Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

MyWAL: performance optimization by removing redundant input/output stack in key-value store*

Xiao ZHANG†‡[1,2,3], Mengyu LI†[2,4], Michael NGULUBE[1,2], Yonghao CHEN[1,2], Yiping ZHAO[1]

[1]*School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China*

[2]*MIIT Key Laboratory of Big Data Storage and Management, Northwestern Polytechnical University, Xi'an 710072, China*

[3]*National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology, Northwestern Polytechnical University, Xi'an 710072, China*

[4]*School of Software, Northwestern Polytechnical University, Xi'an 710072, China*

†E-mail: zhangxiao@nwpu.edu.cn; limy@mail.nwpu.edu.cn

Received Oct. 21, 2022; Revision accepted Mar. 5, 2023; Crosschecked June 27, 2023

Abstract: Based on a log-structured merge (LSM) tree, the key-value (KV) storage system can provide high reading performance and optimize random writing performance. It is widely used in modern data storage systems like e-commerce, online analytics, and real-time communication. An LSM tree stores new KV data in the memory and flushes to disk in batches. To prevent data loss in memory if there is an unexpected crash, RocksDB appends updating data in the write-ahead log (WAL) before updating the memory. However, synchronous WAL significantly reduces writing performance. In this paper, we present a new WAL mechanism named MyWAL. It directly manages raw devices (or partitions) instead of saving data on a traditional file system. These can avoid useless metadata updating and write data sequentially on disks. Experimental results show that MyWAL can significantly improve the data writing performance of RocksDB compared to the traditional WAL for small KV data on solid-state disks (SSDs), as much as five to eight times faster. On non-volatile memory express soild-state drives (NVMe SSDs) and non-volatile memory (NVM), MyWAL can improve data writing performance by 10%–30%. Furthermore, the results of YCSB (Yahoo! Cloud Serving Benchmark) show that the latency decreased by 50% compared with SpanDB.

Key words: Key-value (KV) store; Log-structured merge (LSM) tree; Non-volatile memory (NVM); Non-volatile memory express soild-state drive (NVMe SSD); Write-ahead log (WAL)
https://doi.org/10.1631/FITEE.2200496 **CLC number:** TP392

1 Introduction

A key-value (KV) storage system based on the log-structured merge (LSM) tree (Leavitt, 2010) data structure has better performance and scalability than a traditional relational database. It is widely used in various big data systems. For example, LevelDB (Stonebraker, 2010; Dong et al., 2021) is a fast KV storage library written at Google that provides an ordered mapping from string keys to string values. RocksDB (Facebook, 2019) is a KV store for maximum performance developed by Facebook. The writing performance of the LSM tree is faster than that of B-tree because of its data writing mechanism. The add/update operations are executed in memory, and after sorting and merging, the data in memory are written to the disk.

To avoid in-memory data loss during a system crash, RocksDB records all add/update operations in write-ahead log (WAL). When the memory is full, data will flush into a disk. The KV data are stored on disk

‡ Corresponding author

* Project supported by the National Key Research and Development Project of China (No. 2022YFB2702101), the Shaanxi Province Key Industrial Projects, China (Nos. 2021ZDLGY03-02 and 2021ZDLGY03-08), and the National Natural Science Foundation of China (No. 92152301)

ORCID: Xiao ZHANG, https://orcid.org/0000-0001-7680-1179; Mengyu LI, https://orcid.org/0000-0002-6640-2729

in ordered tables in several layers. In the LSM-tree architecture, there are three kinds of disk writing operations: WAL, flush, and compaction. RocksDB decides when to flush and carry out data compaction, and always writes WAL once there is add/update operation. WAL is an essential mechanism for recovering data in case of a power failure and operating system (OS) crash. However, it has a significant impact on writing performance. For example, it may reduce 50%–90% of writing operations per second (OPS) in different cases.

The writing performance optimization of WAL is important for RocksDB. Non-volatile memory express soild-state drives (NVMe SSDs) and non-volatile memory (NVM) have better input/output (IO) performance than hard disk drive (HDD). Some researchers have proposed optimization solutions based on new storage media or hierarchical architectures. WiscKey saves data on SSD separate storage (Lu et al., 2017). SpanDB optimizes WAL performance by saving WAL on NVMe SSDs (Chen et al., 2021). MatrixKV designes a new L0 layer-management mechanism in NVM that reduces write stalls (Yao et al., 2020). However, these solutions save WAL on the local file system, which maintains unnecessary metadata for the WAL mechanism. For example, when appending a KV operation in WAL, it must update the access time and length of the WAL file.

In this paper, we propose a new WAL mechanism named MyWAL, which manages and saves WAL data to the raw device. It removes unnecessary IO operations caused by the file system, so it can achieve better performance than the current WAL mechanism. We compare the performance under different conditions. The results show that MyWAL can effectively improve the writing performance of KV data on different storage media.

2 Research background and motivation

In this section, we introduce the LSM-tree architecture used in RocksDB and discuss the performance impact of the WAL mechanism. We also compare the writing performance of the file system with that of the raw devices, which motivates our design of MyWAL.

2.1 LSM-tree architecture

LSM tree is a hierarchical data structure. The core idea is that sequential writing performance is much higher than the random writing performance for disk. Fig. 1 shows the typical architecture of the LSM tree used in RocksDB. There are six primary data structures in the LSM tree. Memory table (MMT) and immutable memory table (IMMT) save data in the memory. MMT is a readable and append-write file that saves new or updated KV data. RocksDB saves KV data in a WAL file before it updates in the MMT. When the size of the MMT reaches the threshold, the MMT will change to IMMT. When the size of the IMMT reaches the threshold, the data in IMMT will flush into disks. RocksDB saves KV data in a sorted string table (SST) in several layers. Dumping an IMMT to an SST in L0 is called minor compaction. Merging and resorting several SSTs into a lower layer is called major compaction. RocksDB provides some commonly used compaction algorithms, leveled compaction and universal compaction, to better manage SSTs between different levels.

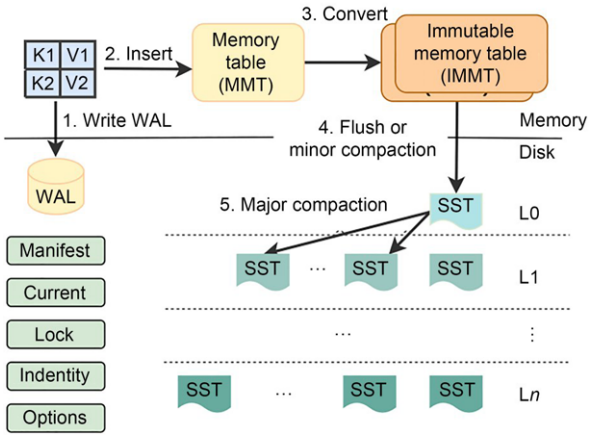

Fig. 1 Architecture of the LSM tree used in RocksDB (LSM: log-structured merge; WAL: write-ahead log; SST: sorted string table)

There are three kinds of disk IO operations in RocksDB: WAL, minor compaction, and major compaction. RocksDB can choose when and how to do compaction, but it has to write WAL as soon as a new update operation executes. The composition of the WAL in RocksDB is shown in Fig. 2. The WAL file is divided into several logical blocks, kBlockSize, the

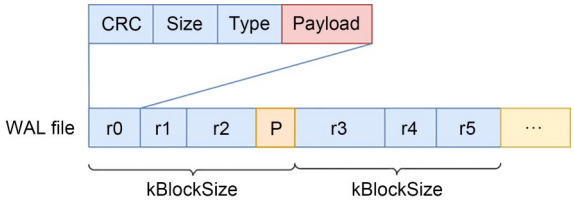

Fig. 2 Data format of a WAL file (WAL: write-ahead log; CRC: cyclic redundancy check)

basic unit of WAL reaing and writing operations, and the default size is 32 KB. Each area is composed of several variable length records (e.g., r0, r1, and r2), and the remaining part (P) is filled with zero. Each record stores the length and type of payload, including update requests and its corresponding 32-bit cyclic redundancy check (CRC) data. When an OS crashes, all data in MMT and IMMT will be lost. In this case, when RocksDB reboots, it can rebuild the MMT and IMMT by reading and performing operations saved in WAL.

2.2 KV storage based on LSM tree

Based on the RocksDB data writing process described in the previous subsection, this paper divides the system's performance overhead into three parts: (1) the problem of writing performance overhead caused by pre-writing logs when the WAL mechanism is enabled, (2) the problem of writing pause because the MMT cannot convert to the IMMT when the data are refreshed from the IMMT to the SST, and (3) the problem of writing amplification caused by the consolidation and compaction operations between SSTs at different levels on the disk.

In addition to these three areas, existing optimization directions based on LSM tree focus mainly on reducing writing amplification (Raju et al., 2017; Yao et al., 2017; Mei et al., 2018), optimizing merge operations (Zhang ZG et al., 2014; Teng et al., 2017), researching special load balancing (Wu et al., 2015; Ren et al., 2017), supporting automatic tuning (Dayan and Idreos, 2018; Zhang YM et al., 2018), designing secondary indexes (Zhu et al., 2017; Absalyamov et al., 2018; Qader et al., 2018; Luo and Carey, 2019), and making full use of the underlying hardware platform for optimization based on new hardware (Athanassoulis et al., 2011; Kannan et al., 2018; Papagiannis et al., 2018).

This subsection discusses the three proposed directions in combination with the above research work.

SpanDB (Chen et al., 2021) proposes storing the WAL and the top-level SST on the speed disk (SD), while the lower-level SST is stored on the capacity disks (CDs). It dynamically adjusts the storage location of new SSTs based on the pressure on the SD and CD. Although SpanDB optimizes the WAL mechanism and adjusts the storage of SSTs, its asynchronous interface storage engine is not suitable. If forced to be embedded in the application, it cannot exert its maximum potential. FlatLSM was also used to try to optimize the WAL mechanism by proposing a persistent memory table (PMT) that separates the index and the data. It uses buffer logs to store KV bytes with a size <256. FlatLSM attempts to address the problem of writing pauses by combining volatile memory and persistent L0, which can reduce the depth of the LSM tree and concentrate IO bandwidth on L0-L1 compaction.

Yao et al. (2020) designed MatrixKV with a new data structure called the matrix container on NVM to manage the L0 layer to reduce writing pauses. MatrixKV also includes a column action to reduce the amount of data compaction from the L0 layer to the L1 layer. To reduce the depth of the LSM tree, the width of each layer is increased, and cross-row hint search is introduced to improve the reading performance. However, its garbage collection mechanism leads to wasted persistent memory (PM) space. Also, its cross-row hint search, designed to improve the reading performance, reduces the flush speed to a certain extent.

dComparison (Pan et al., 2017) proposes delayed compaction to reduce writing amplification. It postpones some compaction to the next compaction for collection, which means it combines multiple virtual compactions into one actual compaction to reduce the total IO and improve the writing performance. Virtual compaction means that metadata are written in the form of virtual SST, and each virtual SST points to the unordered real SSTs. However, the improvement of reading–writing performance is not very good. Therefore, Pan et al. (2017) introduced voluntary counseling and testing (VCT) and virtual SST merge threshold (VSMT) parameters to ensure the reading–writing performance trade-off.

Single-level merge (SLM)-DB (Kaiyrakhmet et al., 2019) puts MMT and IMMT on the PM, and introduces B+-tree on the PM to build a global index for all SSTs. It reduces writing amplification to improve reading–writing performance and optimize reading amplification. However, as data increase, the cost of maintaining the B+-tree increases.

To summarize, we redesign the WAL mechanism in the LSM-tree structure and build it on a new storage media, such as NVMe SSD or NVM, to optimize the performance of synchronous WAL. Most research focuses on the writing amplification and writing pause problems. Only SpanDB optimizes WAL on NVMe SSD, which is highly relevant to our research. Therefore, we compare the performance of SpanDB with MyWAL in this paper.

2.3 Performance loss caused by WAL

RocksDB saves new data in the memory, achieving much higher performance than the disk-based database. However, the data in the memory can be lost if there is a power failure or OS crash without WAL. Therefore, the WAL is the essential mechanism to ensure data availability. Unfortunately, writing WAL reduces the RocksDB writing performance. In this subsection, we evaluate the performance loss caused by WAL.

db_bench is the primary benchmark to evaluate RocksDB's performance. RocksDB inherits it from LevelDB and enhances it to support many additional options. There are several workloads: fillseq, readrandom, overwrite, fillrandom, seekrandom, and readwhilewriting. We select three writing workloads (fillseq, fillrandom, and overwrite) to evaluate the performance loss caused by WAL. The pattern of writing WAL can be specified when the RocksDB starts. The default pattern is writing WAL synchronously. When the application can tolerate some data loss, it can use writing WAL asynchronously for better performance. If the user can tolerate the loss of all memory data, it can disable WAL to obtain the highest performance.

Fig. 3 shows the performance loss caused by WAL from three perspectives. Figs. 3a–3c compare the throughput under different patterns. When the WAL mechanism is disabled, the performance is highest in all three patterns. The throughput of no-WAL is 7%–200% higher than that of async-WAL and 11 to 210 times higher than that of sync-WAL. Meanwhile, the throughput of async-WAL is 10 to 73 times higher than that of sync-WAL. Figs. 3d–3f compare the executed OPS. The OPS of no-WAL is 4%–212% higher than that of async-WAL and 11.36 to 246.92 times higher than that of sync-WAL. Meanwhile, the OPS of async-WAL is 10.56 to 79.65 times higher than that of

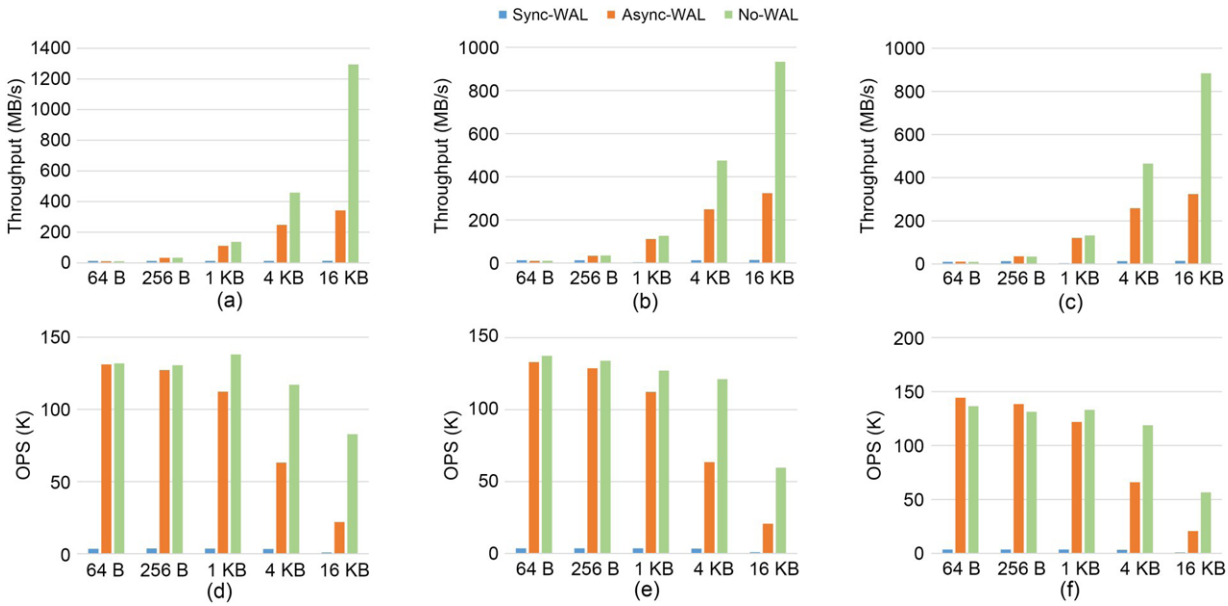

Fig. 3 Performance loss caused by WAL: (a) fillseq throughput; (b) fillrandom throughput; (c) overwrite throughput; (d) fillseq OPS; (e) fillrandom OPS; (f) overwrite OPS (WAL: write-ahead log; OPS: operations per second. The horizontal axis represents the size of each data written by db_bench)

sync-WAL. The WAL mechanism reduces the writing performance by 11 to 246 times.

The bottleneck of updating KV data is writing WAL to the file system. When the WAL is written synchronously, the throughput is <5 MB/s. We tested the sequential writing performance of the file system and raw devices using FIO (https://fio.readthedocs.io/en/latest/fio_doc.html). FIO is an IO tool used for stress/hardware verification. We compared the writing performance on serial advanced technology attachment (SATA) SSD, NVMe SSD, and NVM. We ran FIO v3.28 using the "sync" ioengine with sequential write and random write. Fig. 4 shows the results. The sequential and random writing performances of these media are almost the same. NVMe SSD is the fastest media, and the peak writing performance can reach 2 GB/s. When the data block is >4 KB, there is a significant drop in writing performance, and as the data block increases, its writing performance is not as good as NVMe SSD devices. When NVM uses the file system at 4 KB, the writing performance is reduced by about 77%, even close to that of the NVMe SSD. The throughput of RocksDB is 100 times slower than that of the file system on an SSD. It is necessary to completely abandon the general file system's data organization and management mechanism to avoid unnecessary functions brought by it, thereby eliminating the performance loss of the general file system to the storage media. Therefore, our main task is building new data organization and management mechanism on storage media based on WAL files.

3 Design and implementation

In this section, we introduce the design and implementation of MyWAL. It uses a new data structure to organize the updating log and saves data directly to the raw device. We also discuss how to recover the MMT from WAL in different cases. Finally, we discuss the thread safety of MyWAL in a multi-thread updating environment.

3.1 Reconstruction of WAL data structure

When data are written to RocksDB, RocksDB first writes to the WAL file and then updates the MMT. Fig. 5 shows the process of writing a record to the WAL file. RocksDB uses one thread to write WAL sequentially. It supports multiple threads updating KV data in parallel. Operations of multiple threads will be merged into a write_group. All operations in one write_group are merged into a MergeBatch, where the content is to be written to the WAL. If WriteOptions.sync is true, the WriteToWAL() function writes MergeBatch and flushes to the local file system synchronously. Then RocksDB updates the KV data in the

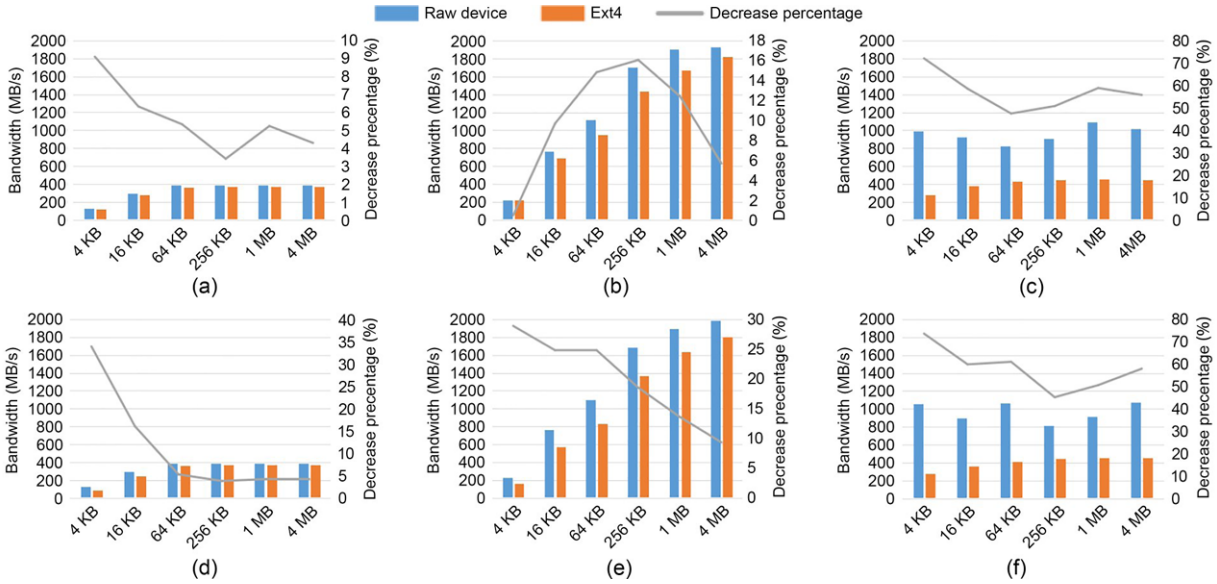

Fig. 4 Comparison of writing performance between raw devices and Ext4: (a) sequential write (SSD); (b) sequential write (NVMe SSD); (c) sequential write (NVM); (d) random write (SSD); (e) random write (NVMe SSD); (f) random write (NVM) (The horizontal axis represents the size of each data written by FIO)

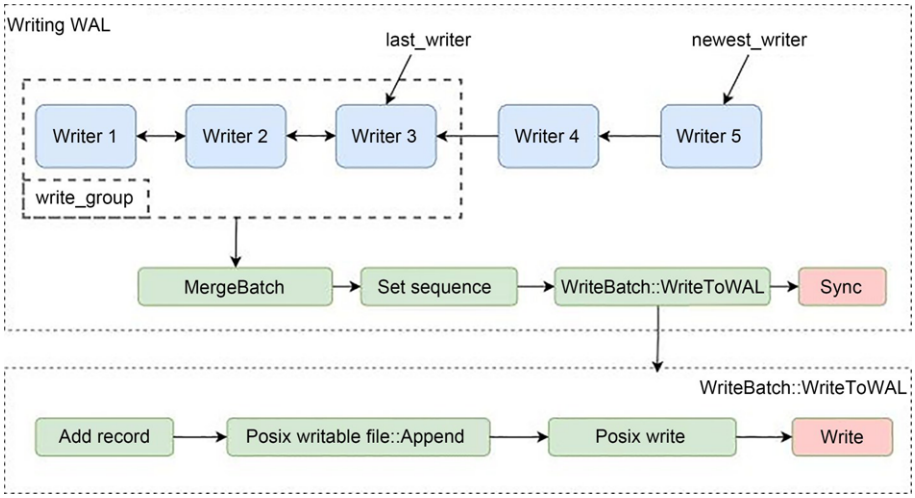

Fig. 5 Process of writing WAL

MMT. If WriteOptions.sync is false, the MergeBatch will not be synchronized to the file system until the OS flushes the buffer periodicity. In this case, the application does not wait for any disk writing IO, but some data may be lost if the system crashes. When RocksDB writes a record into WAL, the file system must allocate space and update data and metadata such as file length and access time.

Linux has a kernel buffer cache or page cache, and most disk IOs are conducted through the buffer. When RocksDB writes data to the WAL file, the kernel usually copies the data to the buffer first. Therefore, the data will be persisted only when the buffer is full or the kernel needs to reuse the buffer to store other data. This mechanism can achieve better IO performance, but may cause data loss if there is a system failure. Linux provides three functions to sync data in the buffer to disk. The sync() function moves all changed buffers to a writing queue and returns. However, it does not wait for IO completion, so it cannot guarantee that data will persist in some cases. The fsync() function saves all data related to the specific file descriptor (fd) to disk, and returns until all data and metadata are persisted. The fdatasync() function is similar to fsync(), except it only updates data and ignores metadata. RocksDB uses fsync() or fdatasync() according to the OS type and configuration. Every WAL write will change the data and file size, so fsync() will issue at least two IOs, one for data and the other for metadata. fdatasync() can perform better, but may increase the risk of data loss. If the file size does not update before RocksDB crashes, the rebuild process cannot recognize the corresponding data.

We design a new data structure to reduce the metadata update operation. We save the WAL directly to the raw device without a file system. The raw device can be a disk, a partition, or a preallocated area, which can be specified in a configuration file. We store multiple WALs in order and then maintain the metadata both in the memory and on the device. Meanwhile, the frequently updated metadata are only saved in the memory. Fig. 6 shows the data structures in the memory and device. The first data structure in the raw device is the header, which saves a magic number, the device's size (or partition's size), and the number of WAL lists. The following is the WAL list. It stores the metadata of the WAL files. Finally, the remaining area is for storing WAL data. Because there is only one writable MMT in RocksDB, there is only one corresponding writable WAL. When RocksDB starts or the size of the MMT exceeds the predefined threshold, RocksDB will create a new WAL file. We save only the start position of the WAL file. If the MMT switches to an IMMT, we store the previous WAL number in the list. The start position of the new file is the end position of the previous WAL file.

File size is an essential feature of a file. It is updated once the data are appended. We maintain the size of the WAL in the memory. If RocksDB closes normally, the contents of all WAL files will be flushed to persistent storage devices. If RocksDB terminates unexpectedly, it has to read the contents of WAL files

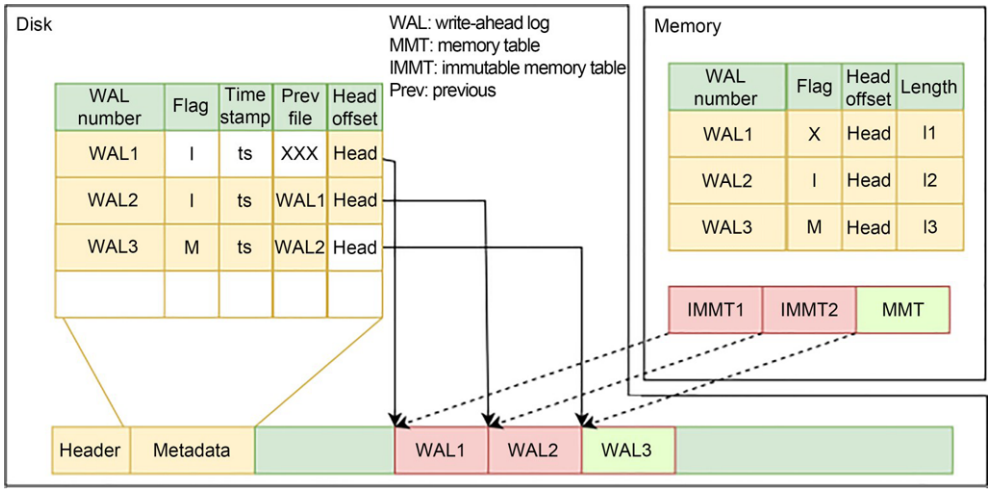

Fig. 6 Disk format and memory tables of MyWAL

to reconstruct the content in the MMT and IMMT. We save the previous log number in the WAL metadata. Because the start position of a WAL file is the end of the previous WAL file, we can construct a list according to the metadata in the WAL list. Except for the last WAL file, other files' start and end positions are known.

To find the end position of the last WAL, we modify the data format of the WAL file. Fig. 7 shows the data format of MyWAL. We add a four-byte flag in each record. We put the system time in the first record in a logical block. For each subsequent record, it saves the CRC of the previous record.

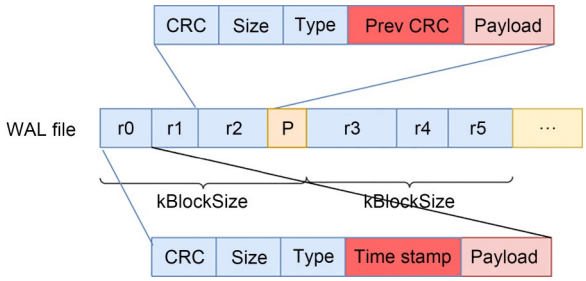

Fig. 7 Data format of a WAL file in MyWAL (WAL: write-ahead log; CRC: cyclic redundancy check)

3.2 Management of space and WAL files

When RocksDB starts, it will create a new WAL. RocksDB will create a new WAL file when the writable MMT exceeds the write_buffer_size or column family (CF) is flushed to the disk. In this case, the MMT will change to an IMMT, and the WAL will be closed and flushed to the disk. It will create a new MMT and corresponding WAL file.

The WAL file is given a log number, such as 0000001.LOG. When the size of a WAL file exceeds max_total_wal_size, it will trigger the refresh operation of the CF. Once the number of WAL files exceeds the limit, RocksDB will force all the active CF data to flush to the disk. Generally, the RocksDB will delete the corresponding WAL file after the MMT is flushed to the disk. Some options will cause RocksDB to retain the flushed WAL for a while, such as WAL_ttl_seconds and WAL_size_limit_MB. We can calculate how much space is enough for WAL files from these options. For example, if RocksDB deletes the WAL file after the MMT is flushed, the required space is max_write_buffer_number×max_total_wal_size. max_write_buffer_number defines how many MMTs are in the system. max_total_wal_size defines the maximum size of a WAL file. We can manage the space as a ring. The new WAL file is adjacent to the end of the previous WAL files. When it reaches the end of the partition, MyWAL will continue writing from the beginning. We can define several individual rings if there are several CFs.

3.2.1 Space and log files management of MyWAL

We design MyWAL to improve the performance of KV data writing by reducing metadata updates on the disk. When RocksDB creates a WAL file, MyWAL will write the metadata to the disk to record the log number, time stamp, and the previous WAL filename

(if it has one). At the same time, MyWAL maintains the metadata information of the WAL file in memory. When RocksDB updates the WAL file, the length information will update in memory but will not write to the disk. A record is the minimum updating unit of the WAL file. MyWAL adds a four-byte flag to the header of each record. The flag of the first record in the WAL file stores the same time stamp, saved in the WAL metadata. Other records save the CRC information of their previous record. With this unique design, MyWAL can calculate the end position of the WAL file by scanning the disk. When a MMT has flushed to the disk, RocksDB will delete the corresponding WAL files. For MyWAL, it deletes only the metadata in the WAL lists. In some cases, RocksDB needs to keep the WAL file for a while, and MyWAL updates the WAL flag instead of deleting the item.

Fig. 8 shows how MyWAL adds a WAL file on the device. There will be two types of free space in the device in MyWAL, generally due to the disk free space area 1 generated by the flush operation and the disk free space area 2 which has not been used during the data writing process. When RocksDB performs an insert operation, MyWAL writes a log file to disk in the form of an append. The specific operation process is as follows:

1. Determine whether the data size of the WAL log file to be written exceeds the capacity of area 2.

2. If it is less than the capacity, MyWAL directly writes the log file to the position pointed to by the tail and modifies the value of the tail without writing to the end of the data file after completion.

3. If it is greater than that, MyWAL writes part of the data W3_1 into area 2 and the rest of the data W3_2 into area 1.

4. Note that a judgment needs to be made when writing the remaining data into area 1. If the remaining data size is larger than area 1, write the remaining data W3_2 into area 1, and then modify the tail to complete a write request.

5. Otherwise, it indicates that the disk capacity is insufficient and cannot meet the storage requirements of the WAL log files in RocksDB, so insufficient disk space is returned.

3.2.2 Data recovery after RocksDB crash

WAL is used to recover lost data after the RocksDB crashes. We will discuss rebuilding the MMT from WAL files on the disk. There are two steps in the reconstruction process. First, we rebuild the WAL list and analyze which WAL files should be rebuilt to the MMT. Next, we restore the contents of each WAL in turn. The method for recovering MMTs from WAL is the same as the original RocksDB. The key innovation is how to obtain the start/end position and the length of the WAL files without updating the metadata. If we know the start and end positions of a WAL file, the rebuild process is the same as the original RocksDB.

We store the previous WAL filename (log number) in each metadata record. Because RocksDB generates the WAL files one by one, we can construct a list according to the relation among files. For each WAL file, if it has a subsequent WAL file, the start position of the subsequent WAL file is the end position of the

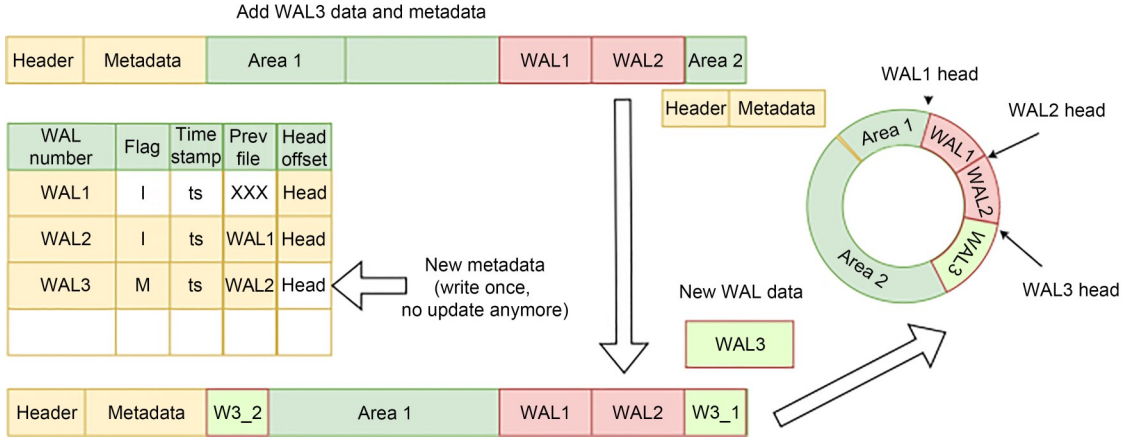

Fig. 8 Adding WAL data and metadata on the device

WAL file. The flag indicates whether the corresponding MMT has been flushed to the disk.

We do not update the file length while writing data for the newest WAL file. After RocksDB calls write() to flush data to the raw device, it cannot determine whether the data are written to the device. Both the OS and device may have caches. Using the flush instruction takes time. A write instruction may write only partial data if there is a power failure, and the atomic writing unit of SSD is 4 KB. A record is the basic writing unit for WAL files. Fig. 9 lists all eight cases in the writing process. We use a solid line to represent the completed writing operations and a dashed line to represent uncompleted writing operations. By doing the CRC of a record, MyWAL can find out if the record is unabridged or broken. For cases 1, 3, and 7, a broken record indicates that the end position is the previous record. In case 5, the data in P are not all 0, indicating that P is the end of the WAL file. Because MyWAL recycles storage space, an old record may follow the newest record in the latest WAL file. A complete record does not mean that it is a correct record. If the time stamp of r3 is earlier than the time stamp of r0, it means r3 is invalid. If the previous CRC of r1 is not the same as the CRC of r0, it means r1 is invalid. This check method is also effective for other records with a previous record.

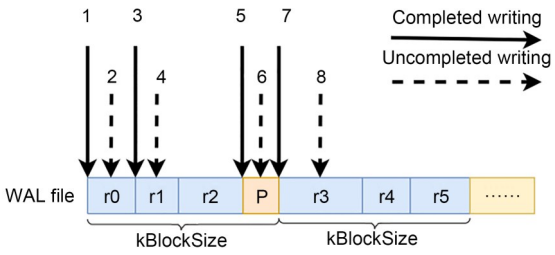

Fig. 9 Rebuilding of the newest WAL file

To verify the correctness of the recovery function, we design the following experiment: the application continuously writes KV data to RocksDB, where the keys are sequentially incremented. Concurrently, the latest KV is written into a shared memory. After a random wait of 1–20 s, we kill the application process and then read the latest KV from the shared memory to verify whether all KV data are correctly saved in RocksDB. We repeat the test 10 000 times, and each time MyWAL correctly reads all KV data.

4 Evaluation

In this section, we first compare the performance of MyWAL with the original RocksDB using db_bench, and compare the performance of MyWAL with the state-of-the-art KV store SpanDB using YCSB (Yahoo! Cloud Serving Benchmark). All experiments are run on a test machine with three kinds of media. Table 1 lists the configuration of the test server.

Table 1 Experimental configuration information

Item	Description
CPU	Intel® Xeon® Gold 5218 CPU @ 2.30 GHz
Memory	Samsung 2666 MHz DDR4 32 GB
SSD	Samsung SSD 860 EVO 250 GB
NVMe SSD	Intel® Optane™ SSD 900P 480 GB
NVM	Two Intel® Optane™ DC persistent memory 128 GB
RocksDB	RocksDB-6.28.2
Benchmark	db_bench of RocksDB-6.28.2
	FIO-3.28
	YCSB-0.17.0

db_bench is the primary tool that is used to benchmark RocksDB performance. RocksDB inherits db_bench from LevelDB and enhances it to support many additional options. Currently, db_bench supports many benchmarks to generate 32 different types of workloads, and its various options can be used to control the tests. We use db_bench to evaluate the performance of the original RocksDB and RocksDB with MyWAL. Because MyWAL optimizes the writing performance of KV data, we choose the following three workloads: (1) fillseq: write sequentially, and insert into the KV database in the order of keys. (2) overwrite: overwrite, and rewrite the corresponding value in RocksDB in random key order. (3) fillrandom: random write, and write KV data in random key order.

We evaluate the performance of RocksDB by writing hundreds of thousands of KV data with the size of value ranges from 16 B to 16 KB. We test the performance on three kinds of media: SSD, Optane NVMe SSD, and Optane NVM. By setting the wal_dir on different positions, we can test the performance of the original WAL or new MyWAL. Because the benchmark makes RocksDB write a large amount of data to the media, it may trigger garbage collection in the

following test. To avoid the effects of garbage collection, we clear the SSD and NVMe SSD before starting a new test.

4.1 Improvement of MyWAL on different devices of a single thread

Fig. 10 shows the throughput comparison results between MyWAL and the original WAL of a single thread. The results of SSD, NVMe SSD, and NVM are shown in three rows. The columns represent the results of three workloads: fillseq, fillrandom, and overwrite. For SSD, the throughput of MyWAL is approximately 4 to 7 times larger than the original WAL in all three workloads when the size of the value is 1 KB, 4 KB, and 16 KB. The throughput of MyWAL has about 1.72 to 3.80 times improvement on NVMe SSD and 1.29 to 6.13 times on the NVM.

The minimum measurable value of throughput for benchmark db_bench is 0.1 MB/s. In the first row of Fig. 10, the throughput of the original WAL is <0.1 MB/s; therefore, the results are 0.

Fig. 11 shows the OPS comparison results between MyWAL and the original WAL of a single thread. When the benchmark issues small writes, the OPS is high and the throughput is low. For SSD, the OPS of MyWAL is approximately 4.50 to 5.54 times larger than that of the original WAL in all three workloads. The OPS of MyWAL is 1.72 to 3.40 times larger than that of the original WAL on NVMe SSD and approximately 1.22 to 6.33 times on the NVM.

Ext4-DAX is a mode that enables direct access (DAX) on the Ext4 file system, allowing data to be written directly to NVM, bypassing the system buffer. It offers better IO performance than the standard Ext4 file system. However, previous research indicates that the performance of RocksDB on Ext4-DAX is similar to Ext4 (Izraelevitz et al., 2019). In this study, we also compare the performance of RocksDB to that of Ext4-DAX. The results show that the performance of MyWAL is 1 to 5 times better than that of Ext4-DAX.

4.2 Improvement of MyWAL on different devices with multi-threads

Fig. 12 shows the throughput comparison results between MyWAL and the original WAL. The results of SSD, NVMe SSD, and NVM are shown in three rows. The columns represent the results of three workloads: fillseq, fillrandom, and overwrite. For SSD, the

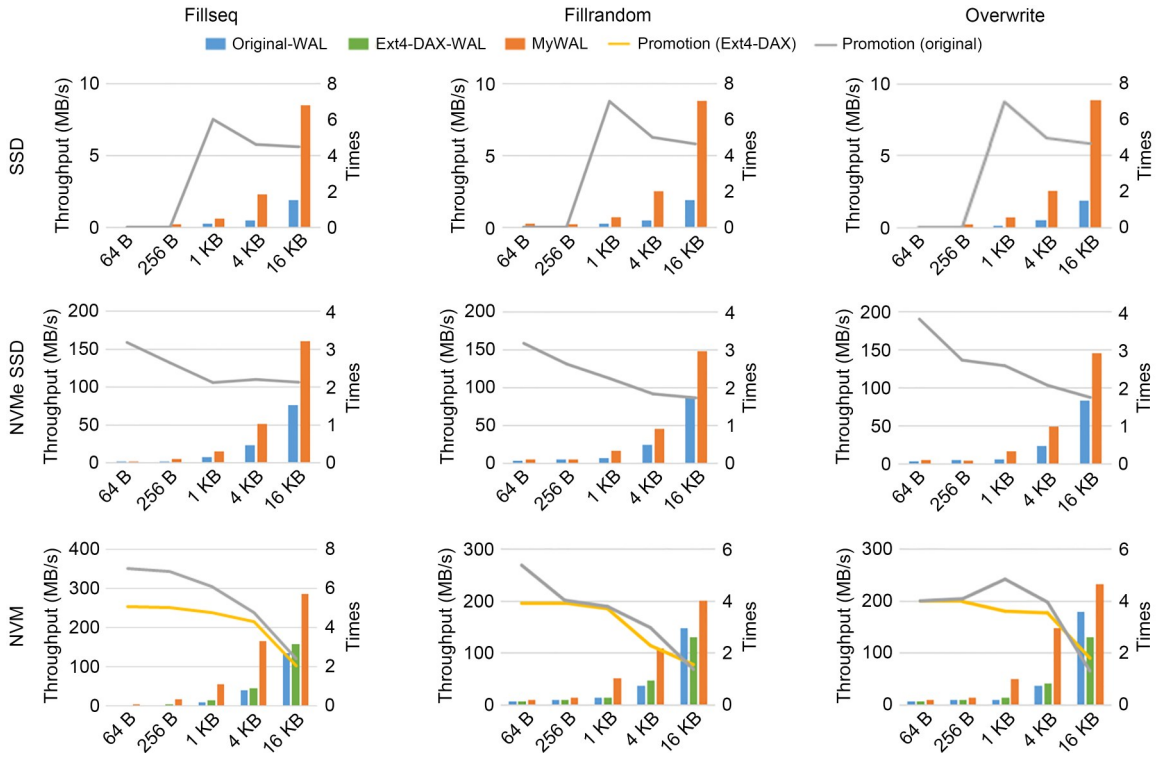

Fig. 10 Throughput comparison between MyWAL and the original WAL of a single thread (The horizontal axis represents the size of each data written by db_bench)

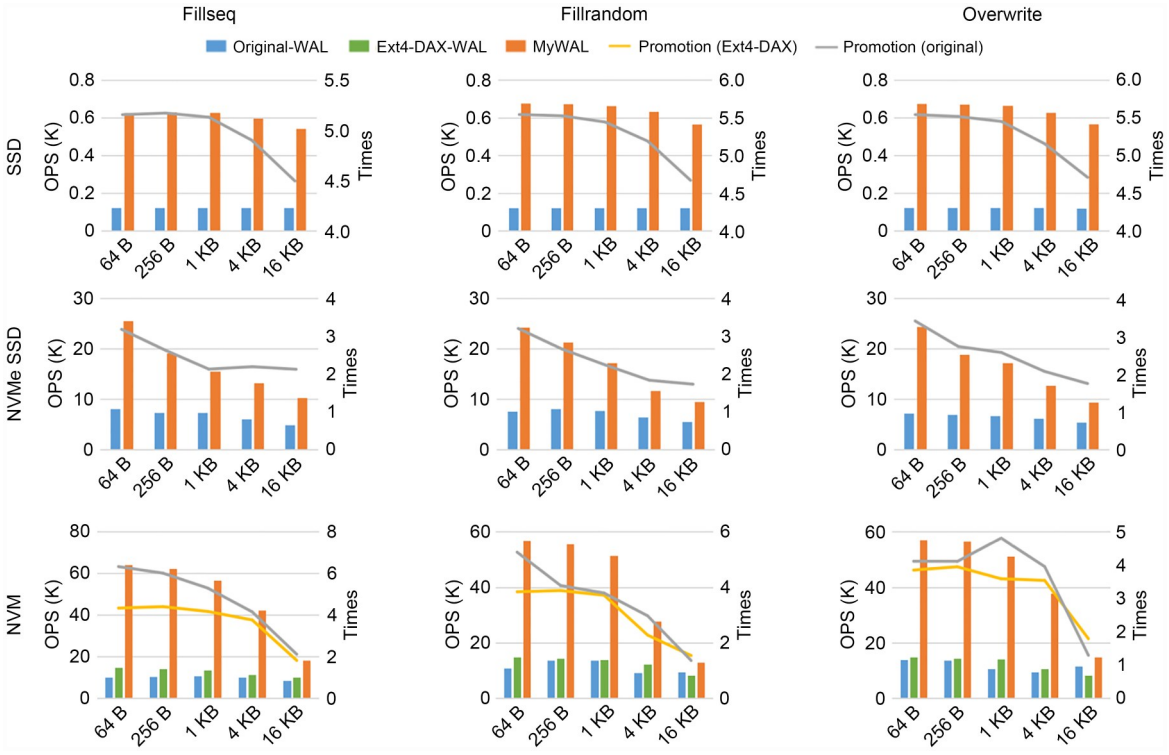

Fig. 11 OPS (operations per second) comparison between MyWAL and the original WAL of a single thread (The horizontal axis represents the size of each data written by db_bench)

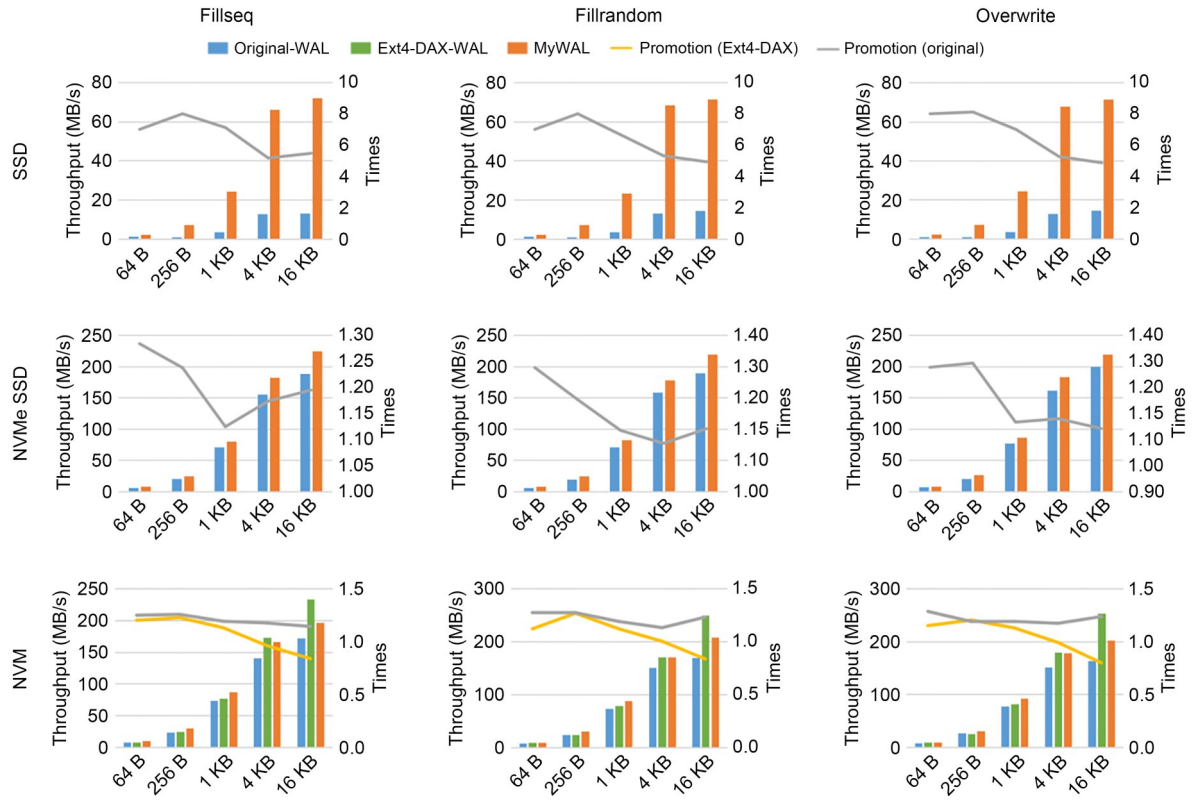

Fig. 12 Throughput comparison between MyWAL and the original WAL with 64 threads (The horizontal axis represents the size of each data written by db_bench)

throughput of MyWAL is approximately 5 to 8 times larger than that of the original WAL in all three workloads. The throughput of MyWAL has about 12%–30% improvement on NVMe SSD and 13%–29% on the NVM.

Fig. 13 shows the OPS comparison results between MyWAL and the original WAL. When the benchmark issues small writes, the OPS is high, and the throughput is low. For SSD, the OPS of MyWAL is approximately 5 to 8 times larger than that of the original WAL in all three workloads. The OPS of MyWAL is approximately 12%–30% larger than that of the original WAL on NVMe SSD and 13%–29% on the NVM. Specifically, the performance of MyWAL is about 10% better than that of Ext4-DAX.

In general, the optimization of MyWAL on SSDs is significant (5 to 8 times higher than that of the original WAL). There is also a 10%–30% performance improvement on NVMe SSDs and NVM. There are some locking and synchronization mechanisms for WAL writing in RocksDB. However, even MyWAL removes the impact of the file system, and the RocksDB cannot fully use the NVMe SSD and other high-speed devices. Furthermore, when writing WAL files to high-speed storage media, the time used for log persistence accounts for a low proportion of writing WALs in RocksDB. Therefore, the overall improvement of RocksDB is less significant than that of SSDs.

4.3 Performance comparison of YCSB

In this subsection, we use the YCSB to compare the performance of MyWAL and SpanDB. YCSB is an open-source benchmark for evaluating the performance of different "KV" and "cloud" serving stores. YCSB includes six typical workloads with different read/write patterns. Because MyWAL optimizes the writing performance, we selected write-related workloads: 100% write, YCSB-A (update heavy workload, 50/50 read/write), YCSB-B (read mostly workload, 95% read), YCSB-E (short ranges), and YCSB-F (read-modify-write). SpanDB provides high-speed parallel WAL writes via sistem pelayanan dokter keluarga (SPDK). It hosts the data on SSD and relocates WAL and the top levels of the LSM tree to the NVMe SSD. We compare the performance of SpanDB, RocksDB, and MyWAL by saving the WAL on the NVMe SSD.

Fig. 14 shows the latency of five workloads for MyWAL, SpanDB, and RocksDB. The writing latency of

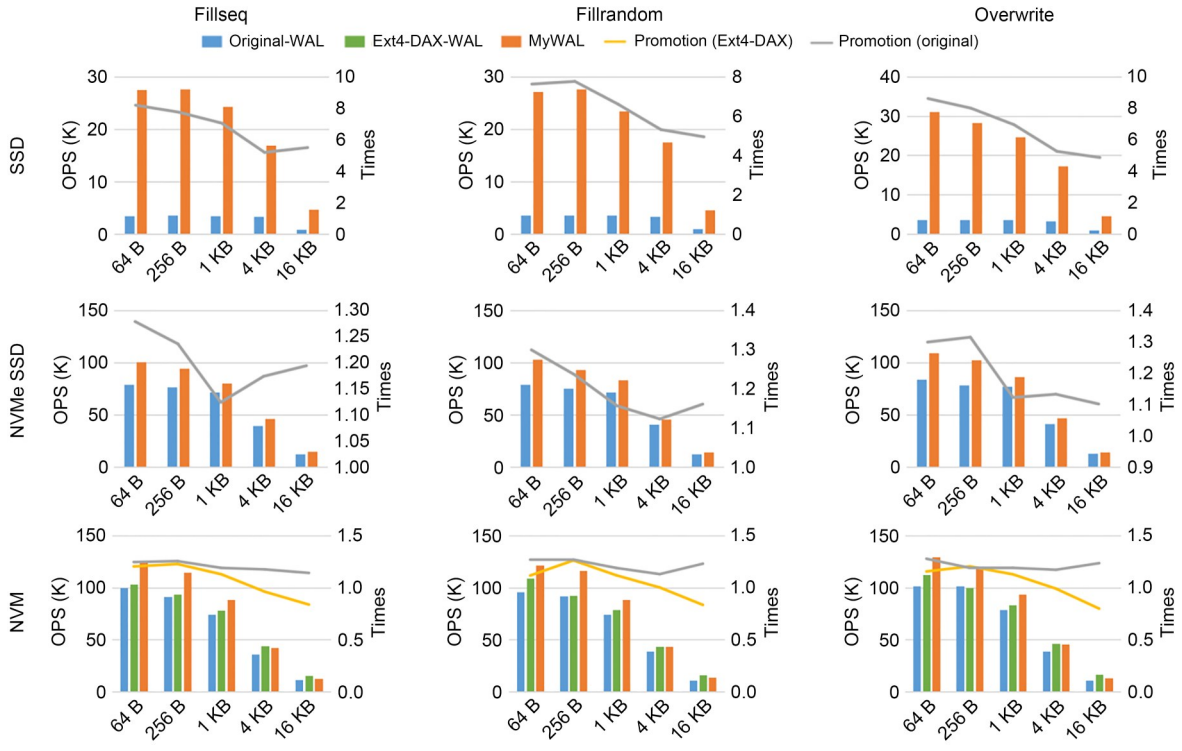

Fig. 13 OPS (operations per second) comparison between MyWAL and the original WAL with 64 threads (The horizontal axis represents the size of each data written by db_bench)

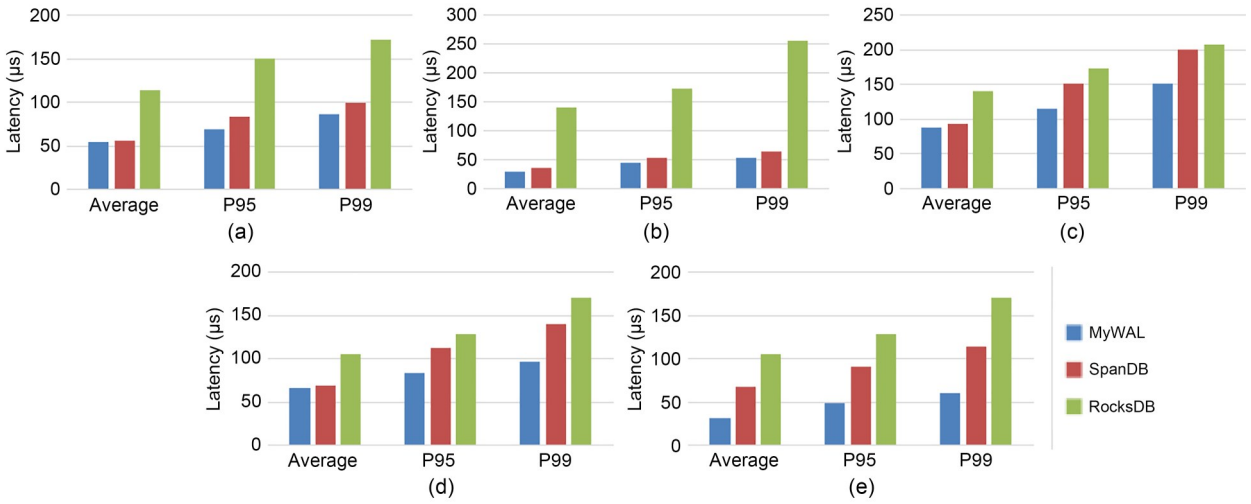

Fig. 14 Comparison of latency of MyWAL, SpanDB, and RocksDB: (a) YCSB-A; (b) YCSB-B; (c) YCSB-E; (d) YCSB-F; (e) 100% write

MyWAL is better than those of SpanDB and RocksDB, especially in the 100% write case. In the 100% write case, MyWAL has excellent performance. The average latency decreases by 50% and 70% compared to those of SpanDB and RocksDB, respectively. The P99 latency of MyWAL is also improved by 47% and 64%, respectively. In YCSB-A, the P95 latency of MyWAL is 18% lower than that of SpanDB and 54% lower than that of RocksDB. In the case of 95% and 5% write, MyWAL can still achieve good performance. In YCSB-B, MyWAL has a 16% performance improvement over SpanDB and a 79% performance improvement over RocksDB. In YCSB-E, the P95 latency of MyWAL decreases by 24% and 33% compared with those of SpanDB and RocksDB, respectively. In YCSB-F, the performance of P99 latency is 24% better than that of SpanDB and 43% better than that of RocksDB.

In general, MyWAL performs much better compared to the state-of-the-art SpanDB and RocksDB. In some heavy writing workload cases, MyWAL has excellent performance. We also test the OPS, and the OPS of MyWAL is about 20%–100% higher than that of RocksDB, but the OPS of SpanDB is much higher than those of MyWAL and RocksDB. Because SpanDB enables asynchronous request processing to mitigate inter-thread synchronization overhead and work efficiently with polling-based IO, it can execute more operations in parallel, especially reading operations.

5 Conclusions

In this work, we present MyWAL, which reconstructs the format of WAL files. It manages space and metadata directly on the raw devices. By removing the useless metadata and unnecessary update operations, MyWAL reduces the latency of writing WAL. MyWAL has excellent performance on SSD, and the results show that it is eight times faster than the original RocksDB. This optimization can be applied on different block devices, such as NVMe SSD and NVM. We also compare the performance of MyWAL with SpanDB on an NVMe SSD, and the results show that our solution has better latency than SpanDB.

Contributors

Xiao ZHANG and Mengyu LI designed the research. Mengyu LI and Yonghao CHEN processed the data. Mengyu LI drafted the paper. Xiao ZHANG helped organize the paper. Xiao ZHANG, Mengyu LI, Michael NGULUBE, Yonghao CHEN, and Yiping ZHAO revised and finalized the paper.

Compliance with ethics guidelines

Xiao ZHANG, Mengyu LI, Michael NGULUBE, Yonghao CHEN, and Yiping ZHAO declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

Absalyamov I, Carey MJ, Tsotras VJ, 2018. Lightweight cardinality estimation in LSM-based systems. Proc Int Conf on Management of Data, p.841-855. https://doi.org/10.1145/3183713.3183761

Athanassoulis M, Chen SM, Ailamaki A, et al., 2011. MaSM: efficient online updates in data warehouses. Proc ACM SIGMOD Int Conf on Management of Data, p.865-876. https://doi.org/10.1145/1989323.1989414

Chen H, Ruan CY, Li C, et al., 2021. SpanDB: a fast, cost-effective LSM-tree based KV store on hybrid storage. 19th USENIX Conf on File and Storage Technologies, p.17-32.

Dayan N, Idreos S, 2018. Dostoevsky: better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. Proc Int Conf on Management of Data, p.505-520. https://doi.org/10.1145/3183713.3196927

Dong SY, Kryczka A, Jin YQ, et al., 2021. Evolution of development priorities in key-value stores serving large-scale applications: the RocksDB experience. 19th USENIX Conf on File and Storage Technologies, p.33-49.

Facebook, 2019. RocksDB, a persistent key-value store for fast storage environments. http://rocksdb. org/ [Accessed on Jan. 7, 2021].

Izraelevitz J, Yang J, Zhang L, et al., 2019. Basic performance measurements of the Intel Optane DC persistent memory module. https://arxiv.org/abs/1903.05714

Kaiyrakhmet O, Lee S, Nam B, et al., 2019. SLM-DB: single-level key-value store with persistent memory. 17th USENIX Conf on File and Storage Technologies, p.191-205.

Kannan S, Bhat N, Gavrilovska A, et al., 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. Proc USENIX Conf on Usenix Annual Technical Conf, p.993-1005.

Leavitt N, 2010. Will NoSQL databases live up to their promise? *Computer*, 43(2):12-14. https://doi.org/10.1109/MC.2010.58

Lu LY, Pillai TS, Gopalakrishnan H, et al., 2017. WiscKey: separating keys from values in SSD-conscious storage. *ACM Trans Stor*, 13(1):5. https://doi.org/10.1145/3033273

Luo C, Carey MJ, 2019. Efficient data ingestion and query processing for LSM-based storage systems. *Proc VLDB Endow*, 12(5):531-543. https://doi.org/10.14778/3303753.3303759

Mei F, Cao Q, Jiang H, et al., 2018. SifrDB: a unified solution for write-optimized key-value stores in large datacenter. Proc ACM Symp on Cloud Computing, p.477-489. https://doi.org/10.1145/3267809.3267829

Pan FF, Yue YL, Xiong J, 2017. dCompaction: delayed compaction for the LSM-tree. *Int J Parallel Prog*, 45(6):1310-1325. https://doi.org/10.1007/s10766-016-0472-z

Papagiannis A, Saloustros G, González-Férez P, et al., 2018. An efficient memory-mapped key-value store for flash storage. Proc ACM Symp on Cloud Computing, p.490-502. https://doi.org/10.1145/3267809.3267824

Qader MA, Cheng SW, Hristidis V, 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. Proc Int Conf on Management of Data, p.551-566. https://doi.org/10.1145/3183713.3196900

Raju P, Kadekodi R, Chidambaram V, et al., 2017. PebblesDB: building key-value stores using fragmented log-structured merge trees. Proc 26th Symp on Operating Systems Principles, p.497-514. https://doi.org/10.1145/3132747.3132765

Ren K, Zheng Q, Arulraj J, et al., 2017. SlimDB: a space-efficient key-value storage engine for semi-sorted data. *Proc VLDB Endow*, 10(13):2037-2048. https://doi.org/10.14778/3151106.3151108

Stonebraker M, 2010. SQL databases v. NoSQL databases. *Commun ACM*, 53(4):10-11. https://doi.org/10.1145/1721654.1721659

Teng DJ, Guo L, Lee R, et al., 2017. LSbM-tree: re-enabling buffer caching in data management for mixed reads and writes. IEEE 37th Int Conf on Distributed Computing Systems, p.68-79. https://doi.org/10.1109/ICDCS.2017.70

Wu XB, Xu YH, Shao ZL, et al., 2015. LSM-trie: an LSM-tree-based ultra-large key-value store for small data items. USENIX Annual Technical Conf, p.71-82.

Yao T, Wan JG, Huang P, et al., 2017. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. Proc 33rd Int Conf on Massive Storage Systems and Technology, p.1-13.

Yao T, Zhang YW, Wan JG, et al., 2020. MatrixKV: reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM. Proc USENIX Conf on Usenix Annual Technical Conf, Article 2.

Zhang YM, Li YK, Guo F, et al., 2018. ElasticBF: fine-grained and elastic bloom filter towards efficient read for LSM-tree-based KV stores. Proc 10th USENIX Conf on Hot Topics in Storage and File Systems, Article 11.

Zhang ZG, Yue YL, He BS, et al., 2014. Pipelined compaction for the LSM-tree. IEEE 28th Int Parallel and Distributed Processing Symp, p.777-786. https://doi.org/10.1109/IPDPS.2014.85

Zhu YC, Zhang Z, Cai P, et al., 2017. An efficient bulk loading approach of secondary index in distributed log-structured data stores. Proc 22nd Int Conf on Database Systems for Advanced Applications, p.87-102. https://doi.org/10.1007/978-3-319-55753-3_6