Deployed System: LavaStore: The Route to a Customized Storage Backend for Database Page Servers

Abstract

A cloud-scale database management system (DBMS) usually utilizes disaggregated pageservers to store clients' data. A pageserver is in charge of materializing pages from redo logs shipped to it, indexing materialized pages, and responding to page requests from the compute layer of the DBMS. The I/O and space efficiency of pageservers are paramount to the overall functionality of the cloud DBMS. At CompanyX, we initially utilized an in-house LSM-tree key-value store TxxxxxDB on top of a generic filesystem like Ext4 to manage the different types of data stored on pageservers. Though generic and having expedited the production development, the "TxxxxxDB + Ext4" solution tends to deliver less-than-expected performance, such as long tail latencies and high write amplification, disregarding that we equipped each pageserver with high-end NVMe SSDs. We have also noticed the latency in the storage backend of pageservers contributes to 25-50% of the end-to-end latency of database workloads. Thus, improving the performance of the storage engine on pageservers is crucial to enhance the database user experience.

To overcome the performance issues of the storage engine on pageservers, we have substituted the "TxxxxxDB + Ext4" solution with a specialized storage engine named LavaStore. LavaStore highly optimized TxxxxxDB to store database pages efficiently and introduces a specialized engine for log-style data called LogEngine. LavaStore reduces write and space amplification of TxxxxxDB and maintains good read performance. With LogEngine, LavaStore stores log data with a high durability guarantee and low latency. LavaStore also includes a userspace filesystem to offer efficient file access for TxxxxxDB and LogEngine. Our evaluation results show that LavaStore outperforms the "TxxxxxDB + Ext4" solution for a suite of micro-benchmarks; for pageservers, LavaStore reduces the average latency of persisting redo logs by 61%, and the average latency of retrieving a data page by 16%. For database workloads, LavaStore increases the end-to-end throughput of CompanyX's cloud database BxxxXDB by 6%, and reduces average transaction latency by 10%.

1 Introduction

Leading cloud-native database management systems, such as Amazon Aurora [49], adopt a disaggregated architecture, in which data pages are stored in dedicated storage nodes (i.e., *pageserver*). For high availability, the pageservers of a cloud database collectively form a quorum-based cluster. From a high level, the activities on a pageserver mainly include three parts: (1) receiving and persisting redo logs from the compute layer; (2) organizing log records and identifying gaps in the log and gossiping with peers to fill in gaps; and (3) coalescing log records into new data pages, and responding to page requests from the compute layer. The performance of pageservers is central to a cloud DBMS as a whole. At CompanyX, we implemented an Aurora-like database named "BxxxXDB" (Sec. 2.1). Initially, the pageserver of BxxxXDB used an internally-maintained RocksDB [36], called "TxxxxxDB" [1] (Sec. 2.2), on top of Ext4 to store redo logs and materialized data pages as well as deltas to these data pages. This choice accelerated the productization process of BxxxXDB by leveraging the friendly key-value (KV) store (KVStore) interfaces and simple semantics of TxxxxxDB.

Despite the "TxxxxxDB + Ext4" solution expedites product development, it suffers from several problems. (**P1**) **Unnecessary cost for ingesting sequential data**. TxxxxxDB introduces long latencies when clients ingest highly sequential data that requires immediate durability (e.g., write-ahead or raft log). Storing such log data in an LSM-tree [41] introduces unnecessary overheads, such as log-on-log, MemTable insertion, and compaction overheads. (**P2**) **High write amplification**. Although LSM-trees typically exhibit significantly lower write amplification than B-trees [43, 45], applications often demand even further reductions in write amplification stemming from LSM internal flushes and compactions [15], especially given that our KVStores exclusively operate on SSDs. SSDs entail an internal garbage collection (GC) process that further contributes to write amplification. Hence, it remains imperative to minimize write amplification for better write performance and lower hardware expenses. (**P3**) **Resource contention**. The background compaction in an LSM-Tree-based KVStore is both I/O and CPU intensive; the compaction jobs could contend CPU, I/O, and memory resources with the foreground jobs, thus hurting user experiences. A canonical way of mitigating these two problems (P2 and P3) is to separate keys and values in the LSM-tree, known as KV separation. However, previous implementations of KV separation by WiscKey [35], RocksDB [36], and Titan [42], have some limitations. For instance, in these systems, the GC mechanisms are inefficient, unable to subside peak storage consumption in time, and even obstruct the clients' write requests when the GC process is activated. (**P4**) **Inferior read performance**. As a write-optimized dictionary, an LSM-tree does not offer high performance for reads. KV separation necessitates one extra read for retrieving values, further diminishing query performance. Real-world deployment has witnessed extremely long tail latency for point lookups (e.g., 100s of milliseconds) in OLTP databases such as BxxxXDB. Standard optimizations, such as bloom filters [48] and data block hash index [38], enhance read performance but come with the trade-off of increased disk and memory consumption. According to the RUM conjecture [2], which states that an access method can only minimize two among read times (R), update cost (U), and memory or storage overhead (M)

at the same time—when optimizing the read-update-memory overheads, optimizing in any two areas negatively impacts the third. The RUM conjecture suggests that optimizing the read performance of an LSM-tree-based KVStores should be in coordination with the caching system of the KVStores. (**P5**) **Inefficiency and inflexibility of the in-kernel filesystem**. Though stacking a KVStore on an in-kernel filesystem (such as Ext4) is a convenient way to implement a storage engine for many applications, an in-kernel filesystem imposes performance overheads. Prior research [32] identified that, for fast devices, kernel involvement is harmful. For example, switching into the kernel via a system call incurs a latency of 5-10 microseconds. Additionally, an in-kernel filesystem introduces unnecessary complexity for applications that do not need a full-fledged filesystem [3]. We also lose the opportunities to further optimize the performance of the I/O stack with an in-kernel filesystem.

To address the above problems in our production environment, we designed LavaStore, a highly optimized storage engine for the pageservers of BxxxXDB. Firstly, to effectively ingest data of high sequentiality(P1), we design a customized engine that follows the semantics of a shared log [8]. The resulting engine has low insertion overhead and a write amplification factor (WAF) close to 1. Secondly, to mitigate the performance issue introduced by intrinsic features of an LSM-tree (P2 and P3), we present a new KV separation technique, which overcomes the issues regarding GC in prior KV separation implementations. Thirdly, we improve the point lookup performance (P4) to achieve $O(1)$ I/O for point lookups in conjunction with KV separation. We utilize a compact, ordered data structure to index the values stored separately from their keys. KV separation requires an effective and efficient mechanism to reclaim the disk spaces of the values that have been deleted. LavaStore implements multiple GC strategies to reclaim disk space in time to reduce the peak storage requirement of a pageserver. Lastly, LavaStore implements a specialized, user-space filesystem called LavaFS for upper-level workloads (P5) to replace full-fledged, in-kernel filesystems (Ext4 or XFS). Since LavaFS lifts a large portion of file access code to the user space, it opens up opportunities for KVStore/filesystem co-designs, which differs from the prior work that only optimizes the KVStore implementation alone.

In summary, this work makes the following contributions:

- We point out that the generic storage solution, which stacks an LSM-tree-based KVStore on an in-kernel filesystem, does not meet the production requirement of the PageStore in BxxxXDB, which is an AuroaDB-like cloud database. On the contrary, we adopt the principle of specialization to implement an efficient storage engine named LavaStore for pageservers in BxxxXDB.
- We highly optimize TxxxxxDB to cater to the characteristics of data stored in pageservers. The optimized TxxxxxDB overcomes the performance issues of prior KV separation implementations, increases point-lookup performance with a memory-efficient data structure along with techniques that improve caching effectiveness, and implements efficient GC mechanisms to reduce peak storage consumption.
- We implement LogEngine, a specialized engine to ingest log data fast while ensuring high persistence guarantees.
- We implement a specialized filesystem, named LavaFS, for TxxxxxDB and LogEngine to fully control the I/O stack in pageserver to exploit the opportunities of more performance optimizations.
- We provide comprehensive evaluation results to show the performance benefits of the techniques in TxxxxxDB, LogEngine and LavaFS. We also demonstrate that LavaStore as a whole is a superior storage engine to the existing solution that utilizes TxxxxxDB on Ext4.

The rest of the paper is organized as follows. Sec. 2 presents the background and motivation of this work. Sec. 3 describes design principles of LavaStore and the high-level description of its key components. Sec. 4 focuses on the techniques that optimize TxxxxxDB. Sec. 5 discusses the techniques of LogEngine. Sec. 6 shows the detailed techniques in LavaFS. In Sec. 7, we provide some empirical measurements of LavaStore. Sec. 8 presents a review of the related work. Finally, Sec. 9 concludes our work.

2 Background and Motivation

In this section, we first present the background of BxxxXDB and its pageservers, and the initial design of the storage backend for the pageservers. Subsequently, we provide an overview of LSM trees and introduce CompanyX's internal LSM-tree-based KVStore, TxxxxxDB, which served as the original storage engine for BxxxXDB's pageservers. Lastly, we highlight the issues we encountered in our production environment to motivate this work.

2.1 BxxxXDB and PageServers

Fig. 1 depicts the overall architecture of BxxxXDB. Inspired by the "log as database" philosophy of Amazon Aurora [49], a BxxxXDB cluster includes a master node that handles both reads and writes, and several read-only replica nodes. The computational layer encompasses a client proxy and SQL engine. This proxy is privy to the system configuration and directs queries between the master and read-only replicas.

BxxxXDB's storage layer consists of a Log Store and a PageStore. The Log Store utilizes append-only distributed blob storage to provide fast redo log persistence with large capacity. The PageStore provides the capability of applying logs to construct data pages and supports random read at page granularity. The PageStore includes a group of storage servers, also known as **pageservers**, and a quorum protocol [46] is used to guarantee data consistency across these pageservers. Each PageStore implements a Replication Framework, responsible for replicating and distributing redo logs. Each redo log

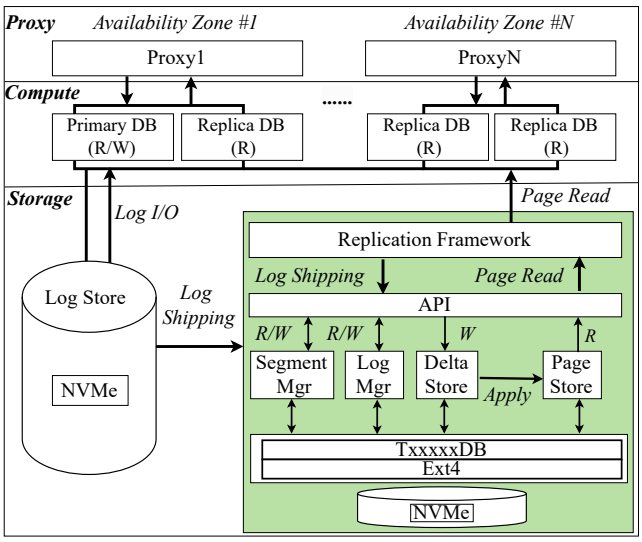


Fig. 1. BxxxXDB architecture. The green box shows the internals of a pageserver that uses the generic storage backend with TxxxxxDB on Ext4.

is assigned a unique LSN (Logical Sequence Number) and redo logs in the same transaction are replicated as a whole batch to ensure atomicity. To ensure high availability, the pageservers need to persist the redo logs shipped from the Log Store. Concurrently, the Log Store would trim its redo logs if they have been persisted in a quorum of pageservers. A gossip protocol [10] is implemented among the pageservers, for one server to retrieve missing redo logs from its peers. In a pageserver, redo logs are sorted as a sequence without any holes before they are applied to the data page index.

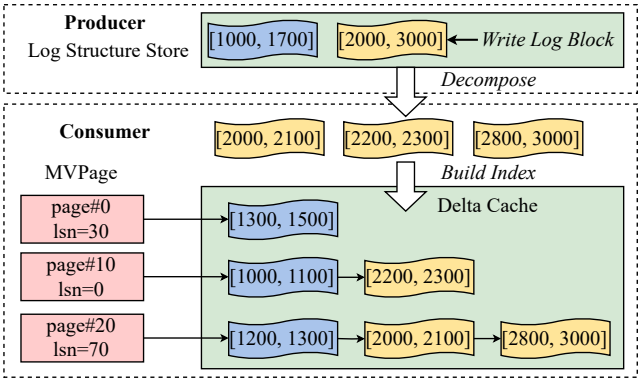


Fig. 2. Redo log processing in pageservers.

The green box in Fig. 1 illustrates a pageserver's internal components: a Segment Manager, a Log Manager, a Delta Store, and a MVPage Store. Externally, a pageserver provides two interfaces: `WriteLog` and `ReadPage`. A pageserver receives segments of redo logs from the Log Store, and keeps information of segments in the Segment Manager. Fig. 2 illustrates how a pageserver processes redo logs received from the Log Store. First, a PageStore decomposes the redo logs in the segments and persists them in the Log Manager. Then, each redo log, keyed with an LSN, is further transformed into a delta log, keyed with a page_id. A PageStore keeps delta logs in the Delta Store and the MVPages, which are multiple versions for each data page, in the MVPage Store. Both Delta Store and MVpage Store are keyed with page_id. Multiple delta logs of the same page are applied to an MVPage to generate another MVpage whose content is more recent.

Initially, pageservers used the same storage engine for all four modules to expedite development. Specifically, a pageserver used TxxxxxDB on Ext4, and stores each module's data in a separated column family, as shown in Fig. 1. As mentioned previously, TxxxxxDB is a fork of RocksDB. We chose TxxxxxDB as the backend of pageservers because LSM-tree-based KVStores have been proven successful in many applications [19]. We present more background information about LSM-trees and TxxxxxDB in the next subsection.

2.2 LSM Tree and TxxxxxDB

Over recent years, LSM-trees have become popular in database engines because of their superior write performance compared to traditional B-tree engines. They efficiently handle writes by buffering them in a MemTable. Once full, the LSM-tree *flush*es this MemTable to disk, producing an on-disk component often implemented as the Sorted-String-Table (SSTable or SST).

In LSM-trees, SSTables can have overlapping key ranges. As a result, a single query might search through multiple SSTables for a value, or return `NotFound` if the key is absent in all searched SSTables, which leads to greater read amplification than in B-trees. Additionally, outdated KV pairs in LSM-trees stay in older SSTables, wasting storage and increasing space amplification. To combat these inefficiencies, LSM-trees perform *compaction*s to reorganize KV pairs across SSTables, optimizing search and reclaiming space.

Many KV databases, including AsterixDB [6], Bigtable [21], Cassandra [7], LevelDB [22], and RocksDB [36], utilize LSM-trees. Of these, RocksDB stands out due to its performance, scalability, and array of configurations, making it ideal for high-demand workloads. At CompanyX, we have adapted and optimized a RocksDB-variant, named TxxxxxDB [1], which was branched from RocksDB `5.18.3`. TxxxxxDB not only enhances performance but also allows for highly compressed storage, achieved by employing advanced algorithms and data structures tailored to the needs of CompanyX's internal services. Multiple internal products of CompanyX, from in-memory databases to distributed KVStores, depend on TxxxxxDB. Currently, numerous instances of TxxxxxDB on Ext4, power CompanyX's popular online offerings like Dxxxxx, Lxxx, Txxxxx, and Txxxxxx.

2.3 Issues of the Generic Solution

While LSM-tree-based KVStores are popular across industries, we at CompanyX experience challenges using

TxxxxxDB on Ext4 for our pageserver's storage backend. We detail these challenges below.

High Write Amplification Compaction, while enhancing query performance by reducing the number of SSTables to search for queries, introduces I/O costs. It necessitates reading from several SSTables and generating a new one, creating significant strain on storage devices. In our use of TxxxxxDB for BxxxXDB's pageservers, we observed a WAF of 6.6 solely at the KVStore layer. Given that write amplification is multiplicative, the total amplification of the database can be much higher. A standard method to lower write amplification of an LSM-tree is separating keys and values [14]. However, implementing KV separation at production quality in a KVStore is challenging. LavaStore offers an efficient KV separation method for LSM-trees, detailed in Sec. 4.1.

Inefficient Space Reclamation KV separation also necessitates GC to reclaim the space occupied by outdated values in the KVStore. Striking a balanced garbage-collection policy for a KV-separated LSM-tree in production is tricky. Overaggressive GC can increase read/write amplification, negating KV separation benefits. Conversely, a gentle approach risks exhausting disk space under heavy workloads, necessitating manually expanding the clients' disk capacity. We introduce a set of GC strategies in Sec. 4.2, aiming to adaptively balance write/read amplification with space efficiency and increase the effectiveness of each run of GC by choosing blob files with a high garbage ratio.

Read Latency Spikes While KV separation reduces data movement during compactions, it hurts read performance. At CompanyX, we experienced that the read latency of pageservers often fluctuates as wide as tens or even hundreds of milliseconds, and the maximum value is close to 1s, while the average read latency is around 300 microseconds. The fluctuation seriously affects the user experience. LavaStore highly optimizes TxxxxxDB's read performance in the context of KV separation, detailed in Sec. 4.3.

High Sync Write Latency To avoid data loss for some types of data (e.g., redo log in BxxxXDB), storage products utilize sync write of a KVStore to ensure the durability of data. A sync write guarantees that once the write operation completes, the data has been persisted to disk and can survive crashes. In our BxxxXDB deployment, we find that the p99.99 latency of persisting a redo log to TxxxxxDB can be as high as tens of milliseconds. Our profiling indicates that these latencies stem from both the KVStore implementation and the filesystem's `fsync` implementation. To address these concerns, LavaStore introduces a specialized log data storage engine (outlined in Sec. 5) that offers robust durability assurances. Additionally, we have developed a userspace filesystem designed to minimize sync write latency, as detailed in Sec. 6.

Inefficient and Inflexible Filesystem Layer Stacking an LSM-based KVStore on an in-kernel system introduces unnecessary overheads, such as the latency of transitions between the kernel and user modes, metadata lookups for read, and metadata updates for sync writes. However, as an LSM-tree KVStore typically has predictable and specific file access patterns, it does not necessarily need all the features of a full-fledged, generic filesystem. For instance, TxxxxxDB or RocksDB's SSTables ranges from 32 MB to 256 MB in size and are stored within a straightforward flat namespace, requiring minimal directory support. Moreover, SSTables are written once sequentially, and read multiple times, either sequentially or at random. The predictable access patterns suggest that one can achieve benefits by tailoring a filesystem specifically for the LSM-tree KVStores. LavaStore introduces a userspace filesystem called LavaFS for LSM-tree KVStores. LavaFS removes unnecessary filesystem features to increase read and write performance, and enables coordination opportunities with the KVStore implementation to further enhance performance. See Sec. 6 for the details.

3 LavaStore's Design

We have discussed that we encountered severe issues when using the generic storage solution, which utilizes an LSM-tree KVStore layered on top of an in-kernel filesystem, for the pageservers of CompanyX. In response, we have developed LavaStore, which employs specialized layers to enhance the efficiency of the I/O stack in pageservers. As depicted in Fig. 3, LavaStore consists of three core components: **(1)** TxxxxxDB, a finely-tuned KVStore; **(2)** LogEngine, a dedicated engine for storing redo logs; and **(3)** LavaFS, an append-only filesystem operating in userspace. Our design for LavaStore follows two fundamental principles, which we elaborate on in the subsequent sections.

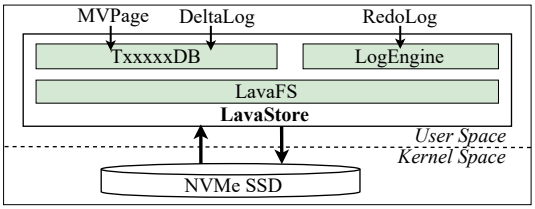


Fig. 3. LavaStore architecture.

3.1 Specialized Engines for Different Data

Firstly, we introduce a highly refined TxxxxxDB that has undergone years of development to handle both MVPages and delta pages efficiently. Given that MVPages and delta pages constitute the majority of data in a database, optimizing resource usage in the backend storage system is a primary concern. TxxxxxDB achieves resource savings through its well-crafted KV separation technique. Additionally, within the context of BxxxXDB, where low read latency is crucial, TxxxxxDB implements a memory-efficient index for blobs to accelerate reads. Furthermore, TxxxxxDB enhances its caching mechanism by taking a holistic approach to cache data and index blocks.

Next, we introduce LogEngine to address the inefficiencies of storing redo logs in TxxxxxDB. LogEngine's primary goal is to improve log shipping efficiency. Persisting shipped redo logs from LogStore is essential to prevent data loss when multiple servers are powered down. Faster shipment of redo logs to pageservers ensures that standby database nodes (BxxxXDB's read-only nodes) access the most recent data sooner, enhancing data freshness. To reduce the latency, LogEngine directly appends redo logs to the underlying filesystem, eliminating TxxxxxDB's sync write overheads. When transferring to a pageserver, redo logs are grouped into large segments, each spanning several GB. There are scenarios when a pageserver might accumulate hundreds of such segments rapidly. LogEngine's design consolidates numerous redo log writes into a single action, minimizing write amplification and reducing metadata update costs.

3.2 A Specialized Filesystem

The second aspect of LavaStore's design is a specialized filesystem, LavaFS, for both TxxxxxDB and LogEngine. We find that optimizing only the KV layer does not effectively address all the performance issues in our production environment. Compared to Ext4, LavaFS improves overall performance with the following three major design considerations.

Userspace Filesystem LavaFS is a userspace filesystem, which only implements the needed features and APIs for LogEngine and TxxxxxDB. The motivation behind creating a userspace filesystem stems from the rapid evolution of storage hardware. Traditional in-kernel filesystems often lag in supporting this new hardware at production quality due to prolonged development cycles. In contrast, developing a mature userspace filesystem takes less time thanks to a variety of libraries, testing, and debugging tools. Moreover, a userspace filesystem is easier to maintain.

Lightweight Metadata As a simplified filesystem, LavaFS has lightweight metadata, with each in-memory `inode` only consumming 10s of bytes, much less than Ext4's 256 B. Thus, the metadata can reside fully in memory. Specifically, LavaFS stores the entire up-to-date metadata in memory. This design eliminates disk accesses for metadata during the read and write path, ultimately enhancing overall system efficiency.

Lightweight Journaling Journaling is a must to ensure crash consistency in a filesystem. We find that journaling in Ext4 introduces unpredictable latencies and high write amplification for `fsync`-intensive workloads (e.g., persist redo log) due to `inode` grouping or additional bookkeeping metadata added by JDB2. LavaFS implements a lightweight journal using a physical log, which consists of multiple extents. LavaFS commits metadata changes to the by flushing the in-memory journal to the physical journal. In our measurements, LavaFS has much lower tail latencies (Sec. 6.2).

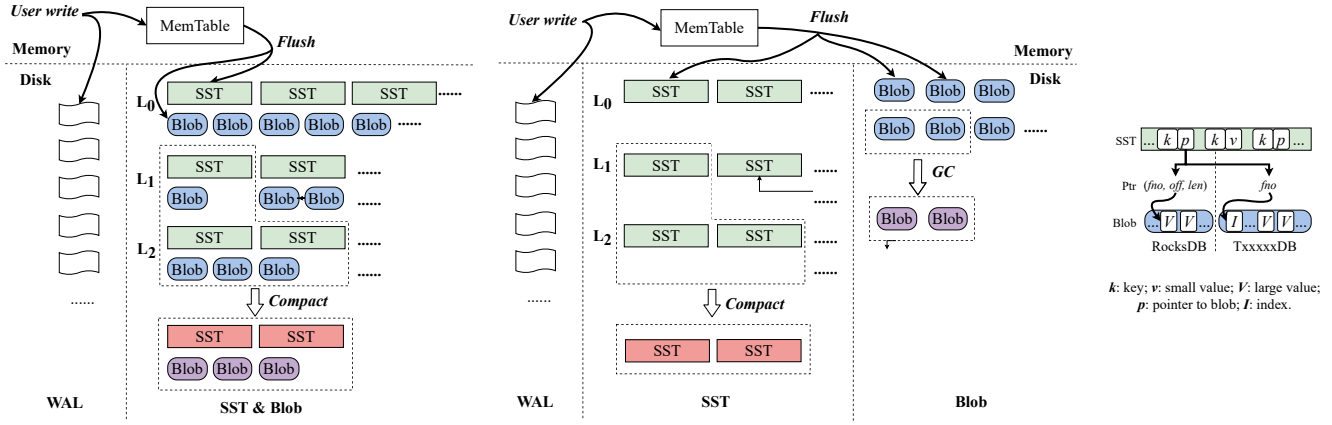
4 Optimizations in TxxxxxDB

In 2019, we introduced the stable version of KV separation in TxxxxxDB, replacing RocksDB's `BlobDB` entirely. This transition was necessary because RocksDB's version did not meet our performance requirements and lacked essential features like column families and merge operators as needed by our customers. While TxxxxxDB's KV separation effectively addressed the issue of high write amplification, its design necessitated a separate GC process that was not fully optimized, leading to suboptimal space utilization. Additionally, KV separation introduced some read overhead, while our customers still expected $O(1)$ disk access for reading values. Since then, we have dedicated significant efforts to optimizations aimed at mitigating both space and read amplification. This section is structured as follows: In Sec. 4.1, we delve into KV separation and its implementation in both RocksDB and TxxxxxDB. Subsequently, we explore optimizations aimed at mitigating space amplification and read amplification associated with KV separation in TxxxxxDB in Sec. 4.2 and Sec. 4.3, respectively. Further optimizations incorporated into TxxxxxDB can be found in Appx. A.

4.1 KV Separation in RocksDB and TxxxxxDB

The fundamental concept of KV separation revolves around retaining small values within SSTables, while storing large values in blob files, with pointers to blob files stored in SSTables (Fig. 4c). This ensures that only the smaller SSTables are involved in compaction processes, ultimately reducing write amplification. However, managing and reclaiming obsolete large values still requires a GC process. Fig. 4 depicts the difference in KV separation strategies and GC approaches between RocksDB and TxxxxxDB, each with unique trade-offs.

Both RocksDB and TxxxxxDB employ similar methods for flushing MemTables, which involves storing small values and pointers to blob files within SSTables, while storing large values in blob files. In RocksDB, the primary encoding for the blob file's pointer ($\approx$15 B) includes information such as the file number (`fno`), offset (`off`), and length (`len`) of the value within the blob file [37]. Therefore, obtaining the pointer value enables direct reading of the value from the blob file with just one file seek. On the other hand, TxxxxxDB's blob file follows the identical format as SSTable, encompassing both index and KVs. The pointers stored in SSTables are just file numbers (`fno`, 8 B) referencing the blobs. To retrieve a blob value, the file number locates the blob file, followed by a binary search of multiple seeks over the blob index to pinpoint the exact value location. Fig. 4c illustrates the distinctions in SSTable and blob format between RocksDB and TxxxxxDB. A value in blob becomes obsolete (garbage) when it is overridden by a new version of the same key or a deletion entry during a compaction. To delete this garbage value, RocksDB integrates GC into normal compaction processes to rebuild SSTables and blobs in one shot, adopting an



(a) RocksDB (GC included in compaction) (b) TxxxxxDB (GC independent of compaction) (c) SST/blob format

Fig. 4. Comparison of key-value separation between RocksDB and TxxxxxDB.

aggressive approach that aims at reducing space but results in higher write amplification. In contrast, TxxxxxDB's GC is gradual, with compaction calculating and logging garbage ratios, and a separate GC worker rebuilding blob files without impacting frontend writes. Rebuilding a blob file can either retain or alter its file number. In scenarios where multiple blob files are merged to create a single blob file, their original file numbers are organized hierarchically within the manifest. This arrangement enables access to the new blob file using the previous file numbers. This design provides flexibility in GC strategies and demonstrates efficiency in write and space management in our production environment.

In summary, RocksDB prioritizes space reduction in its GC strategy, while TxxxxxDB accepts some space overhead when free space is abundant for lower write amplification. Our `db_bench` tests (**A** in Sec. 7.2.1) indicate that KV-separated RocksDB and TxxxxxDB use 32% and 103% more disk space than their non-KV-separated counterparts, respectively. Regarding compaction time, the base RocksDB and TxxxxxDB both take 93 minutes, whereas KV-separated RocksDB takes 26 minutes, and TxxxxxDB only 2 minutes. These results affirm that KV separation reduces write amplification, leading to shorter compaction times but increases space amplification for both systems. After compaction, RocksDB reclaims more space via integrated GC, while TxxxxxDB avoids GC during compaction, resulting in higher space amplification-without impacting foreground tasks. The initial KV separation approach, though effective at first, revealed limitations with our business growth. As workloads intensified and customer expectations for read performance increased, we prioritized optimizing LavaStore to overcome these challenges.

4.2 Optimizing GC for KV Separation

To enhance disk space reclamation efficiency, we have implemented two key techniques. Firstly, we calculate a blob file's garbage ratio based on data size, enabling us to identify files that can efficiently free up space. Secondly, we have increased concurrent background GC jobs by adding threads, with an adaptive strategy that adjusts threads based on disk usage. During low usage, we prioritize space conservation, postponing GC for files with low garbage ratios, while during high usage, we employ more threads for aggressive GC to meet SLA requirements. GC threads have lower priority than compaction threads and can be allocated for compactions as needed. Fig. 5 illustrates the relationship between GC thread count and the garbage ratio as disk usage grows.

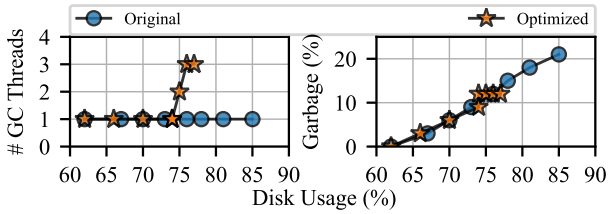


Fig. 5. Number of GC threads and overall garbage ratio versus disk usage (**WriteOnly** in Sec. 7.3).

These GC optimizations yielded substantial benefits in a synthesized, write-only workload. On average, we observed a 7% increase in the garbage ratio within blob files, resulting in a 30% reduction in the number of GC jobs performed during the same time frame. Furthermore, the peak disk usage saw a decrease of 8%, dropping from 85%. We set the 85% level as a threshold to guarantee the presence of enough temporary space for compactions and GC operations, as exceeding this threshold would lead to the disk becoming read-only, causing a halt in incoming write operations. This newfound efficiency allows us to contemplate raising the threshold, increasing our data storage capacity with the same hardware infrastructure. This enhances system performance and contributes to cost savings by optimizing hardware utilization.

4.3 Optimizing Read for KV Separation

To enhance read performance for TxxxxxDB with KV separation, we integrated a memory-efficient Crit-Bit tree index [28]

to blob files. Using just 2.3 bytes per key, the compact blob index is cache-friendly, ensuring rapid in-memory lookups. It requires small disk space in a blob file, raising disk usage by only 1.6‰, and can be loaded into memory swiftly.

To optimize block cache utilization, we introduced a novel cache system called the unified cache. This system segregates index blocks into a dedicated segment, allowing dynamic size allocation between the index block and data block caches. To ensure each `Get` involves at most one disk I/O, we prioritize caching index blocks with higher importance. This is primarily achieved by adopting a more compact index within blob files, as the original SSTable index in blob files is unused. With the new blob index and unified cache, we achieved a significant reduction of approximately 20% to 30% in both average and tail Get latency figures, along with a corresponding decrease in the average disk read size (as shown in Fig. 6).

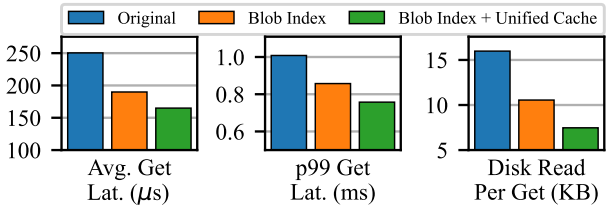


Fig. 6. Comparing `Get` performance using the new blob index and unified cache (**C** in Sec. 7.2.1) with TxxxxxDB.

5 LogEngine: A Customized Log Store

As discussed in Sec. 2.1, a pageserver needs to quickly persist redo logs from received segments. In peak scenarios, a pageserver can accumulate hundreds of segments in a short time. Performance in persisting these redo logs is vital. Given their specific access pattern—primarily sequential writes using Log Sequence Numbers (LSNs) and occasional sequential reads for recovery—a general KVStore like TxxxxxDB proves inefficient. To address this, we use a dedicated log store, LogEngine, to store redo logs on a pageserver. We delve into its design and techniques below.

5.1 LogEngine Overview

Functionalities LogEngine provides a pageserver with a set of APIs to store redo logs. A client of LogEngine can `Create`, `Open`, `Delete` a log file. Within LogEngine, these user-generated log files are termed **Logical Logs**. Once an existing logical log is opened, a user can append new records to it, specifying an LSN and associated data. Users can also `Trim` a logical log based on a given LSN to remove outdated data—like when a database clears its redo log post-recovery, after applying the changes in the redo logs into its index. Additionally, LogEngine offers `Read`, `Seek`, and `Next` APIs, allowing users to fetch a log record by its LSN or scan the log file from a specified LSN using an iterator.

Design Constraints and Goals For LogEngine to meet production requirements, several core problems need to be addressed. Firstly, LogEngine needs to handle redo logs that are in non-sequential order. For example, pageserver A may ask a peer server B for a missing log entry whose LSN is smaller than the maximum LSN in A. The need to accommodate occasional non-sequential entries distinguishes LogEngine from other log stores, such as LogDB [34]. Second, LogEngine is designed to be resource-efficient (e.g., I/O and memory), ensuring it does not hog resources shared by other system services. Thirdly, LogEngine aims to have good performance for the common cases. In LogEngine, append operations are the most frequent, followed by trim, then delete and read. LogEngine prioritizes the performance of frequent operations. LogEngine aims to recover faster, compared to the TxxxxxDB-based solution, since the bootstrapping of LogEngine is a part of the recovery of a pageserver. Lastly, LogEngine should be space-efficient. The trimmed or deleted redo logs' space should be reclaimed in time.

Specifically, LogEngine tries to achieve three primary objectives: **(1)** reduce the latency of persisting redo logs in a pageserver, ensuring adequate read and recovery performance; **(2)** minimize write amplification to save disk bandwidth; and **(3)** ensure that the in-memory data structures consume less than 1% of the total log data size.

System Internals Functionally, LogEngine logs the operations from users on shared logs [8]. LogEngine categorizes user operations into data operations (like `Append`) and metadata operations (like `Create`, `Delete`, and `Trim`). LogEngine comprises four main modules: data logging, metadata logging, logical file metadata management, and the persistence layer.

The data logging module serves `Append` operations from users, and the metadata logging module serves `Create` / `Delete` / `Trim` operations. Internally, both modules buffers logs for data or metadata operations from multiple users in a queue, and a background task is responsible for coalescing these logs and persisting the coalesced log to a disk.

The metadata operations of logical logs are stored separately in the metalog, while the data operations are stored in the datalog. Specifically, both data and metadata logging modules have a directory as the physical storage of the log records. LogEngine manages the filesystem files through the persistence layer. We call each file a **Physical Log**. A physical log stores a sequence of log records for users. Each physical log has a configurable capacity limit (e.g., 256 MB) in LogEngine. When the size of a physical log reaches the capacity limit, a new physical log is generated for storing new log records. We call the physical log for metadata operations **Meta Log**, and data operations **Data Log**.

LogEngine maintains two pieces of in-memory data, namespace and logical log metadata. Namespace maps logical log file names to internal file IDs. The logical file metadata maintains the mappings from each LSN to the physical location of the log record associated with this LSN. To save memory,

LogEngine does not keep all the mappings for a logical log in memory. LogEngine uses physical logs to store a logical file's metadata spilled from memory. When a logical file accumulates a number of in-memory mappings, LogEngine triggers a **MetaFlush** operation to pack all existing in-memory mappings and store them in a physical log maintained by the metadata logging module. We elaborate on logical file metadata management details in Sec. 5.2.

5.2 Logical File Metadata

Two-level Indexing LogEngine maintains the mappings from LSNs to physical log locations for each logical log in a two-level indexing. We call a logical file's metadata **TLI-Meta**. Fig. 7 illustrates the details of metadata management of a logical log file. A logical log's in-memory metadata has two parts: direct mapping (L0) and indirect mapping (L1). L0 maps LSNs to physical locations of log data; L1 maps an end LSN to the physical location of MetaFlush records, each of which encodes a group of mappings of $\langle lsn, log\_data\_location \rangle$.

Serving User Requests We explain that LogEngine serves user requests as follows. For a `Put` request, it first appends the log data to a physical log and stores the LSN and the physical location in L0 after the log record is persisted. As the number of entries in L0 surpasses a threshold, all existing L0 entries are packed into multiple MetaFlush records (256 per record) and persisted in the metadata log. The largest LSN in a MetaFlush record, and the on-disk location of the MetaFlush records are inserted into L1. An L1 entry also keeps the set of all LSNs of the MetaFlush record. The LSN list introduces 8-B memory cost for each log record. If the average size of each append is 1,000 B, the memory cost is around 0.8% of the total log size. This two-level index design caches the most recent mappings in memory and facilitates future reads.

For a `Trim` request, after persisting the trim operation in the meta log, LogEngine removes all the entries in the trimmed LSN range in L0. For L1 entries, LogEngine removes a whole entry if all LSNs in this entry fall into the trimmed LSN range; otherwise, LogEngine removes the trimmed LSNs in the LSN set kept in the L1 entry.

For a `Read`, LogEngine's implementation is similar to the point query in a regular LSM-tree. It first searches L0 for the requested LSN and then L1. To scan a logical log, LogEngine creates an iterator for this logical log. Internally, LogEngine creates two iterators for both L0 and L1 and merging the results of these two iterators.

Checkpoint and Metadata Compaction LogEngine utilizes a background task to checkpoint L1 and namespace to a checkpoint file to accelerate recovery. The checkpoint process follows the steps of a fuzzy checkpoint [44], to reduce obstruction for user requests. In the checkpoint process, LogEngine merges the MetaFlush records in the old checkpoint file (base) with those in the physical log (incremental) for metadata operations and generate a new checkpoint file. The compaction process also serves the purpose of GC obsolete MetaFlush records, similar to compaction in an LSM-tree.

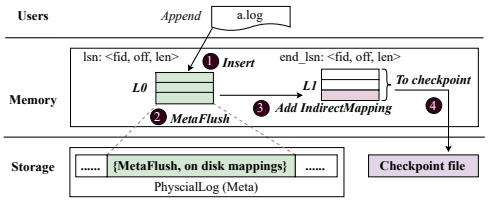


Fig. 7. LogEngine metadata management.

5.3 GC, Recovery, and Performance

Garbage Collection To reclaim space in data logs, LogEngine relocates the valid records of a logical log to the most recent physical log. Specifically, when the amount of garbage data in the data log exceeds a threshold, the oldest data log is selected for GC. To avoid scanning a whole data log, LogEngine adds a reverse index table containing all logical log ids and the LSNs of the log records in the data log at the end of the data log. This way, during GC, only the reverse index table needs to be read. The GC process is as follows: (1) Read the reverse index table at the end of the data log; (2) Iterate all the entries in the reverse index table and for each entry check if the metadata of the logical log, indicated by logical log id in the entry, includes the LSN encoded in the entry. If the LSN is not included, the record is invalid and skipped; (3) Write the valid append records into the current data log as an incoming append from a client. (4) The data log is deleted once all valid data has been migrated. In practice, GC introduces very few I/Os besides reading the reverse index table because logical logs are quickly trimmed or deleted.

Recovery During startup, LogEngine first loads the most recent checkpoint to recover the namespace and L1 for each logical log. Then, LogEngine scans and replays the meta log to bring the L1 of each logical log to the state before the last shutdown or crash. At last, it scans the data log to rebuild the L0 entries for each logical file. We present more details in Appx. B.

Performance Fig. 8 compares the performance of durable writes in LogEngine and TxxxxxDB. LogEngine has much lower average and tail latencies than TxxxxxDB, for writes that require to immediate durability.

6 LavaFS: A Customized Filesystem

We find that only optimizing the KV layer does not address all the performance issues we encountered in production. To avoid the performance overheads introduced in an in-kernel filesystem, we build a userspace filesystem named LavaFS. We elaborate on the details of LavaFS below.

6.1 LavaFS Overview

LavaFS Layout As shown in Fig. 9, LavaFS boasts a simple filesystem layout, encompassing three distinct types of filesystem data: the superblock, journal, and data. The superblock

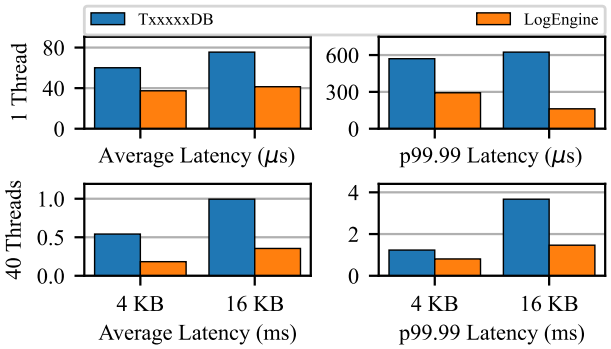


Fig. 8. Latencies of writes with immediate persistence for TxxxxxDB and LogEngine (**WriteOnly** in Sec. 7.3).

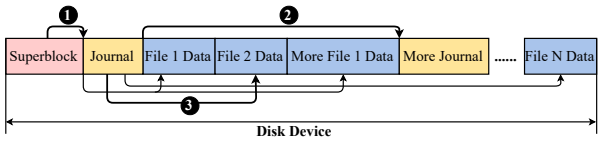


Fig. 9. LavaFS layout. (1) stores Journal's start location; (2) the last transaction may connect to additional Journal's location if needed; and (3) Journal contains `inode` update transactions that can be replayed to rebuild `inodes`, which include file extents' disk locations.

contains global filesystem metadata, including journal `inode` content, UUID, version, block size, and more. It resides at the second 4 KB of the device. The journal preserves the most recent checkpoint of the filesystem operation log, which can be replayed during filesystem mount to fully recover in-memory file metadata. The data section stores the content of individual files, with each data block representing an extent—a contiguous LBA range.

Writes and Journaling For writes, a LavaFS client uses `append` to add new data to the end of an opened file; internally, LavaFS creates a write buffer for each file to aggregate append data. The size of the write buffer is 512 KB. A user can call the `flush` API to flush the data blocks to the device. LavaFS stores the up-to-date metadata of each file fully in memory, and eliminates lookups to the on-disk index for I/Os. A `flush` also updates the in-memory `inode` to reflect the new size of the file and the `inode` is marked dirty. A user calls `fsync` to persist the `inode` changes to a disk to make file data and metadata consistent. LavaFS maintains an in-memory transaction log for all filesystem updates. For example, if a directory is created, a **mkdir** transaction record is appended to this log. When a user calls `fsync` or `sync_metadata`, the in-memory transaction log is persisted to the disk to be a part of LavaFS's journal. When LavaFS is just mounted, the recovery process replays the transaction records in the journal to rebuild the in-memory `inodes` for all files in LavaFS. Compared with Ext4, LavaFS's journal mechanism is more lightweight. In JBD2, the kernel thread that Ext4 utilizes for journaling, a transaction comprises the transaction header block (4 KB), one or more log blocks ($4n$ KB), and the transaction commit block (4 KB). These metadata blocks in the a JBD2 transaction records introduce high write amplification for systems that issue frequent writes that need immediate persistence.

Asynchronous Journal Compaction LavaFS needs to compact its journal to bound the duration of its recovery and reclaim space. LavaFS implements asynchronous journal compaction with a background thread. To compact the journal asynchronously, LavaFS first makes a copy of the current in-memory data. During the copy, the background thread only needs to briefly hold the LavaFS global lock since all data is in memory. Once the copy is done, LavaFS creates a new journal to accommodate new changes to the in-memory metadata, and releases the lock so that a LavaFS client can change in-memory metadata. Meanwhile, the background thread checkpoints the copied metadata to the disk, and appends a **Jump Record** after the checkpointed data. The Jump record points to the most recent journal data generated by LavaFS clients right after the checkpoint process starts. Finally, the background thread retakes the global lock to reclaim the extents of the old journal and update the log pointer in the superblock. By employing this approach, the journal compaction process does not obstruct newly incoming I/O operations, ensuring smoother system operation. Through asynchronous journal compaction, we've cut the peak latency of synchronous insertions with a KV pair with 36 B key and 16,000 B value to TxxxxxDB from 57 milliseconds to 17 milliseconds in our `db_bench` tests.

KVStore-Filesystem Co-designs To further reduce the frequency of in-memory metadata updates, we conducted cross-layer optimizations between LavaFS and TxxxxxDB. LavaFS implements `fdatasync` APIs for TxxxxxDB to avoid the usage of `fsync` for TxxxxxDB's Write-Ahead-Log (WAL) (see Fig. 4). Unlike `fsync`, which involves two separate I/O operations for data and metadata, respectively, `fdatasync` only persists file data. In LavaFS, if no new extent needs to be allocated to accommodate the data flushed by an `fdatasync`, the metadata of the file is not updated; otherwise, the in-memory metadata of the file is changed and the new metadata is persisted in the journal. Hence, for `fdatasync`, metadata updates are tied to the number of extent allocations, not API invocations, as is the case with `fsync`. As a result, we have effectively reduced the WAF at filesystem layer to nearly 1, considering that LavaFS only has one metadata write for the entire 1-MB extent.

To utilize `fdatasync`, a user needs to be aware of one caveat. Specifically, the file size in the `inode` may not reflect the real size of the file since `fdatasync` does not always update the in-memory metadata. Therefore, a user can not use the file size returned by stat to find the tail of a file. Instead, a user needs to scan the last extent to identify the boundary between valid and invalid data. Therefore, a user has to implement a mechanism to distinguish between valid data and

invalid data. For the WAL of TxxxxxDB, we rely on the checksum in each log entry to validate the content in the last extent of the WAL. During recovery, TxxxxxDB conducts a linear scan of each extent, and it verifies whether the extent is in the correct record format and possesses correct checksum. Once the current extent read fails the verification, all subsequent extents, including the current one, will be discarded.

6.2 LavaFS Performance

We compare the performance of sequential write and random read, access patterns cared by LogEngine and TxxxxxDB, with FIO tests on LavaFS and Ext4. The sequential write tests include two cases. As shown in Fig. 10, the first case uses one thread to create 400 files and write 256 MB (i.e., regular SSTable size) 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 KVStore (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 precreated 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 sync writes; for 4 KB I/O size LavaFS has a WAF of 2 when using `fsync`, while Ext4 has a WAF of 8. Additionally, for multi-threaded non-sync writes, LavaFS also exhibits superior performance compared to Ext4. 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.

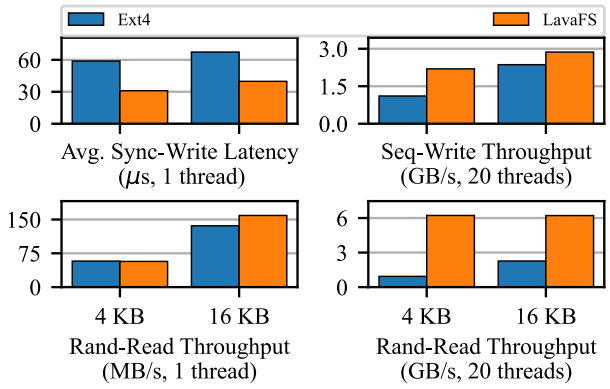


Fig. 10. FIO performance comparison for Ext4 and LavaFS.

Fig. 11 shows the write and read performance of TxxxxxDB on LavaFS and Ext4 with `db_bench`. In both workloads, TxxxxxDB performs better on LavaFS than Ext4 with lower average and tail latencies. Compared with Ext4, LavaFS has in-memory metadata. 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.

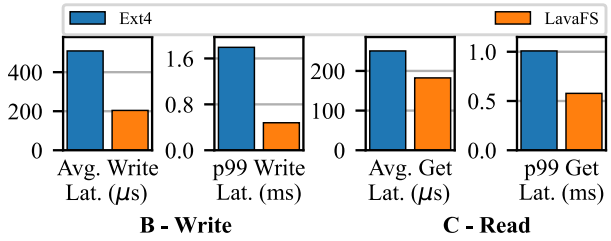


Fig. 11. Write and read performance comparison of TxxxxxDB on different filesystems (**B** and **C** in Sec. 7.2.1).

7 Evaluation

In this section, we evaluate LavaStore and conduct comparisons with the storage engine that utilizes TxxxxxDB on Ext4. Additionally, we perform microbenchmarks to compare LavaStore with upstream RocksDB. Our assessment aims to address the following questions: **(Q1)** How does LavaStore's perform compared to TxxxxxDB and the upstream RocksDB for typical KVStore workloads (Sec. 7.2), **(Q2)** What is the performance of LavaStore when handling log-style data compared to TxxxxxDB (Sec. 7.3), and **(Q3)** To what extent does LavaStore enhance the performance of BxxxXDB in comparison to the storage engine using TxxxxxDB on Ext4 (Sec. 7.3).

7.1 Hardware Configuration

We test RocksDB, TxxxxxDB and LavaStore on a set of machines with the same hardware. We compare LavaStore with TxxxxxDB and RocksDB on a single disk of a single machine with KVStore workloads (`db_bench`), and we compare two storage engines of pageservers, LavaStore and TxxxxxDB on Ext4, with `SysBench` in a cluster-based environment. Each test machine has an Intel Xeon Silver 4314 CPU featuring 64 cores, 377 GB of DRAM, 5 Intel SSDPF2KX038TZ NVMe SSDs (each with a capacity of 3.84 TB). The test machines are running a 64-bit Debian 10 operating system with Linux kernel version 5.15.103. In all `SysBench` tests, we use a BxxxXDB cluster with 3 compute nodes and 3 storage nodes with identical hardware. Each storage node utilizes all five disks on the machine.

7.2 Performance Evaluation using `db_bench`

7.2.1 `db_bench` Configurations

We utilize `db_bench`, which is the standard tool for benchmarking the performance of RocksDB and the variants derived from RocksDB, to evaluate and compare the performance of the latest version `8.5.3` of RocksDB, the original TxxxxxDB, and LavaStore (the optimized TxxxxxDB on

LavaFS). We use the original `benchmark.sh` script [39] from RocksDB version `8.5.3` for all our tests. We focus on evaluating the performance write operations, read (`Get`) operations, short-range and long-range scans of these KVStores. We test each KVStore with the following five workloads (script workload names enclosed in parentheses): **A - Load** (`bulkload`) with 1 thread doing batched writes without syncing and WAL (`db_bench`'s `fillrandom` workload), followed by a manual compaction (`compact` workload), for fast preconditioning; **B - Write** (`overwrite`) with 16 threads doing random `Puts`; **C - Read** (`readwhilewriting`) with 16 threads doing random `Gets`, while 1 thread doing random `Puts`; **D - Short Scan** and **E - Long Scan** (`fwdrangewhilewriting`) with 16 threads doing random iterator `Seeks`, followed by 10 and 1,000 iterator `Nexts`, respectively, while 1 thread doing random `Puts`.

We run the test script with default parameters, such as a key length of 36, a value length of 16,000, and a 50% compression ratio using `ZSTD`. The total raw KV size is set to precisely 1 TB, equivalent to 68,565,205 operations and roughly 500 GB of compressed on-disk data; the KV separation threshold in RocksDB/TxxxxxDB is configured as 512 for `min_blob_size`/`blob_size`; the cache size is configured to maintain a typical 1:500 cache-to-total data size ratio, resulting in a 1 GB block cache size; in RocksDB, the blob cache is enabled and shared with the block cache; in TxxxxxDB, when a unified cache is used, 800 MB is allocated for the index block cache, and the data block cache uses the rest; the `USE_O_DIRECT` flag is set to enable direct I/O for read, flush, and compaction on Ext4, thereby avoiding the usage of the system's page cache, to have a fair comparison with LavaFS, which operates in direct I/O mode, as 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 to restrict the write rate to 60 MB/s and prevent write stalls, with a maximum run-time of 30 minutes; the default filesystem of the KVStores is Ext4 unless explicitly specified to be `LavaFS`. The median values of three runs are reported.

7.2.2 `db_bench` Results

Fig. 12 shows the performance comparison between LavaStore (optimized TxxxxxDB + LavaFS) with upstream RocksDB and the original TxxxxxDB. In the first row, LavaStore's average write latency is 60% lower than the original TxxxxxDB and 53% lower than RocksDB. LavaStore also has much lower tail write latency. In the second row, LavaStore has the highest QPS with the lowest read latencies, outperforming RocksDB by 23% on average and 9% in tail read latency, respectively. This improvement stems from the optimized TxxxxxDB's cache utilization, benefited from the new blob index and unified cache design. Short Scan performs similarly to point queries (the second row), and we omitted it in the figure. In the third row, LavaStore showcases a 31% improvement in average iteration latency and a 39% improvement in tail iteration latency compared to the original TxxxxxDB, although it remains slower than RocksDB due to RocksDB's better suitability for long-range scans. Despite this, LavaStore consistently outperforms in terms of write and read performance, with notable reductions in latency.

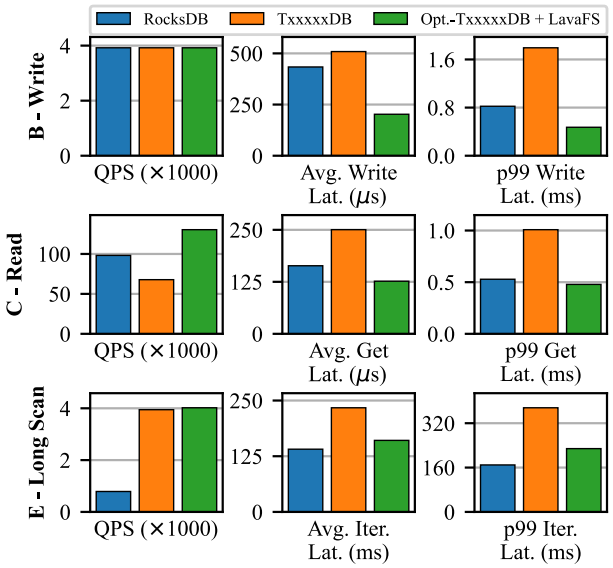


Fig. 12. Write, read and scan performance of RocksDB, TxxxxxDB and LavaStore (optimized TxxxxxDB + LavaFS).

7.3 `SysBench` Results

We also run `SysBench` [26] on a BxxxXDB cluster (Sec. 7.1), where each pageserver uses 5 instances of either TxxxxxDB on Ext4 or LavaStore. The BxxxXDB cluster manages 6 datasets, each containing 250 tables, with 25 million rows per table, and each row consists of 200 B. 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. To demonstrate the performance improvements brought by LogEngine and LavaStore, we use performance results measured on pageservers and BxxxXDB clients with two different storage engines.

Performance of PageServers As shown in Tab. 1, the average latency of `WriteLog` of pageservers is reduced by 61% and 48% with LavaStore as the backend storage engine for WriteOnly and ReadWrite tests, respectively. This demonstrates the effectiveness of LogEngine. Moreover, the average latency of ReadPage API also reduces by 16% for ReadWrite test, which demonstrates the effectiveness of the read performance optimizations introduced in optimized TxxxxxDB.

Performance of BxxxXDB. From the perspective of BxxxXDB, LavaStore increases the number of transactions per second by 6% and 10%, compared with using TxxxxxDB as the backend of pageservers, for WriteOnly and ReadWrite tests, respectively. Similarly, LavaStore increases the number of queries per second by 6% and 10% for WriteOnly

and ReadWrite tests, respectively. LavaStore also reduces the average latency of database operations by 6% and 10% for WriteOnly and ReadWrite tests, respectively. Lastly, the end-to-end WAF of BxxxXDB reduces by 24% and 44% for WriteOnly and ReadWrite tests, respectively. The `SysBench` results demonstrate that LavaStore is an effective storage engine on pageservers for BxxxXDB.

	WriteOnly		**ReadWrite**	
	TxxxxxDB	LavaStore	TxxxxxDB	LavaStore
WriteLog (μs)	3585	1394	1358	695
ReadPage (μs)	N/A	N/A	1189	1001
DB TPS	200461	212546	12328	13577
DB QPS	1202780	1275280	246564	271551
DB LAT (us)	3835	3620	62263	56531
E2E WAF	7.3	5.6	35.94	20.25

Tab. 1. Performance comparison of pageserver and BxxxXDB with `SysBench` workloads.

Metrics from Production Deployments Fig. 13 shows the performance number of a pageserver from our production environment. The `ReadPage` latency is around 370 microseconds. The latency of reading data pages is an enabler for BxxxXDB to answer OLTP queries with sub-millisecond latencies. Moreover, the `WriteLog` latency is around 400 microseconds, allowing BxxxXDB to answer OLAP queries with sub-second freshness.

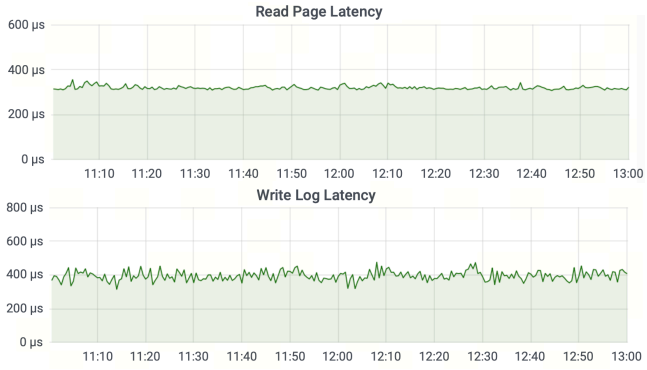


Fig. 13. Latencies of `ReadPage` and `WriteLog` for a PageServer in a production deployment.

8 Related Work

This section summarizes prior work relevant to this work.

Storage Engines for PageServers Though there are multiple research papers discussing the internal of disaggregated databases, such as Alibaba PolarDB [12], Amazon Aurora [49], Huawei Taurus [16], and Microsoft Socrates [5], few papers mentioned the internal design of the pageservers. LavaStore is the first work in the literature to reveal how to design an efficient storage engine for pageservers.

Key-Value Separation KV separation, popularized by WiscKey [35] in 2016 with LevelDB, has seen adoption in various systems. BadgerDB [17], written in Go, closely aligns with WiscKey's approach. RocksDB introduced the stable version of BlobDB in 2021 [47], as discussed in Sec. 4.1. Titan [42], a RocksDB plugin, shares a similar strategy but uses a post-compaction callback for GC, impacting its efficiency. Titan also introduced *LevelMerge*, a method for in-compaction GC limited to the last two levels. iLSM-SSD [29] combines WiscKey with near-data processing and implements a KV separation-based LSM-tree in limited SSD memory. HashKV [13] aims for high update performance on top of KV separation under update-intensive workloads. Although insightful, these KV separation ideas do not address read performance degradation or the introduced GC complexity.

Optimizing Queries for LSM-trees State-of-the-art LSM engines employ various in-memory data structures, including filters, indexes, and caches, to enhance lookup performance. RocksDB [19] and LevelDB [22] use fence pointers to accelerate reads to the sorted runs. Most LSM engines also use Bloom filters [48] to skip SSTables that do not contain the queried key. ElasticBF and Modular Bloom filters handle access skew, while Ribbon filter [18] balances index time and space. Cuckoo filters [20] are alternatives to Bloom filters. Different to these works, LavaStore contributes techniques—a memory-efficient index and caching policies—to improve the read performance of a KV-separated KVStore.

Efficient Log Stores LogDB [34], a component of LogDevice, is layered over RocksDB and manages time-ordered collections. LogDB has a WAF of at least 2 since log data is in both WAL and SSTables. Moreover, LogDB only supports strictly sequential data. Meanwhile, Border-Collie [24] offers a wait-free, read-optimal logging algorithm, determining upper bounds even with sporadic idle threads. Different from Border-Collie, LogEngine optimizes for durable writes and memory efficiency.

Filesystem Optimization for KVStores Recent research [4, 9, 23, 25, 33, 50] has focused on improving filesystems for KVStores and distributed storage. For instance, exF2FS [40] is a transactional, log-structured filesystem enhancing performance in applications like SQLite [27] and RocksDB. Max [31] is tailored for flash storage, improving scalability and performance, especially in multi-threaded RocksDB operations. Both, however, are generic in-kernel systems, while LavaFS is a userspace filesystem specifically for LSM-tree KVStores. Techniques from exF2FS and MAX, like GC and multi-core optimization, can be applied to LavaFS.

9 Conclusion

In conclusion, LavaStore represents a shift from generic storage solutions to a more specialized, integrated approach, for database pageservers. By focusing on the unique requirements of BxxxXDB and ensuring tight integration between its components, LavaStore offers a robust, high-performance storage solution tailored to CompanyX's needs. In future work, we plan to conduct HW-SW co-design [30] to reduce in-device write amplification with ZNS [11] SSDs.

References

[1] CompanyX. TxxxxxDB. https://github.com/companyx/txxxxxdb.

[2] In *The Krzyż Conjecture: Theory and Methods*, pages 289–294. WORLD SCIENTIFIC, April 2021.

[3] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[4] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027, 2020.

[5] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1743–1756, 2019.

[6] Apache. AsterixDB. https://asterixdb.apache.org.

[7] Apache. Cassandra. http://cassandra.apache.org.

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

[9] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed LSM-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–316, 2020.

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

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

[12] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 29–41, 2020.

[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. HashKV: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019, 2018.

[14] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171, 2020.

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

[16] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1463–1478, 2020.

[17] Dgraph. BadgerDB. https://github.com/dgraph-io/badger.

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

[19] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

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

[21] Google. Bigtable. https://cloud.google.com/bigtable.

[22] Google. LevelDB. https://github.com/google/leveldb.

[23] Igjae Kim, J Hyun Kim, Minu Chung, Hyungon Moon, and Sam H Noh. A log-structured merge tree-aware message authentication scheme for persistent key-value stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 363–380, 2022.

[24] Jongbin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 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*, pages 723–740, 2019.

[25] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S Kim, and Sungjin Lee. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 75–92, 2021.

[26] Alexey Kopytov. SysBench manual. *MySQL AB*, pages 2–3, 2012.

[27] Jay Kreibich. *Using SQLite*. O'Reilly Media, Inc., 2010.

[28] Adam Langley. Crit-bit trees. https://www.imperialviolet.org/binary/critbit.pdf.

[29] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. iLSM-SSD: An intelligent LSM-tree based key-value SSD for data analytics. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 384–395. IEEE, 2019.

[30] Huaicheng Li, Martin L Putra, Ronald Shi, Xing Lin, Gregory R Ganger, and Haryadi S Gunawi. lODA: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 263–279, 2021.

[31] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A multicore-accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 877–891, 2021.

[32] Jing Liu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Sudarsun Kannan. File systems as processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[33] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 819–835, 2021.

[34] LogDevice. LogDB. https://logdevice.io/docs/LogsDB.html.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[36] Meta. RocksDB. https://rocksdb.org.

[37] Meta. RocksDB Wiki: BlobDB. https://github.com/facebook/rocksdb/wiki/BlobDB.

[38] Meta. RocksDB Wiki: Data Block Hash Index. https://github.com/facebook/rocksdb/wiki/Data-Block-Hash-Index.

[39] Meta. RocksDB Wiki: Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.

[40] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. exF2FS: Transaction support in log-structured filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 345–362, 2022.

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

[42] PingCAP. Titan. https://github.com/tikv/titan.

[43] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. Closing the B+-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 69–82, 2022.

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

[45] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[46] Dale Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

[47] Levi Tamasi. Integrated BlobDB. https://rocksdb.org/blog/2021/05/26/integrated-blob-db.html.

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

[49] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[50] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H Noh, and Beomseok Nam. On stacking a persistent memory file system on legacy file systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 281–296, 2023.