. In *Proceedings of the 2019 International Conference on Management of Data*. 723–740.

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

[37] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*. 1670–1684.

[38] Alexey Kopytov. 2012. SysBench manual. *MySQL AB* (2012), 2–3.

[39] Adam Langley. 2008. Crit-bit Trees. Retrieved January 18, 2024 from https://www.imperialviolet.org/binary/critbit.pdf

[40] Jieun Lee, Sangmin Seo, Jonghwan Choi, and Sanghyun Park. 2024. K2vTune: A workload-aware configuration tuning for RocksDB. *Information Processing & Management* 61, 1 (2024), 103567.

[41] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[42] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1091–1104.

[43] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.

[44] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 739–752.

[45] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 149–166.

[46] LogDevice. 2023. LogDB. Retrieved January 18, 2024 from https://logdevice.io/docs/LogsDB.html

[47] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2021. TridentKV: A read-Optimized LSM-tree based KV store via adaptive indexing and space-efficient partitioning. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1953–1966.

[48] 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 Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[49] Ziyi Lu, Qiang Cao, Hong Jiang, Shucheng Wang, and Yuanyuan Dong. 2022. p^{2}KVS: a portable 2-dimensional parallelizing framework to improve scalability of key-value stores on SSDs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 575–591.

[50] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[51] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[52] Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J Tsotras, and Neal E Young. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. *The VLDB Journal* 30 (2021), 361–378.

[53] Qizhong Mao, Mohiuddin Abdul Qader, and Vagelis Hristidis. 2023. Comparison of LSM indexing techniques for storing spatial data. *Journal of Big Data* 10, 1 (2023), 51.

[54] Meta. 2018. RocksDB Wiki: Data Block Hash Index. Retrieved January 18, 2024 from https://github.com/facebook/rocksdb/wiki/Data-Block-Hash-Index

[55] Meta. 2022. RocksDB Wiki: BlobDB. Retrieved January 18, 2024 from https://github.com/facebook/rocksdb/wiki/BlobDB

[56] Meta. 2022. RocksDB Wiki: Performance Benchmarks. Retrieved January 18, 2024 from https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks

[57] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[58] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2022. bloomRF: On performing range-queries in Bloom-Filters with piecewise-monotone hash functions and prefix hashing. *arXiv preprint arXiv:2207.04789* (2022).

[59] Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, and Manos Athanassoulis. 2022. LSM-Tree Under (Memory) Pressure. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[60] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and InfluxDB. *Studienarbeit, Université Libre de Bruxelles* 12 (2017), 1–44.

[61] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. 2022. exF2FS: Transaction Support in Log-Structured Filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 345–362.

[62] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385.

[63] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45 (2017), 1310–1325.

[64] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[65] Gotze Philipp, Baumann Stephan, and Sattler Kai-Uwe. 2018. An NVM-aware storage layout for analytical workloads. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 110–115.

[66] Ivan Luiz Picoli, Philippe Bonnet, and Pinar Tözün. 2019. LSM management on computational storage. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–3.

[67] PingCAP. 2023. Titan. Retrieved January 18, 2024 from https://github.com/tikv/titan

[68] Aleksandar Prokopec. 2018. Cache-tries: concurrent lock-free hash tries with constant-time operations. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 137–151.

[69] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[70] Kenneth Salem and Hector Garcia-Molina. 1989. Checkpointing memory-resident databases. In *ICDE*. 452–462.

[71] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, designing, and optimizing LSM-based data stores. In *Proceedings of the 2022 International Conference on Management of Data*. 2489–2497.

[72] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3578–3584.

[73] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. 2023. Real-time LSM-trees for HTAP workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1208–1220.

[74] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. FenceKV: Enabling efficient range query for key-value separation. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3375–3386.

[75] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2011), 131–155.

[76] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.

[77] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[78] Xiaoliang Wang, Peiquan Jin, Bei Hua, Hai Long, and Wei Huang. 2022. Reducing write amplification of LSM-tree with block-grained compaction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3119–3131.

[79] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[80] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[81] Jianshun Zhang, Fang Wang, Sheng Qiu, Yi Wang, Jiaxin Ou, Junxun Huang, Baoquan Li, Peng Fang, and Dan Feng. 2024. Scavenger: Better Space-Time Trade-Offs for Key-Value Separated LSM-trees. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4072–4085.

[82] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward high-performance and scalable KV storage in mixed workloads via unified indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 313–324.

[83] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2022. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–30.

[84] Zhiquan Zheng. 2019. Titan: A RocksDB Plugin to Reduce Write Amplification. Retrieved January 18, 2024 from https://www.pingcap.com/blog/titan-storage-engine-design-and-implementation

[85] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.

[86] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing Bloom filter CPU overhead in LSM-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware*. 1–10.

[87] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data*. 685–699.