Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

DS-LSM-tree: SSTable-Decoupled LSM-tree

Tao Cai

caitao@ujs.edu.cn

JiangSu University

Yikang Deng

JiangSu University

Dejiao Niu

JiangSu University

Zeyu Huang

JiangSu University

Qiujing Huang

JiangSu University

Maotao Jiang

JiangSu University

Research Article

Keywords: LSM-tree, ZNS SSD, NVM, Hybrid Storage

Posted Date: April 6th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8587363/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

DS-LSM-tree: SSTable-Decoupled LSM-tree

Tao Cai[1*], Yikang Deng[2], Dejiao Niu[3], Zeyu Huang[4], Qiujing Huang[5], Maotao Jiang[6]

[1*]School of Computer Science and Communication Engineering, JiangSu University, Zhenjiang, 212013, Jiangsu, China.

*Corresponding author(s). E-mail(s): caitao@ujs.edu.cn;
Contributing authors: dengyikang123456@126.com; djniu@ujs.edu.cn;
981087820@qq.com; 15294845059@163.com; 2222508039@stmail.ujs.edu.cn;

Abstract

The Log-Structured Merge Tree (LSM-tree) is widely adopted in key–value (KV) storage systems due to its ability to transform random writes into sequential writes. However, LSM-tree still suffers from issues such as write amplification and periodic performance degradation, which become more severe on emerging storage devices with strict sequential-write constraints, such as Zoned Namespace Solid State Drives (ZNS SSDs). Meanwhile, the introduction of Non-Volatile Memory (NVM) provides new opportunities for optimizing the internal structure of LSM-tree.We observe that traditional SSTables tightly couple metadata with data blocks, causing frequent metadata updates to trigger unnecessary rewrites of large data regions, thereby exacerbating write amplification. To address this problem, we propose DS-LSM-tree, an LSM-tree design based on SSTable decoupling. In DS-LSM-tree, large and infrequently accessed data blocks are sequentially stored on ZNS SSDs, while frequently accessed and fine-grained metadata are migrated to NVM, alleviating the mismatch between update granularity and storage device characteristics.Based on the decoupled SSTable design, we further develop a ZNS-aware SSTable metadata structure and propose an NVM-based logical merge strategy for SSTable data blocks, which reduces unnecessary physical data migrations. We implement a prototype system named SPMS based on DS-LSM-tree and evaluate it using YCSB benchmarks. Experimental results show that, compared with the traditional LSM-tree–based system ListDB, SPMS improves write throughput by up to 51.2%, increases read throughput by 5.1%, and reduces write amplification by 53.1%.

Keywords: LSM-tree, ZNS SSD, NVM, Hybrid Storage

1 Introduction

In recent years, with the rapid growth of large-scale data-intensive applications, key–value (KV) storage systems have been widely adopted in scenarios such as cloud computing, big data processing, and online services. By transforming random writes into sequential writes, the Log-Structured Merge Tree (LSM-tree) achieves high write performance and has become the core data structure of mainstream KV storage systems, including LevelDB and RocksDB [6, 13]. However, LSM-tree relies on a multi-level structure and periodic compaction to maintain data ordering. While this design improves write throughput, it inevitably introduces issues such as write amplification and performance fluctuations, which become more pronounced in environments with emerging storage devices [5].

Zoned Namespace Solid State Drives (ZNS SSDs) partition storage space into multiple zones and enforce sequential writes, which improves sequential write performance while imposing strict constraints on data layout and update patterns in upper-layer storage systems [1]. In recent years, existing studies have optimized the performance of LSM-tree–based systems on ZNS SSDs from perspectives such as system architecture, zone allocation, and compaction scheduling [2]. However, these studies primarily

focus on the placement and migration of SSTables across different levels or zones, and rarely analyze the root causes of write amplification from the perspective of SSTable internal structure.

We further observe that, in traditional LSM-tree designs, SSTables typically store metadata and data blocks in a tightly coupled manner. During system operation, SSTable metadata—such as indexes, filters, and statistical information—are frequently updated, whereas the data blocks are often large in size and expensive to rewrite. On conventional block-based storage devices, this mismatch in update granularity can be partially mitigated through in-place overwrites or device-managed garbage collection. However, in ZNS SSD environments, even small-scale metadata updates may trigger rewrites or data migrations at the zone granularity, thereby significantly amplifying write volume and becoming a critical factor that limits LSM-tree performance and device lifetime.

Non-Volatile Memory (NVM) offers near-DRAM access latency, high read and write performance, and byte-addressability, making it a promising component for high-performance storage systems [16]. However, in hybrid storage environments composed of NVM and ZNS SSDs, traditional LSM-tree designs still treat SSTables as indivisible management units, failing to distinguish different types of data in terms of update frequency and write granularity. As a result, even when only a small amount of metadata within an SSTable needs to be updated, large-scale data rewrites or migrations on ZNS SSDs may still be triggered. This leads to persistent write amplification in hybrid storage systems and may further exacerbate the problem due to increased data placement complexity.

To address the aforementioned issues, we restructure the SSTable organization in traditional LSM-tree designs and propose DS-LSM-tree, an LSM-tree architecture based on SSTable decoupling. The core idea of DS-LSM-tree is to decouple SSTable metadata from data blocks by migrating frequently accessed but fine-grained metadata to NVM, while placing infrequently accessed and large-volume key–value data on ZNS SSDs. This design alleviates the mismatch between update granularity and storage device characteristics.Building on the decoupled SSTable structure, we further design an SSTable logical merge strategy that performs merge operations at the logical level, thereby reducing physical data migrations. As a result, DS-LSM-tree effectively lowers write amplification and improves write efficiency.

The main contributions of this work are summarized as follows.

- To address the tight coupling of key–value data and metadata in traditional SSTables, we decouple SSTables into a data region and a metadata region. By analyzing the different access characteristics of key–value data and metadata, we store the large-capacity SSTable data region on ZNS SSDs, while migrating the SSTable metadata region to NVM. This design improves key–value lookup and access efficiency on ZNS SSDs.
- Based on SSTable decoupling, we design a new SSTable metadata structure tailored to the access characteristics of NVM and ZNS SSDs. The proposed metadata design improves LSM-tree access efficiency while reducing write amplification on ZNS SSDs.
- Leveraging the sequential append constraint of zones in ZNS SSDs, we propose an SSTable logical merge strategy and construct an SSTable logical merge area in NVM. This approach avoids unnecessary write operations on ZNS SSDs caused by SSTable merging, while maintaining low NVM space overhead and extending the service lifetime of ZNS SSDs.
- We implement a prototype system named SPMS based on DS-LSM-tree and evaluate it using YCSB benchmarks. Experimental results show that, compared with the LSM-tree–based system ListDB, SPMS improves write throughput by up to 51.3%, increases read throughput by 5.1%, and reduces write amplification by 53.1%.

2 Related works

2.1 Write Amplification Optimization in LSM-tree

As a data structure widely adopted in key–value storage systems, LSM-tree has long attracted extensive research attention due to its write amplification problem. Existing studies mainly optimize LSM-tree from the perspectives of compaction strategies, hierarchical organization, and caching mechanisms. For example, write amplification can be mitigated by adjusting the size ratios of SSTables across different levels or tuning compaction trigger conditions to reduce unnecessary cascading compactions [18].

In addition, introducing compaction buffers or delayed compaction mechanisms can alleviate the read and write pressure imposed on underlying storage devices during the compaction process [4]. These approaches are effective in reducing write amplification to some extent on traditional SSDs or HDDs.However, their optimization objectives are typically confined to compaction scheduling and hierarchical organization. In addition, some studies attempt to leverage high-speed storage media such as

DRAM or NVM to cache intermediate compaction results, thereby reducing frequent accesses to slower external storage devices [17]. These approaches can effectively reduce compaction latency and improve system throughput. However, they often require additional DRAM or NVM capacity and still manage SSTables as indivisible storage units. Overall, existing write amplification optimization methods for LSM-tree mainly focus on the compaction process itself and rarely investigate the root causes of write amplification from the perspective of SSTable internal structure.

2.2 Key Value Separation and SSTable Structure Optimization

To reduce the overhead caused by large-scale data migration during LSM-tree compaction, key value separation has emerged as an important research direction in recent years. Representative studies separate keys and their index information from value data, allowing compaction operations to primarily operate on much smaller key indexes. This design significantly reduces the number of value data rewrites during compaction [11]. Such approaches have achieved effective write amplification reduction and are particularly suitable for workloads with large value sizes.

However, existing key value separation approaches primarily focus on reducing the migration cost of value data. Their optimization targets are usually defined at the logical level of key value separation rather than the physical organization within SSTables. In these systems, SSTables are still treated as indivisible units for storage and management, and their internal metadata such as indexes and filters remain tightly coupled with data blocks. As a result, updates to SSTable metadata may still trigger large-scale writes to underlying storage devices. In contrast, our work focuses on the mismatch between update granularity of metadata and data blocks within SSTables and restructures SSTables through decoupling at the structural level.

2.3 Key Value Storage Systems for ZNS SSDs

With the emergence of ZNS SSDs, an increasing number of studies have explored how to adapt LSM-tree to storage environments with sequential zone writes and host-managed storage devices. One category of work designs file systems or storage interfaces tailored for ZNS SSDs, which provide sequential write friendly abstractions for upper-layer key value storage systems [12]. Another category focuses on the application layer and leverages the access and compaction characteristics of LSM-tree to design zone allocation strategies that are aware of compaction behavior, thereby reducing the migration of valid data during zone reset operations [8].

These approaches have achieved certain effectiveness in reducing write amplification and extending device lifetime in ZNS SSD environments. However, their optimization granularity is mainly limited to the zone or SSTable level, and SSTables are typically treated as the smallest units for data placement and reclamation. How the internal structure of SSTables affects zone utilization and write amplification has not been systematically analyzed or specifically optimized in existing studies. Therefore, it remains necessary to explore optimization opportunities for LSM-tree on ZNS SSDs from a finer-grained structural perspective.

2.4 LSM-tree for Hybrid Storage

With the emergence of Non-Volatile Memory NVM, an increasing number of studies have explored the integration of NVM with traditional storage devices such as SSDs to construct hybrid storage systems that simultaneously provide high performance and large capacity [3]. These studies typically leverage the low access latency and high endurance of NVM by placing critical data structures or hot data in NVM, thereby improving overall system performance [14].

In LSM-tree related research, some studies attempt to store index structures, logs, or small-scale data in NVM to reduce write latency or lower the data migration overhead during compaction [15]. These approaches improve write performance to a certain extent. However, their optimization granularity is mainly limited to the key value or index level, and SSTables are still organized and managed as integral units without targeted restructuring of their internal structure. As a result, although such methods leverage the performance benefits of hybrid storage, they remain unable to avoid write amplification caused by inappropriate SSTable structure design in ZNS SSD environments.

In summary, existing studies have explored the write amplification problem from multiple perspectives, including LSM-tree compaction optimization, key value separation, and hybrid storage systems. However, most prior work still treats SSTables as indivisible basic storage units and pays limited attention to the internal structural design between metadata and data blocks within SSTables and its impact

on write amplification. In contrast, our work starts from the internal structure of SSTables and proposes an LSM-tree design based on SSTable decoupling. Through decoupled storage and a logical merge mechanism, our approach provides a new direction for reducing write amplification.

3 SSTable Decoupling and Restructuring Strategy

Existing LSM-tree based key value storage systems typically store key value pairs and metadata of SSTables together on ZNS SSDs. This design has gradually exposed systematic limitations in practical deployments. Specifically, when key value pairs in an SSTable are updated, ZNS SSDs mainly rely on sequential write operations, which usually append new data at the end of a zone. Although this mechanism ensures data consistency and crash safety, it results in multiple physical versions of the same logical data residing on the storage medium. Such data redundancy grows rapidly with increasing update frequency, leading not only to reduced storage space utilization but also to severe write amplification.

Meanwhile, we observe that during key lookup, an SSTable first parses its footer to obtain the index information of the Meta Block and the Index Block. It then uses the Index Block to quickly locate the starting address of the target key within the corresponding Data Block and employs the Bloom filter to determine whether the target key exists in that Data Block. During this process, SSTable metadata is accessed frequently, whereas key value data is mainly read sequentially only after a lookup hits. As a result, SSTable metadata exhibits characteristics of high access frequency and small scale updates, while key value data is dominated by large volume storage and low access frequency. Based on this observation, we propose an SSTable decoupling and restructuring strategy.

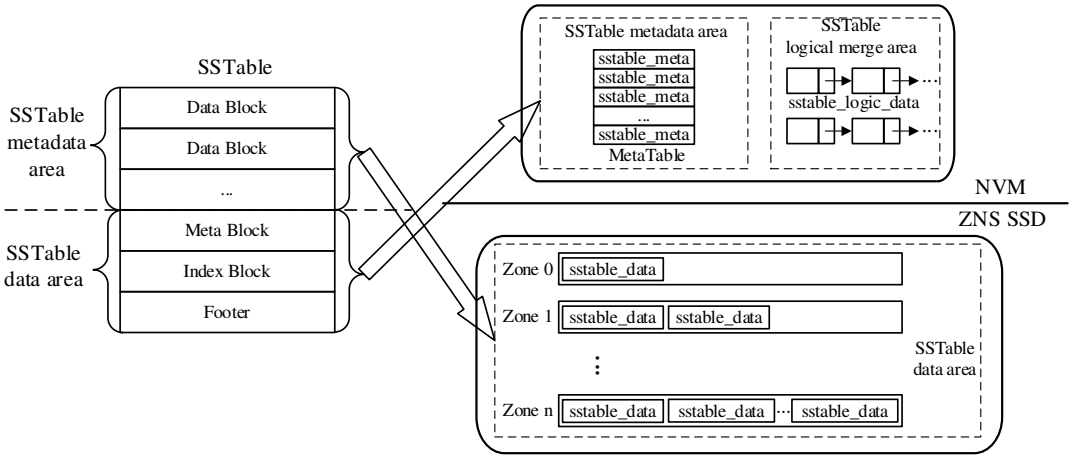


Fig. 1 Description of decoupling and reconstruction strategy for SSTable

As shown in Figure 1, an SSTable is decomposed into a data region referred to as sstable_data and a metadata region referred to as sstable_meta. The region accounts for the majority of storage capacity and is accessed infrequently, so it is retained in the sequential write area of ZNS SSDs to fully leverage their large capacity advantage. In contrast, the sstable_meta region is accessed frequently and is therefore migrated to NVM, which provides byte addressability and non volatility.After SSTable decoupling, the storage organization of metadata and data regions changes significantly. The metadata is no longer colocated with key value data on ZNS SSDs. Instead, it is independently managed in a metadata table called MetaTable in NVM to fully exploit the low latency and byte addressable characteristics of NVM. In addition, to further optimize SSTable merging behavior, we construct a logical merge area for SSTables in NVM, which stores logically merged structures referred to as sstable_logic_data. This design reduces data rewrite overhead during merge operations.

4 Zone Aware SSTable Metadata

After SSTable metadata is migrated from ZNS SSDs to NVM, write amplification is reduced. However, the original metadata design is still based on block level storage characteristics of traditional SSDs and is not optimized for the zone management mechanism of ZNS SSDs. In conventional SSTable structures, metadata is typically stored in the Meta Block and Index Block and mainly includes information such

as the associated level, key range, and block offset. These designs are intended to accommodate random writes and block based management in traditional SSDs.In contrast, ZNS SSDs use zones as the minimum management unit, enforce sequential writes, and do not support in place updates. As a result, traditional SSTable metadata structures exhibit inherent limitations when deployed on ZNS SSDs. In particular, during key value lookup operations, these limitations may introduce additional access latency and negatively impact overall system performance. Therefore, based on the decoupled SSTable design, we reconstruct the SSTable metadata structure, as illustrated in Figure 2.

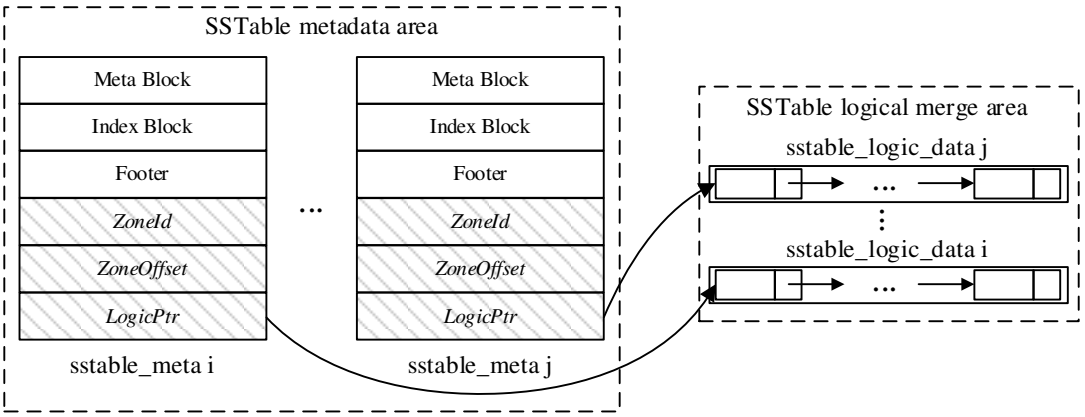


Fig. 2 Structure of SSTable metadata

To adapt to the zone management characteristics of ZNS SSDs while fully leveraging the high speed access capability of NVM, we embed several new metadata fields into the metadata region, including ZoneId, ZoneOffset, and LogicPtr. ZoneId uniquely identifies the zone in the ZNS SSD data region where sstable_data is located, enabling fast zone level positioning and avoiding the overhead introduced by traditional block level addressing. ZoneOffset further provides the precise offset of sstable_data within the zone, allowing the system to efficiently locate the exact position. In addition, LogicPtr stores a pointer to the logically merged sstable_logic_data, which facilitates subsequent writes to sstable_data.

The SSTable decoupling and restructuring strategy improves system performance in multiple aspects. First, NVM supports overwrite operations, which allows metadata updates to avoid the append only write constraint of ZNS SSDs. This fundamentally reduces the generation of redundant data and effectively alleviates write amplification. Second, NVM provides nanosecond level access latency. Compared with traditional SSDs, its lower access overhead significantly improves metadata lookup efficiency and accelerates LSM-tree query processing

In addition, decoupling metadata from key value data enables zones on ZNS SSDs to be used exclusively for storing large sequentially written data blocks. This design preserves zone space continuity and improves utilization efficiency, thereby increasing the effective data density within each zone. Meanwhile, embedding new metadata fields in the metadata region enables fast localization of sstable_data within a zone. This reduces the overhead of zone level lookup and improves the efficiency of logical updates. As a result, the proposed strategy avoids systemic performance degradation caused by frequent metadata updates in hybrid storage architectures and further optimizes overall storage system performance.

5 SSTable Logical Merge Strategy

In traditional LSM-tree designs, SSTable merge operations require multiple SSTable files to be merged into a new SSTable file, which involves extensive data migration and rewriting. On conventional SSDs, although such operations introduce a certain level of write amplification, their impact is relatively manageable due to support for random writes. However, on ZNS SSDs, where data is written sequentially at the zone level and in place updates are not supported, merge operations introduce a series of challenges.First, merge operations require zones to be erased and rewritten, which not only exacerbates write amplification but also significantly increases write latency, thereby degrading system performance. Second, frequent zone erase and rewrite operations accelerate wear on ZNS SSDs and shorten device lifetime. These issues make traditional merge operations inefficient on ZNS SSDs. To address these challenges, we propose an SSTable logical merge strategy.

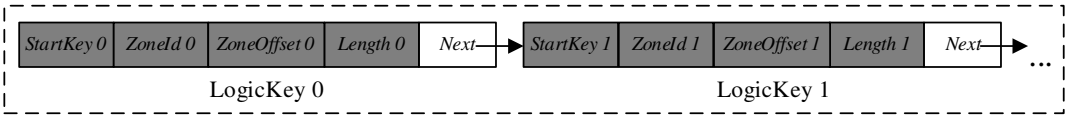


Fig. 3 Structure of sstable_logic_data

We first present the structure of sstable_logic_data after SSTable logical merging, as illustrated in Figure 3. The sstable_logic_data structure is designed to efficiently utilize Non Volatile Memory resources and to accommodate metadata update requirements after logical merging. This structure records metadata of contiguous keys in sstable_data through LogicKey entries.Each LogicKey entry includes several key fields. StartKey represents the starting key of a contiguous key range. ZoneId identifies the zone on the ZNS SSD where the starting key is located, and ZoneOffset specifies the corresponding offset within that zone. Length denotes the span of the contiguous key range. The Next pointer links to the subsequent LogicKey entry. Based on this structure, SSTable logical merging is performed, as shown in Figure 4.

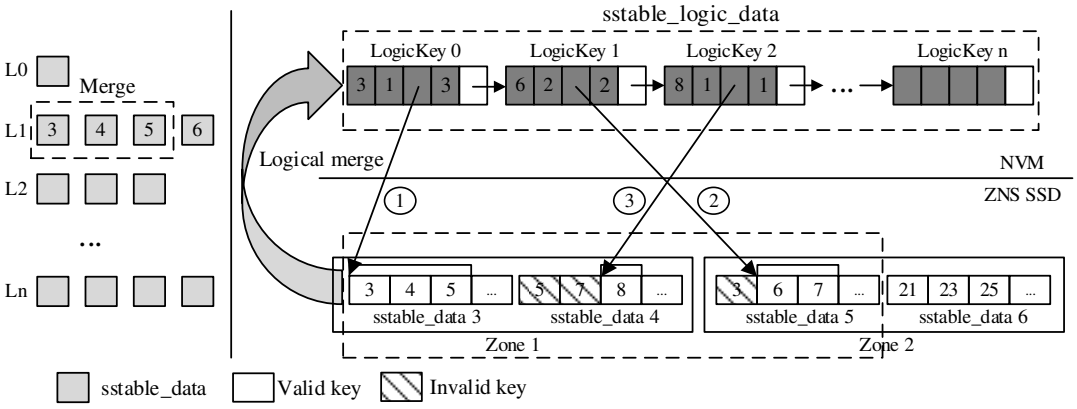


Fig. 4 Description of logical merge strategy for SSTable

When sstable_data in the ZNS SSD data region triggers a merge operation, the system first reads the smallest key among the sstable_data to be merged and determines the maximum span of its contiguous keys. The zone identifier, zone offset, and span length of the starting key are then recorded in the LogicKey entry indexed as i in sstable_logic_data. Next, the largest key covered by LogicKey i is compared with keys from other sstable_data instances that participate in the merge. The system identifies the smallest key that is larger than the maximum key of LogicKey i, reads this key and determines the maximum span of its contiguous keys, and records the corresponding zone identifier, zone offset, and span length in LogicKey entry indexed as i plus one in sstable_logic_data. This comparison and insertion process is repeated iteratively until the last key from all sstable_data instances to be merged is recorded in sstable_logic_data.

After logical merging is completed, a new metadata entry referred to as sstable_meta is inserted into the MetaTable to point to the newly generated sstable_logic_data, and the metadata of the original sstable_data instances to be merged is removed. When the amount of invalid sstable_data within a zone on the ZNS SSD exceeds a predefined threshold, the system cooperatively uses sstable_logic_data and sstable_meta to migrate the newly generated sstable_data to the corresponding zone in the data region through device supported copy operations. Meanwhile, the sstable_logic_data in the logical merge area is deleted. The detailed procedure is presented in Algorithm 1.

The SSTable logical merge strategy effectively reduces the number of write operations and the frequency of zone erasures on ZNS SSDs, thereby mitigating write amplification and extending device lifetime. In addition, this strategy fully leverages NVM for efficient metadata management, which not only improves metadata access efficiency but also optimizes space utilization in key value storage systems.

Algorithm 1 SSTable Logical Merge Procedure

Input: *sstable_data*
Output: *sstable_logic_data*, *sstable_meta*

when *sstable_data* trigger compaction in ZNS SSD **then**
  for each *sstable_data* **in** sstable_data_list **do**
    keysList = ReadKeysFromZone(*sstable_data*);
  end for
  if keysList is not empty **then**
    $Startkey$ = GetKeyListMinKey (keysList);
    $ZoneId, ZoneOffset$ = GetMinKeyAddr(Startkey);
    $Length$ = FindMaxContinuousRange(Startkey, keysList);
    $LogicKeyi$ = CreateLogicKey($Startkey, ZoneId, ZoneOffset, Length$);
    UpdateNode(*sstable_logic_data*, $LogicKeyi$);
    RemoveKey(keysList, $Startkey, Length$);
    i++;
  else
    sstable_meta = CreateMeta(*sstable_logic_data*);
    InsertMetaTable(*sstable_meta*);
    RemoveInvalidMeta(*sstable_data*);
  end if
end when
for each Zone **in** ZNS SSD **do**
  if the invalid *sstable_data* in the Zone exceeds the threshold **then**
    WriteToZNS(*sstable_logic_data*, *sstable_meta*);
    sstable_logic_data.delete();
  end if
end for

6 Evaluation

To evaluate the effectiveness of the SSTable decoupled LSM-tree, we build a prototype system named SPMS based on a previously established experimental environment. Specifically, we modify the source code of the PMem driver and NVMeVirt [9] to decouple and restructure the SSTable organization and to incorporate the SSTable logical merge strategy. In addition, we construct several comparison systems, including a RocksDB prototype system based on traditional SSDs, a SplitKV prototype system that combines traditional SSDs and PMem [7], and a ListDB prototype system based on PMem [10]. These systems are evaluated and compared with SPMS.

YCSB is used as the benchmarking tool in our experiments. The number of client threads is configured as 1, 4, 8, 16, and 32 to evaluate throughput, write amplification, write latency, and concurrency performance under multi-threaded workloads for each prototype system. All experiments are conducted on a single server, and the detailed software and hardware configurations are presented in Table 1.

Table 1 Configuration of Test Environment

Component	Configuration
CPU	Intel Xeon Platinum 8300 3.00 GHz
Memory	128GB
NVDIMM	2 * 512GB Intel Optane DC Persistent Memory
ZNS SSD	NVMeVirt, 128GB, 64MB Zone Size
Disk	512GB SATA SSD
OS	CentOS 7.0, Kernel 5.18.18

6.1 Write Throughput under Different Data Volumes

We first use the YCSB Load workload, which consists of 100% put operations. The number of threads is set to 16, and different numbers of key value pairs are randomly generated and inserted into RocksDB,

SplitKV, ListDB, and SPMS. We then measure write throughput under different data volumes. The experimental results are shown in Figure 5.

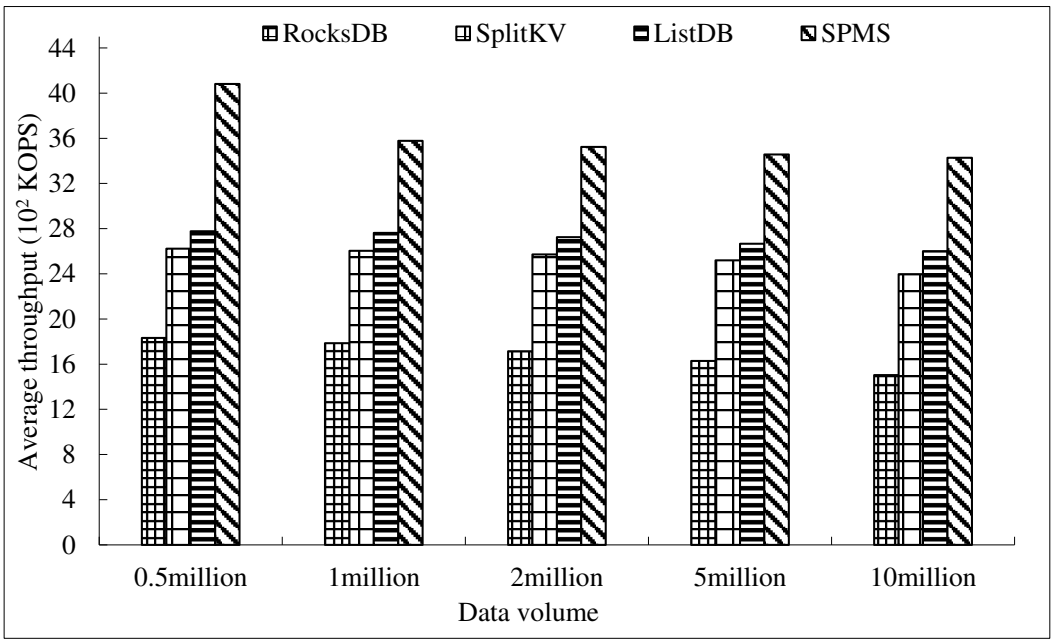


Fig. 5 Write throughput under different data volumes

As shown in Figure 5, SPMS consistently achieves higher write throughput than the other prototype systems. Compared with RocksDB, SPMS improves write throughput by 100.2% to 128.0%. Compared with SplitKV, SPMS achieves improvements ranging from 37.1% to 55.5%. Compared with ListDB, SPMS improves write throughput by 29.5% to 46.9%. This performance gain is attributed to the SSTable decoupling and restructuring strategy adopted by SPMS, which stores the SSTable metadata region and data region separately on NVM and ZNS SSDs during write operations. NVM provides write performance close to that of DRAM, while ZNS SSDs offer higher concurrent write capability than traditional SSDs. In addition, the proposed SSTable logical merge strategy effectively matches the characteristics of NVM and maintains high write efficiency.

As the amount of data written to each prototype system increases from one million to ten million key value pairs, the write throughput of RocksDB, SplitKV, and ListDB decreases by 15.8%, 7.9%, and 5.8%, respectively. In contrast, the write throughput of SPMS decreases by only 4.1%, indicating stronger write performance stability. RocksDB adopts a traditional LSM-tree for data storage, and the frequency of inter level compactions increases as data volume grows. This behavior introduces significant additional write overhead and triggers more garbage collection operations, which further reduce write throughput. Although SplitKV integrates traditional SSDs and NVM, it still relies on the LSM-tree structure. As data volume increases, its write path continues to perform frequent compaction operations, especially when NVM space approaches capacity, forcing data migration to SSDs and incurring additional write overhead. ListDB exhibits a smaller performance degradation compared with RocksDB and SplitKV. However, as data scale increases, ListDB still suffers from rising metadata management overhead, which gradually degrades write performance. In contrast, SPMS decomposes and persists SSTables according to the characteristics of different storage devices and maintains stable write performance even as data volume increases.

6.2 Write Throughput under Multi Threaded Concurrency

Next, we use YCSB to generate one million key value pairs and insert them into RocksDB, SplitKV, ListDB, and SPMS, respectively. By varying the number of threads, we evaluate the write throughput of each prototype system under multi threaded concurrent workloads. The experimental results are shown in Figure 6.

The experimental results indicate that under multi threaded workloads, SPMS improves write throughput by 29.1% to 137.0% compared with RocksDB. Compared with SplitKV, SPMS achieves

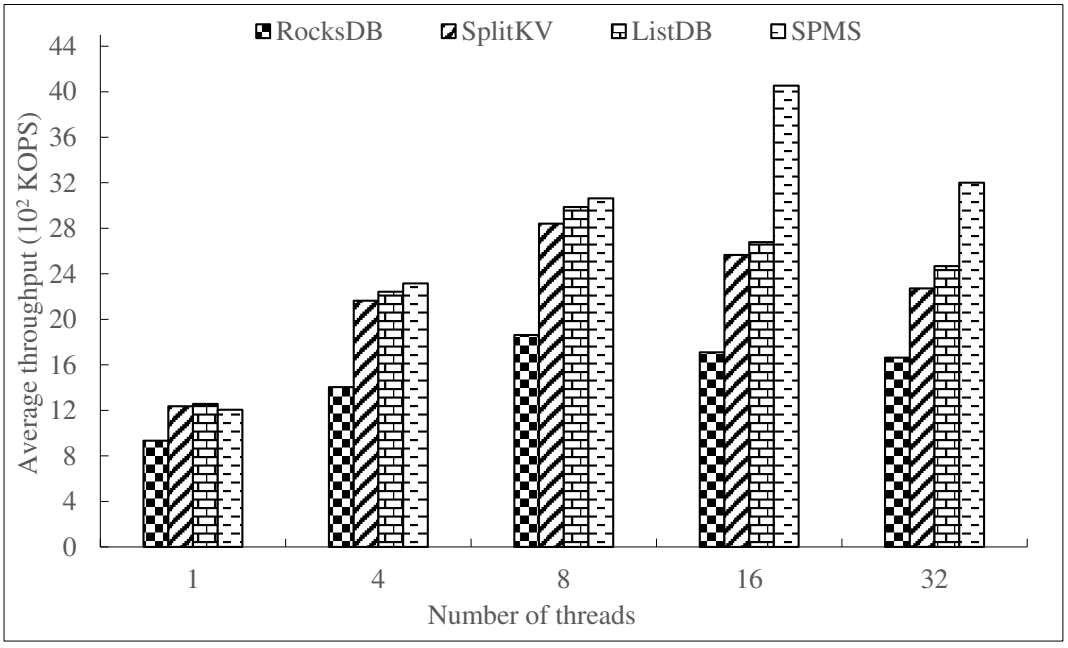


Fig. 6 Write throughput under multiple threads

improvements ranging from 2.4% to 57.9%, and it improves write throughput by 2.5% to 51.2% compared with ListDB. When the number of threads is small, the write throughput of SPMS is slightly lower than that of SplitKV and ListDB. This behavior is mainly because, in single threaded scenarios, SPMS cannot perform parallel writes to the data region and the metadata region, which constrains the write path and limits overall throughput.However, as the number of threads increases, the write throughput of SPMS steadily improves and demonstrates good scalability. In particular, with 16 threads, the write throughput of SPMS is 57.9% higher than that of SplitKV and 51.2% higher than that of ListDB.

SplitKV and ListDB adopt NVM and traditional SSDs as storage devices. Under workloads with a small number of threads, the high throughput capability of NVM can be effectively utilized. However, when the number of threads increases to eight or more, the write performance of NVM reaches its peak. With sixteen and thirty two threads, write throughput begins to decline due to the I/O bandwidth limitations of the storage devices.

In contrast, SPMS adopts a hybrid storage architecture composed of NVM and ZNS SSDs and fully exploits the advantages of ZNS SSDs under highly concurrent write workloads. When the number of threads increases to sixteen, the sequential write characteristics and high concurrency handling capability of ZNS SSDs are effectively utilized, allowing SPMS to continue achieving increasing write throughput. In addition, by decoupling SSTables, SPMS enables key value data and metadata to be mapped to storage devices that best match their access characteristics. This design further improves storage efficiency, reduces write conflicts, and increases overall write throughput.

6.3 Read Throughput under Different Data Volumes

We use YCSB with workload C, which consists of 100% get operations, and set the number of threads to 16. By varying the number of key value pairs stored in each prototype system, we evaluate read throughput under different data volumes. The experimental results are shown in Figure 7.

As shown in Figure 7, SPMS consistently achieves higher read throughput than the other prototype systems under different data volumes. Compared with RocksDB, SplitKV, and ListDB, SPMS improves read throughput by 123.6% to 130.3%, 68.6% to 71.2%, and 0.8% to 5.1%, respectively. This performance improvement is mainly attributed to the optimization strategies adopted by SPMS in data storage and metadata management.

SPMS employs a new SSTable metadata management mechanism that enables query requests to locate target data more efficiently and reduces unnecessary I/O accesses, thereby improving read performance. In addition, SPMS migrates SSTable metadata to NVM and leverages the low latency and high read and write performance of NVM to accelerate metadata access. As the amount of written data increases, the read throughput of each prototype system gradually decreases. This behavior occurs because SplitKV is

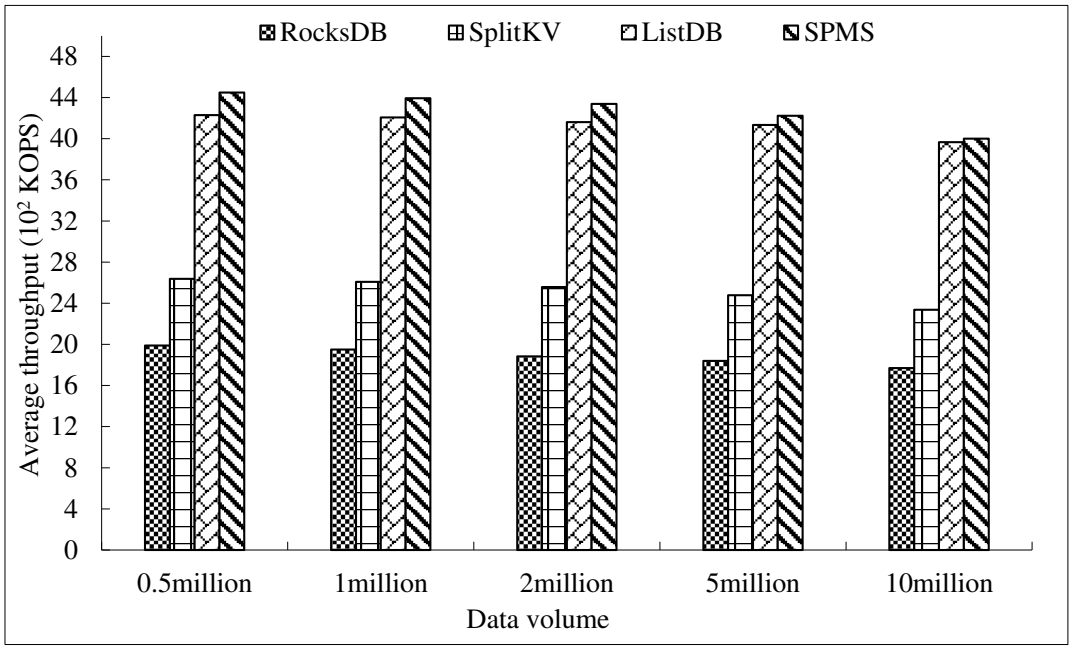


Fig. 7 Read throughput under different data volumes

designed to store hot data in NVM and cold data in traditional SSDs in order to improve read and write performance. However, as data volume increases, the hot and cold status of data may change. Some data that was originally stored as hot data in NVM may be demoted to cold data and migrated to SSDs. As a result, query operations still need to access slower SSDs, which leads to reduced read throughput. In contrast, ListDB stores data in NVM using an append only approach. While this design improves write performance, data may be stored in a non contiguous manner during queries. Consequently, the system needs to traverse long log chains to locate target data.As data volume grows, the length of log chains increases, which further raises traversal overhead during query processing and eventually degrades read throughput.

6.4 Read Throughput under Multi Threaded Concurrency

We use YCSB with workload C, which consists of 100% get operations. The number of threads is set to 1, 4, 8, 16, and 32 to evaluate the read throughput of each prototype system under multi threaded concurrent workloads. The experimental results are shown in Figure 8.

As shown in Figure 8, the read throughput of SPMS gradually increases as the number of threads grows. With one and four threads, SPMS achieves higher read throughput than RocksDB and SplitKV, but slightly lower throughput than ListDB. This behavior is mainly because ListDB optimizes its index structure, which effectively improves read throughput under workloads with a small number of threads.

In contrast, SPMS adopts an architecture that separates SSTable key value data and metadata. Under single threaded workloads, SPMS needs to access multiple devices during read operations, which introduces additional I/O overhead and results in slightly lower read performance compared with ListDB. However, as the number of threads increases, the advantages of SPMS become more evident. When the number of threads reaches sixteen, SPMS improves read throughput by 2.9% compared with ListDB. This result indicates that under multi threaded concurrent workloads, the new SSTable metadata design adopted by SPMS significantly improves lookup efficiency on ZNS SSDs.By decoupling SSTable metadata and migrating it to NVM, SPMS leverages the near DRAM read and write performance of NVM to reduce data access latency under multi threaded workloads, thereby improving overall read throughput.

6.5 Throughput under Mixed Workloads with Different Data Volumes

We use YCSB with workload A, which consists of 50% get operations and 50% update operations. The number of threads is set to 16. By varying the number of key value pairs stored in each prototype system, we evaluate mixed workload throughput under different data volumes. The experimental results are shown in Figure 9.

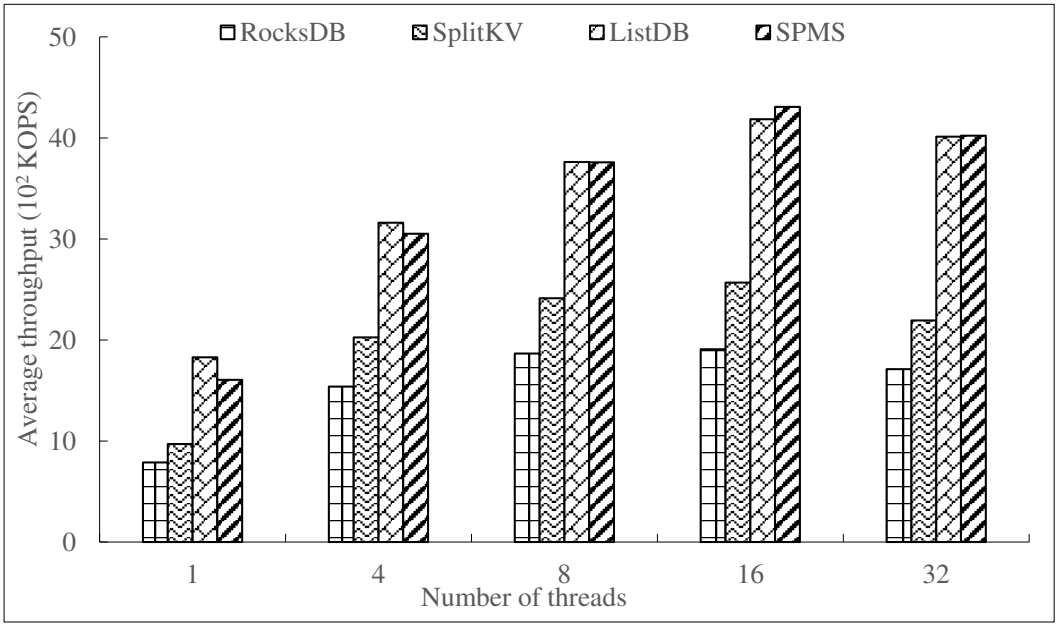


Fig. 8 Read throughput under multiple threads

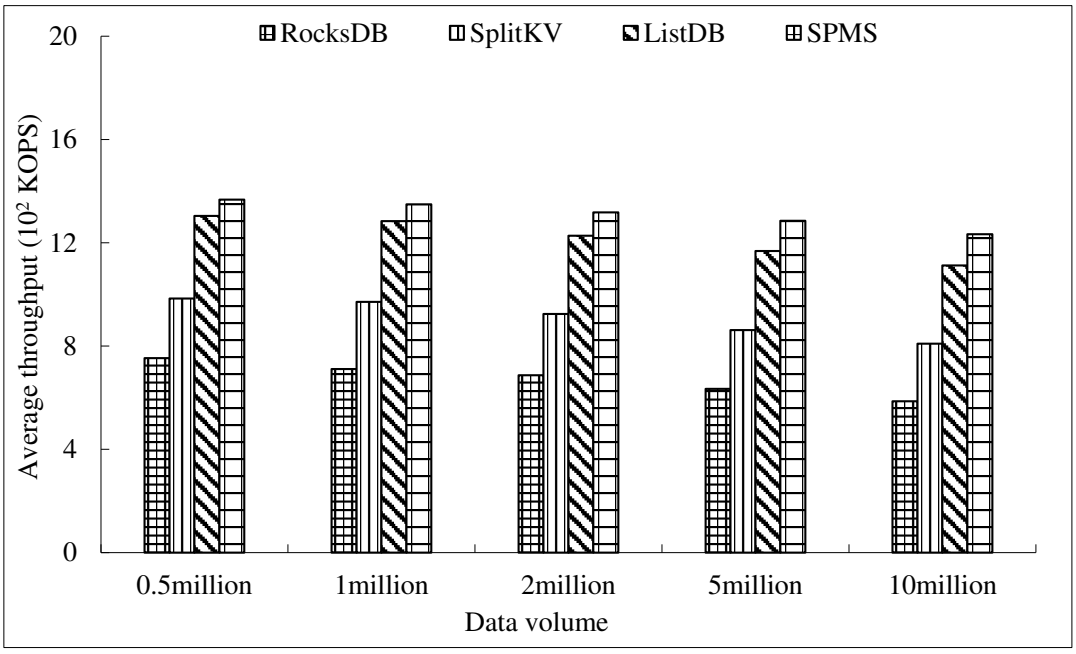


Fig. 9 Hybrid operation throughput under different data volumes

The results shown in Figure 9 indicate that under mixed workload scenarios, SPMS consistently achieves significantly higher throughput than the other prototype systems across different data volumes. Specifically, compared with RocksDB, SplitKV, and ListDB, SPMS improves throughput by 81.5% to 110.4%, 38.9% to 52.4%, and 4.8% to 10.8%, respectively. This advantage is mainly attributed to the optimized storage architecture and metadata management strategy adopted by SPMS, which enables more efficient handling of mixed read and write workloads.

SPMS adopts a hybrid storage architecture composed of NVM and ZNS SSDs. It leverages NVM to store the SSTable metadata region and accelerate index access, while distributing the SSTable data region across ZNS SSDs in a zone aware manner. This design effectively reduces write amplification and improves read efficiency, thereby enhancing overall throughput under mixed workloads. In addition, SPMS introduces an SSTable logical merge strategy that reduces unnecessary data movement and I/O accesses during merge operations, thereby lowering the additional storage overhead caused by merging. In

contrast, RocksDB still relies on traditional SSDs as storage devices and is more severely affected by write amplification under mixed workloads, which leads to a noticeable decline in mixed workload throughput. Although SplitKV and ListDB partially leverage NVM, they are prone to limitations imposed by NVM read and write bandwidth as data volume increases. This issue becomes more pronounced under highly concurrent workloads, where storage bottlenecks are more likely to occur and degrade overall throughput.

From the experimental trends, it can be observed that as data scale grows, SPMS exhibits a smaller decline in mixed workload throughput compared with the other systems. This result indicates that SPMS provides better scalability and stability when handling large scale data workloads.

6.6 Write Amplification under Different Key Value Sizes

To evaluate write amplification behavior of different key value storage systems, we use db_bench to measure write amplification for RocksDB, SplitKV, ListDB, and SPMS. In the experiments, a fixed key size of 16 bytes is used, while values with different sizes are tested to analyze the impact of key value size on write amplification. The experimental results are shown in Figure 10.

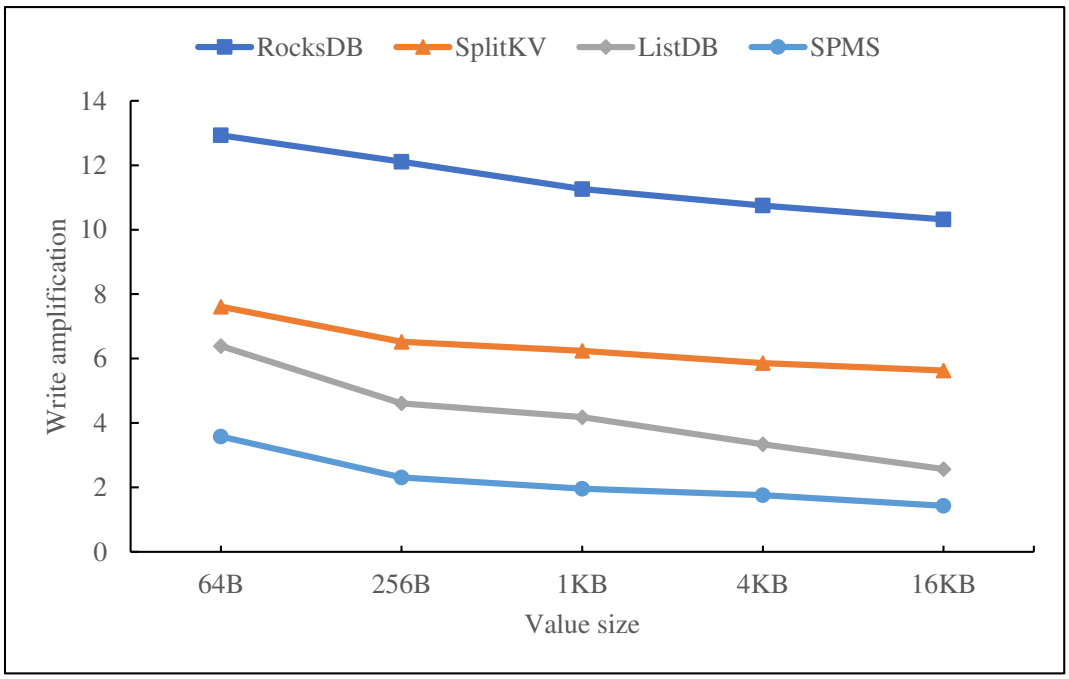


Fig. 10 Write amplification

The experimental results show that as the key value size increases, the write amplification of RocksDB remains consistently higher than 10. This behavior occurs because RocksDB adopts a traditional LSM-tree structure and does not optimize write amplification during compaction. In addition, RocksDB relies on conventional SSD storage and performs block level erase and write operations, which exacerbate frequent data rewrites during compaction and further increase write amplification.

In comparison, the write amplification of SplitKV remains between 6 and 8. Although SplitKV maintains key indexes within the LSM-tree, updating values requires modifying the value pointers recorded in the LSM-tree, which triggers rewrites of related SSTables. These rewrites introduce additional write operations, so write amplification is reduced compared with RocksDB but remains relatively high.

Among all evaluated prototype systems, SPMS demonstrates significant advantages. Compared with RocksDB, SplitKV, and ListDB, SPMS reduces write amplification by 72.3% to 86.1%, 52.9% to 74.6%, and 43.9% to 53.1%, respectively. This improvement is mainly attributed to the SSTable logical merge strategy adopted by SPMS, which merges multiple SSTables at the logical level rather than rewriting physical data, thereby reducing unnecessary data rewrites and compaction overhead.

6.7 Ablation Study

Based on SPMS, we remove the SSTable logical merge strategy and adopt a traditional intra level compaction strategy to construct a prototype system referred to as SPMS w/o. By varying the amount of written data, we measure the number of zone reset operations for each prototype system. The prototype systems are configured with 2048 zones, and the program erase cycle of each zone is set to 10. The experimental results are shown in Figure 11.

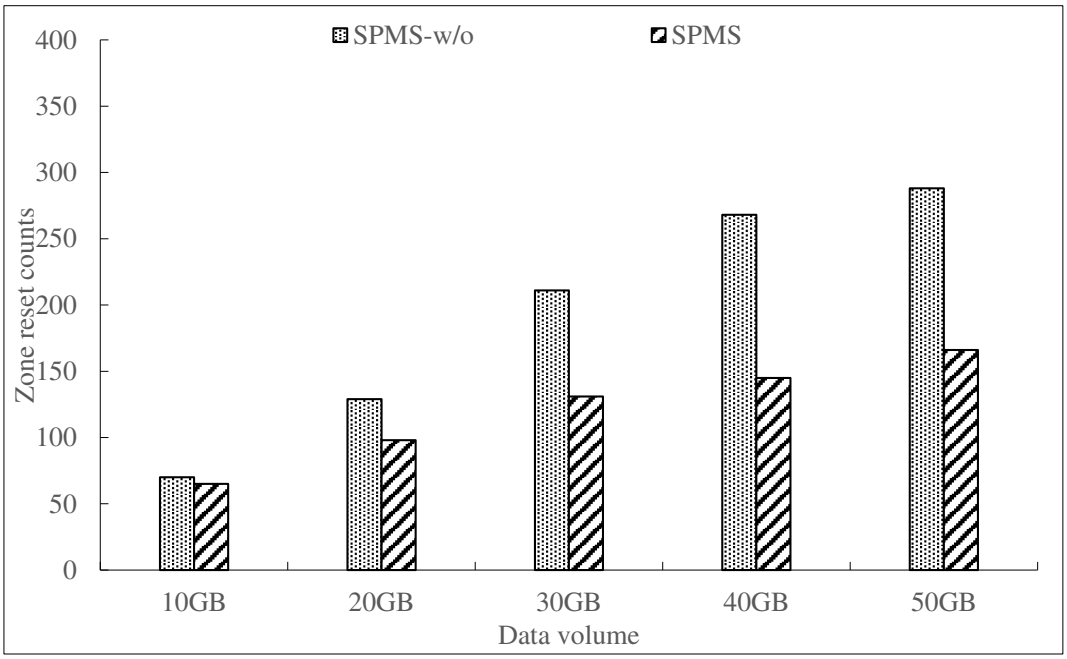


Fig. 11 Number of zone resets

The results shown in Figure 11 indicate that as data volume increases, the prototype system SPMS w/o, which adopts a traditional intra level compaction strategy, consistently exhibits a higher number of zone reset operations than SPMS, which employs the SSTable logical merge strategy. This difference is not significant when the data volume is small. However, as data volume grows, the gap gradually widens.When the data volume reaches 10 GB, the system remains in a write intensive state and only a limited number of SSTable merge operations occur. As a result, the difference in zone reset counts between the two prototype systems is relatively small. When the data volume increases to 20 GB, a certain number of SSTable merge operations are triggered within the system, and the number of zone resets in SPMS w o becomes 24.0% higher than that in SPMS. As the data volume further increases to 30 GB and 40 GB, the gap continues to widen, reaching 37.9% and 45.8%, respectively.

These results demonstrate that the SSTable logical merge strategy adopted by SPMS significantly improves adaptability to ZNS SSDs. By reducing the amount of key value data written during SSTable merge operations, SPMS is able to migrate valid key value data to new zones through device supported copy operations when invalid SSTables accumulate to a certain level, thereby effectively reducing the number of zone reset operations. In contrast, SPMS w/o relies on a traditional intra level compaction strategy, which causes a large number of invalid SSTable files to remain within zones and frequently triggers zone reset operations.

When the data volume increases to 50 GB, the number of zone reset operations for both prototype systems gradually stabilizes. This behavior occurs because as data volume continues to grow, existing SSTables are progressively merged into lower levels, and no merge operations are triggered for a relatively long period.

7 Conclusion

We investigate the limitations of existing LSM-tree based key value storage systems on ZNS SSDs, with a particular focus on the write amplification caused by tightly coupled storage of key value data and

metadata in SSTables and the insufficient adaptation to ZNS SSD characteristics. To address these issues, we propose DS-LSM-tree, an LSM-tree design based on SSTable decoupling.

The proposed design consists of several key components. First, we present an SSTable decoupling and restructuring strategy and introduce a new SSTable metadata structure. Next, we propose an SSTable logical merge strategy to reduce unnecessary data rewrites during merge operations. Finally, we implement a prototype system based on DS-LSM-tree and conduct comprehensive evaluations using YCSB benchmarks. Experimental results demonstrate the performance advantages of DS-LSM-tree compared with RocksDB, SplitKV, and ListDB.

References

[1] Bjørling M (2020) Zone append: A new way of writing to zoned storage. Santa Clara, CA, February USENIX Association

[2] Bjørling M, Aghayev A, Holmberg H, et al (2021) ZNS: avoiding the block interface tax for flash-based ssds. In: Calciu I, Kuenning G (eds) Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021. USENIX Association, pp 689–703, URL https://www.usenix.org/conference/atc21/presentation/bjorling

[3] Chen H, Ruan C, Li C, et al (2021) Spandb: A fast, cost-effective lsm-tree based KV store on hybrid storage. In: Aguilera MK, Yadgar G (eds) 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021. USENIX Association, pp 17–32, URL https://www.usenix.org/conference/fast21/presentation/chen-hao

[4] Dayan N, Idreos S (2018) Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In: Das G, Jermaine CM, Bernstein PA (eds) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. ACM, pp 505–520, https://doi.org/10.1145/3183713.3196927

[5] Dayan N, Athanassoulis M, Idreos S (2017) Monkey: Optimal navigable key-value store. In: Salihoglu S, Zhou W, Chirkova R, et al (eds) Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. ACM, pp 79–94, https://doi.org/10.1145/3035918.3064054

[6] Dong S, Kryczka A, Jin Y, et al (2021) Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. ACM Trans Storage 17(4):26:1–26:32. https://doi.org/10.1145/3483840

[7] Han S, Jiang D, Xiong J (2020) Splitkv: Splitting IO paths for different sized key-value items with advanced storage devices. In: Badam A, Chidambaram V (eds) 12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020. USENIX Association, URL https://www.usenix.org/conference/hotstorage20/presentation/han

[8] Hwang I, Lee S, Kim S, et al (2025) Z-LFS: A zoned namespace-tailored log-structured file system for commodity small-zone ZNS ssds. In: Altinbüken D, Stutsman R (eds) Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025. USENIX Association, pp 547–562, URL https://www.usenix.org/conference/atc25/presentation/hwang

[9] Kim S, Shim J, Lee E, et al (2023) Nvmevirt: A versatile software-defined virtual nvme device. In: Goel A, Naor D (eds) 21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023. USENIX Association, pp 379–394, URL https://www.usenix.org/conference/fast23/presentation/kim-sang-hoon

[10] Kim W, Park C, Kim D, et al (2022) Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory. In: Aguilera MK, Weatherspoon H (eds) 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022. USENIX Association, pp 161–177, URL https://www.usenix.org/conference/

osdi22/presentation/kim

[11] Lu L, Pillai TS, Gopalakrishnan H, et al (2017) Wisckey: Separating keys from values in ssd-conscious storage. ACM Trans Storage 13(1):5:1–5:28. https://doi.org/10.1145/3033273

[12] Min J, Zhao C, Liu M, et al (2023) ezns: An elastic zoned namespace for commodity ZNS ssds. In: Geambasu R, Nightingale E (eds) 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023. USENIX Association, pp 461–477, URL https://www.usenix.org/conference/osdi23/presentation/min

[13] O'Neil PE, Cheng E, Gawlick D, et al (1996) The log-structured merge-tree (lsm-tree). Acta Informatica 33(4):351–385. https://doi.org/10.1007/S002360050048

[14] Qiu J, Yuan F, Gao M, et al (2025) Hotrap: Hot record retention and promotion for lsm-trees with tiered storage. In: Altinbüken D, Stutsman R (eds) Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025. USENIX Association, pp 497–511, URL https://www.usenix.org/conference/atc25/presentation/qiu

[15] Raina A, Lu J, Cidon A, et al (2023) Efficient compactions between storage tiers with prismdb. In: Aamodt TM, Jerger NDE, Swift MM (eds) Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023. ACM, pp 179–193, https://doi.org/10.1145/3582016.3582052

[16] Rao DS, Kumar S, Keshavamurthy AS, et al (2014) System software for persistent memory. In: Bulterman DCA, Bos H, Rowstron AIT, et al (eds) Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014. ACM, pp 15:1–15:15, https://doi.org/10.1145/2592798.2592814

[17] Yao T, Zhang Y, Wan J, et al (2020) Matrixkv: Reducing write stalls and write amplification in lsm-tree based KV stores with matrix container in NVM. In: Gavrilovska A, Zadok E (eds) Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020. USENIX Association, pp 17–31, URL https://www.usenix.org/conference/atc20/presentation/yao

[18] Zhang Q, Li Y, Pan Y, et al (2025) Mitigating resource usage dependency in sorting-based KV stores on hybrid storage devices via operation decoupling. In: Altinbüken D, Stutsman R (eds) Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025. USENIX Association, pp 513–529, URL https://www.usenix.org/conference/atc25/presentation/zhang-qingyang