FlatLSM: Write-Optimized LSM-Tree for PM-Based KV Stores

KEWEN HE, YUJIE AN, YIJING LUO, XIAOGUANG LIU, and GANG WANG,
Nankai University

19

The Log-Structured Merge Tree (LSM-Tree) is widely used in key-value (KV) stores because of its excwrite performance. But LSM-Tree-based KV stores still have the overhead of write-ahead log and write stall caused by slow L_0 flush and L_0-L_1 compaction. New byte-addressable, persistent memory (PM) devices bring an opportunity to improve the write performance of LSM-Tree. Previous studies on PM-based LSM-Tree have not fully exploited PM's "dual role" of main memory and external storage. In this article, we analyze two strategies of memtables based on PM and the reasons write stall problems occur in the first place. Inspired by the analysis result, we propose FlatLSM, a specially designed flat LSM-Tree for non-volatile memory based KV stores. First, we propose PMTable with separated index and data. The PM Log utilizes the Buffer Log to store KVs of size less than 256B. Second, to solve the write stall problem, FlatLSM merges the volatile memtables and the persistent L_0 into large PMTables, which can reduce the depth of LSM-Tree and concentrate I/O bandwidth on L_0-L_1 compaction. To mitigate write stall caused by flushing large PMTables to SSD, we propose a parallel flush/compaction algorithm based on KV separation. We implemented FlatLSM based on RocksDB and evaluated its performance on Intel's latest PM device, the Intel Optane DC PMM with the state-of-the-art PM-based LSM-Tree KV stores, FlatLSM improves the throughput 5.2× on random write workload and 2.55× on YCSB-A.

CCS Concepts: • **Information storage systems → Hierarchical storage management**; • **Hardware → Non-volatile memory**; • **Information systems → Cloud based storage;**

Additional Key Words and Phrases: Persistent memory, key-value stores, LSM-Tree

ACM Reference format:
Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. 2023. FlatLSM: Write-Optimized LSM-Tree for PM-Based KV Stores. *ACM Trans. Storage* 19, 2, Article 19 (March 2023), 26 pages.
https://doi.org/10.1145/3579855

1 INTRODUCTION

Persistent **Key-Value (KV)** stores have become the essential infrastructure in various applications, such as search engines [8], advertising systems [7], e-commerce systems [14], and photo systems [7]. In the write-intensive scenarios, the **Log-Structured Merge Tree (LSM-Tree)** [34]

Y. An is the joint first author of this article.

This work was partly supported by the NSF of China (62272253, 62272252, 62141412) and Fundamental Research Funds for the Central Universities.

Author's address: K. He, Y. An, Y. Luo, X. Liu (corresponding author), and G. Wang (corresponding author), Nankai University, No. 38, Tongyan Road, Jinnan District, Tianjin, 300350; emails: {hekw, anyj, luoyij, liuxg, wgzwp}@nbjl.nankai.edu.cn.

1553-3077/2023/03-ART19 $15.00
https://doi.org/10.1145/3579855

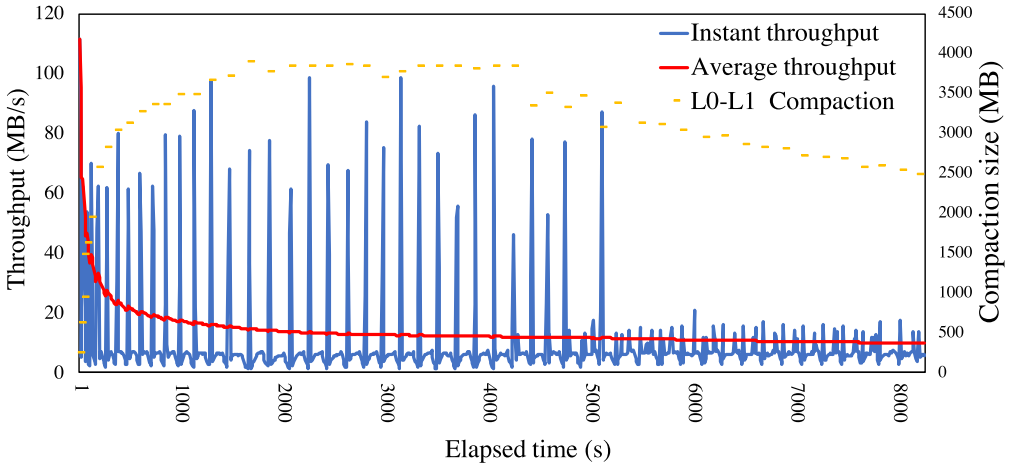


Fig. 1. Instant throughput and average throughput of RocksDB. The blue curve shows the instant throughput every 10 seconds. The red curve shows the accumulative average throughput.

is the basic index structure for KV stores. KV stores that use LSM-Tree include LevelDB [19], RocksDB [17], Cassandra [16], and HBase [3]. To obtain better write performance, LSM-Tree utilizes memtables in DRAM to handle Put/Get requests. The **Write-Ahead Log (WAL)** is also used to ensure recovery from crash. To ensure data persistence and exploiting the high sequential write performance of HDD and SSD, LSM-Tree is designed to perform flush in the background and adopt a multi-level structure on HDD/SSD. For garbage collection, the compaction of adjacent levels is performed to delete the expired KVs. The compaction operations are performed as follows. The data from immutable memtables is flushed to L_0 first, then L_0 is compacted into L_1, and so on. These strategies significantly improve write performance.

However, these designs also bring three problems. The first problem is *limited memtable capacity and heavy WAL overhead*. The size of memtables is small (128MB default) due to the limited DRAM capacity. To guarantee crash consistency, the overhead of WAL cannot be reduced.The second problem is *write stall*. Write stall will cause frequent throughput degradation and long tail latency. Figure 1 shows the random write throughput of RocksDB with 16B key size and 4KB value size. The throughput suffers periodical fluctuations and even drops to zero. The long tail latency is unbearable for KV stores. Balmau et al. [6] analyzed the cause of the write stall. They found that the root cause is that flushing to L_0 and L_0-L_1 compaction is slow so that the background flush cannot keep up with the speed of front-end write requests processing. The third problem is *write amplification*. Since LSM-Tree uses the multi-level structure on SSD, each level is 10× larger than the upper level. The read-modify-write compaction makes it necessary to rewrite lower level data, which causes the amount of data written to SSD 10× larger than the size of KVs when performing compaction in the worst case. There are many solutions to solve this problem [5, 13, 29, 31, 35, 42, 43].

Recently, persistent, byte-addressable **Persistent Memory (PM)** technologies, like PCM [36], ReRAM [39], and Intel Optane **DC Persistent Memory (DC PMM)** [11], bring much higher performance than traditional SSDs [41]. PM is a kind of **Non-Volatile Memory (NVM)**. In this article, PM and NVM refer to the same device (Intel Optane DC PMM). This provides an opportunity to solve the aforementioned problems of LSM-Tree-based KV stores. Some studies have tried this path. However, they either managed to use PM in place of (or part of) DRAM to reduce WAL overhead [24] or (partly) replace SSD with PM to alleviate the write stall problem [43]. To our best knowledge, no previous work simultaneously exploited PM's "dual role" of main memory and

external storage, which is the most important trait of PM, to optimize LSM-Tree. In this work, we try this path. First, *persistent*. WAL can be dropped to reduce the write latency with in-PM memtables, and the levels of LSM-Tree can be decreased to reduce the amount of data rewriting. Second, *fast access to PM*. The performance of PM is much higher than SSDs and close to DRAM [22]. Therefore, data in PM can be in-place updated much faster than SSDs. Third, *high capacity*. The current Intel Optane DC PMM device supports three capacities: 128GB, 256GB, and 512GB [21]. The high capacity of PM makes it possible to increase the capacity of the memtables.

Following this path, we propose FlatLSM, a DRAM-PM-SSD three-level LSM-Tree KV store that makes full use of PM. There are two major design choices of FlatLSM. First, we design PMTable with separated index and data. We also design Buffer Log and PM Log to make full use of PM because the small write (<256B) is inefficient on PM [22]. Second, to solve write stall, we flatten the structure of LSM-Tree and propose the parallel flush/compaction algorithm based on KV separation in L_0-L_1 compaction.

In summary, the contributions of this work include the following:

- We analyze the common designs of memtables based on PM and propose PMTable with index on DRAM and KVs on PM. To avoid writing small KVs (<256B) to PM directly, we design the Buffer Log and PM Log.
- We combine persistent large PMTables and unordered L_0 into a single level on PM. The first level on SSD remains ordered (we call it $Ordered - L_1$).
- We propose a parallel flush/compaction algorithm based on KV separation to reduce data rewriting and increase the throughput of flush/compaction to $Ordered - L_1$ on SSD.

2 BACKGROUND

In this section, we will provide the background of PM and LSM-Tree.

2.1 Persistent Memory

PM technologies like PCM [36] and Intel Optane DC PMM [11] have emerged recently. These devices have the properties of byte-addressable and fast random read/write like DRAM. However, they also have the ability of data persistence like HDD and SSD. Data can be accessed through memory interfaces like *memcpy()* and file systems interfaces like *read()/write()* [1]. Quite a few studies have paid attention to optimizing data structures for PM devices. For example, some studies focus on crash consistency and performance of B+trees [20, 33], hash tables [32, 44], and hybrid indexes [38] on PM. PM brings new opportunities to database engines [4].

Intel Optane DC Persistent Memory (DC PMM) is the first commercially available persistent device. DC PMM works through DIMM interfaces. Data on DC PMM is accessed through the memory bus. DC PMM's write latency is close to DRAM because of the internal write buffer. Its read latency is 3× higher than that of DRAM. DC PMM's write bandwidth and read bandwidth are respectively one-sixth and one-third of those of DRAM [22, 41]. The bandwidth can be improved linearly by increasing the number of DC PMM, making it closer to DRAM. A DC PMM file is mapped to virtual memory through *mmap()*. Data can be accessed through *memcpy()* instead of the file systems and I/O path.

2.2 Log-Structured Merge Trees

LSM-Tree is a persistent index structure proposed by O'Neil et al [34]. RocksDB [17] and LevelDB [19] are both persistent KV stores designed based on LSM-Tree. LSM-Tree is specially designed for the great sequential read and write performance of HDD and SSD, and is suitable for write-heavy workload. As shown in Figure 2, RocksDB is composed of two components: the DRAM component and the SSD component. The DRAM component is composed of two skiplists. One is

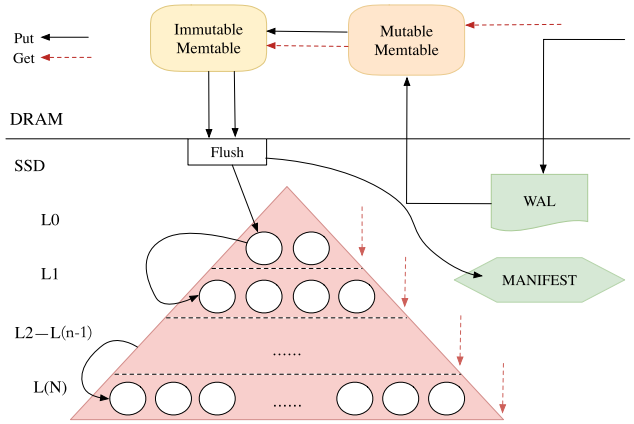


Fig. 2. The architecture of RocksDB.

a mutable memtable, which is responsible for KV insertions and in-place updates. The other is an immutable memtable, which is responsible for flushing data from DRAM to L_0 on SSD. The basic unit of KVs on SSD is the **Sorted String Table (SSTable)**. SSTables are organized into levels (L_0, L_1, L_2, …, L_n). Each level is 10× larger than the previous level. The keys in the higher levels are new, and the keys in the lower levels are old. There is WAL on persistent devices (like SSD, PM) to guarantee the crash consistency of data in DRAM.

When insertion requests arrive, they are written to WAL first, then the KVs are inserted into the mutable memtable. KVs in the memtable can be in-place updated. When the mutable memtable is full or WAL is full, the mutable memtable will be converted into an immutable memtable [32] and then be flushed to L_0 on SSD. A new mutable memtable will be created to receive new requests. To keep the same amount of memory space occupied, RocksDB maintains a fixed number of memtables. Therefore, the creation of the new mutable memtable will be blocked until the previous immutable memtable (if exists) is completely flushed to L_0. This is referred to as *write stall*.

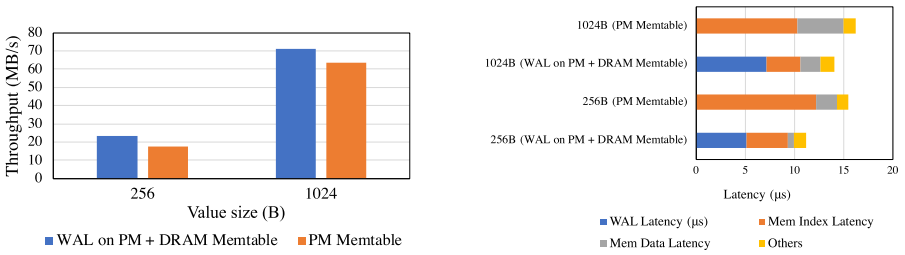
SSTables are designed as log-structure. Therefore, the key ranges of the SSTables in L_0 overlap each other. When the size of L_0 reaches the threshold, the L_0-L_1 compaction will be performed by reading the KVs from L_0 and L_1, then merging them and finally writing the result to L_1. Compactions also happen between other adjacent levels.

Manifest stores the metadata for all SSTables. The metadata includes the levels and the minimum/maximum keys of SSTables. Therefore, Manifest will be updated during flush and compaction. The search operation relies on the key ranges and levels of SSTables, which can be extracted from Manifest.

For read operations, LSM-Tree has serious read amplification problems. It has to search in turn in the mutable memtables, the immutable memtables, L_0, L_1, and so on until the target key is found or fails at the last level.

3 MOTIVATION

As shown in Figure 2 in the architecture of RocksDB, two aspects affect the write performance of LSM-Tree. One is the performance of proceeding write requests on foreground threads, dominated by WAL and memtables. The other is flush and compactions on background threads. The slow flush and compactions cause the write stall problems. Write stall will make the latency of proceeding write requests increase. This section will analyze the performance of memtables based on PM and the root cause for write stall.



(a) Throughput (b) Latency Component

Fig. 3. The throughput and latency component of PM-based memtables. These figures show the result of experiment under 16B key and 256B,1024B value random writing. WAL Latency, Mem Index Latency, Mem Data Latency, and Others in (b) represent the WAL latency, memtable index writing latency, memtable data writing latency, and latency of others, respectively.

3.1 PM-Based Memtable

There are two main strategies for designing PM-based memtables. One is to place WAL on PM to speed up the write performance of WAL and memtables on DRAM entirely. We call this approach *WAL on PM + DRAM Memtable*. This method is utilized by MatrixKV [43]. The method requires few changes to the original design and is easy to implement. Another approach is proposed by NoveLSM [24] and SLM-DB [23], which is utilizing persistent skiplists on PM to replace DRAM memtables. Since persistent skiplists can guarantee the crash consistency, there is no need for WAL. We call this approach *PM Memtable*. The simulation of NoveLSM on PM [24] shows that PM Memtable has better write performance than WAL on PM + DRAM Memtable. Figure 3 shows the throughput and latency breakdown of 4GB WAL on PM + DRAM Memtable and PM Memtable with 16B key and 256B, 1024B value insertions randomly on the Intel Optane DC PMM device. Figure 3(a) shows that compared to WAL on PM + DRAM Memtable, the throughput of PM Memtable decreases by 25.32% and 10.28% at the value size of 256B and 1024B, respectively. This result on the real device is opposed to simulation. As Figure 3(b) shows, the mem index latency of PM Memtable is 1.36× and 0.44× higher than that of WAL on PM for value size of 256B and 1024B, respectively. The mem index latency of PM Memtable accounts for 78.77%/63.38% of the overall latency. Following the results of this experiment, it is the slow PM Memtable Index Insertion that makes the PM Memtable approach slower than the WAL on PM + DRAM Memtable approach. The reason for the poor performance of PM Memtable index insertion is the poor performance of Intel Optane DC PMM on small granularity writes. When the write granularity is greater than 256B, Intel Optane DC PMM obtains high write performance [22]. But the insertions of the skiplist index are generally small granularity writes (<256B), which makes the write performance of the skiplist index on PM poor.

In summary, a slow skiplist index on PM makes PM Memtable worse than WAL on PM + DRAM memtables. The design of PM-based memtables should eliminate WAL and reduce small skiplist index insertions on PM.

3.2 Write Stall

The write stall problem affects the throughput of LSM-Tree and causes long tail latency. As shown in Figure 1, 80GB random write requests are fed to RocksDB in this experiment. The key size is 16B, and the value size is 4096B. The instant throughput shows periodic changes. The troughs of the throughput curve imply the occurrences of write stall. The cause of write stall is that L_0 is full and hence makes immutable memtables not able to be flushed to L_0. The cause of the full L_0 is the slow

L_0-L_1 compaction. As Figure 1 shows, L_0-L_1 compaction occurs intermittently. According to the yellow line in Figure 1, the instant throughput returns to peak immediately after the compaction completes. The data size of each L_0-L_1 compaction stays between 2GB and 3GB (except in the beginning). The first reason for the slow L_0-L_1 compaction is that two I/O operations are needed to write data to L_1: immutable memtables→ L_0 and L_0-L_1 compaction. The two I/Os make SSTables in $L1$ totally ordered but also introduce two disadvantages: (1) immutable memtables→ L_0 flush competes for the bandwidth with L_0-L_1 compaction, and (2) since L_0 is on SSD, L_0-L_1 compaction has to read data from the slow SSD for compaction. The second reason is that it is difficult to parallelize L_0-L_1 compaction. Since the key ranges of SSTables overlap, compaction cannot be performed in parallel directly. Otherwise, L_1 cannot be ordered totally. Single-threaded compaction makes it hard to utilize the full bandwidth of a persistent device like SSD. The third reason is the large amount of rewritten data. To keep L_1 in order, L_0-L_1 compaction is based on read-modify-write, which leads to rewriting KVs on L_1. If you increase the size of L_0 and L_1, the problem of rewritten data volume become more serious.

In summary, the structure of L_0-L_1 in LSM-Tree is the crucial reason for the write stall problem. We need to propose a method to speed up L_0 flush and L_0-L_1 compaction by reducing the redundant hierarchy, paralleling and reducing the amount of data rewrite.

4 DESIGN

To improve the write performance of memtables and solve the write stall problem, FlatLSM follows three design principles:

- Separate the index and data and eliminate WAL. To reduce small writes to PM, the index should be stored in DRAM and the data should be stored in PM. Moreover, less than 256B KVs should be flushed to PM in batch. With the persistence of PM, WAL should be removed.
- Flatten the hierarchy. Because KVs on PMTable have been already persisted, L_0 should be removed to reduce the redundant I/O operations caused by L_0 flush. The first level in SSD is in order, and we call it $Ordered - L_1$. This design choice increases the width of L_1 and decreases the depth of the tree.
- Parallel flush/compaction operations. Flushing/compacting large PMTables to $Ordered - L_1$ causes the write stall problem to be more serious when L_0 is removed. To solve this problem, we design the parallel flush/compaction algorithm based on KV separation.

4.1 Overview of FlatLSM

Based on the preceding design principles, we design FlatLSM. Figure 4 shows its structure. FlatLSM consists of PMTables on DRAM-PM and SSTables with separated KVs on SSD. The PMTables include a single mutable PMTable and multiple immutable PMTables. The size of each PMTable is 2GB by default. Each PMTable is composed of the skiplist index in DRAM and the PM Log stored on PM. The capacity of $Ordered - L_1$ is 40GB, and the other level on SSD is L_2.

4.2 Design of PMTable

In this section, we describe the detailed design of PMTables from the three aspects: the structure of PMTables, the process of write and read, and crash consistency and data recovery.

4.2.1 The Structure of PMTable. PMTable is mainly composed of the index, PM Log, and Buffer Log. The three structures will be described in the following. It should be noted that the Bloom filter in PMTables is the same as RocksDB [17], and we will not discuss it in this article.

Index. There are frequent small write on skiplist and B+-Tree indexes when inserting and updating. PMTables put all indexes in DRAM. Each bottom node of the skiplist stores pointers to the

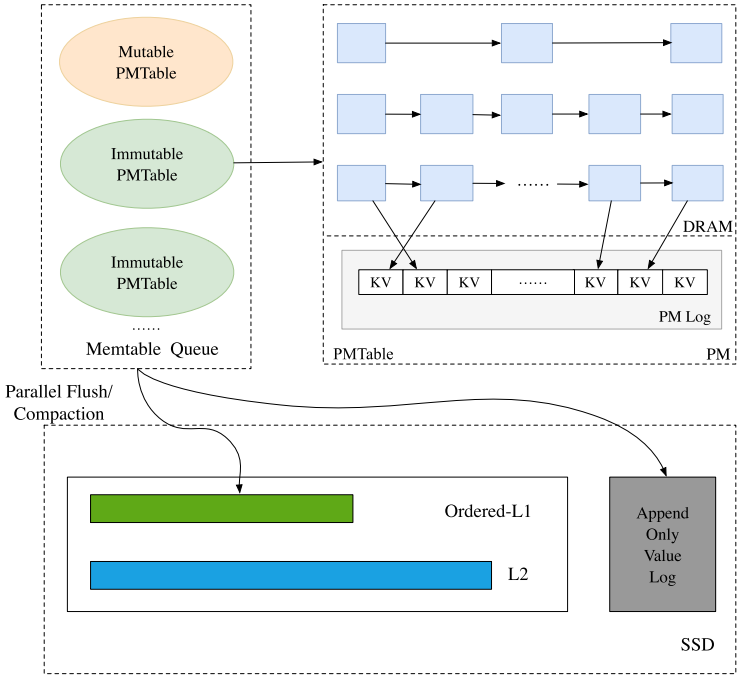


Fig. 4. The structure of FlatLSM.

KV entry in PM Log. When recovering, we need to traverse PM Log to rebuild the index. With the separation of index and data, many kinds of index structures can be used, such as skiplist and Masstree [30]. FlatLSM uses the former one.

To reduce small write operations to PM, PMTable deploys two types of logs: Buffer Log on DRAM and PM Log on PM. Small KVs (<256B) are stored temporarily in Buffer Log, and large KVs are stored in PM Log. The structures of Buffer Log and PM Log are shown in Figure 5.

PM Log. To provide persistence, we design a special structure for the file header. First, a 4B `Magic Number` identifies the file and its content should be "PMEM." If we find that this field is not "PMEM" during recovery, we identify the file as being corrupted. Second, an 8B `FileSize` specifies the capacity of PM Log that is initialized when PM Log is created. Third, an 8B `UsedSize` records the used space of KVs. `UsedSize` also implies the tail offset of PM Log during recovery. `UsedSize` is the key field for PM Log crash consistency, which is described in detail in Section 4.2.3. Finally, variable-sized log entries are appended.

Buffer Log. In addition to the mismatch between the small granularity of index update and the optimal writing granularity of PM, writing small KVs (<256B) is also detrimental to PM performance. Therefore, if a KV is less than 256B, it will be written to Buffer Log in DRAM first. The KVs in Buffer Log will be flushed to PM when the accumulated size is larger than 256B. We use three volatile pointers to record the necessary locations: `Buf_Start` records the beginning of Buffer Log, `Flush_Ptr` stores the beginning of the area that should be flushed to PM Log, and `Alloc_Ptr` specifies the beginning of the free space in Buffer Log. When we append log entries to Buffer Log, `Alloc_Ptr` moves forward. Unlike PM Log, generally a relatively small Buffer Log is enough; therefore, in FlatLSM, Buffer Log will be dynamically allocated in blocks of 256MB.

Log Entry. Each Log Entry stores a KV pair. To provide persistence with small metadata, Log Entry consists of a 2B OP, a KeySize, a Key, a ValueSize, and a Value. OP means the operation

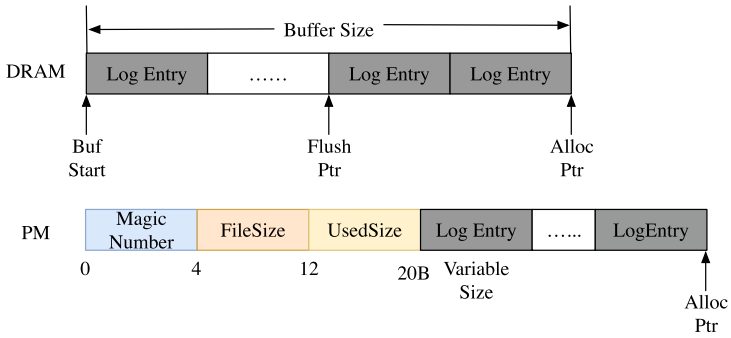


Fig. 5. Buffer Log and PM Log.

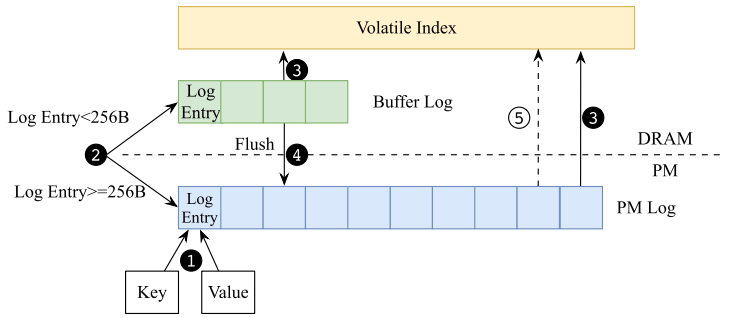


Fig. 6. The process of writing PMTable.

performed on the KV; 0 stands for insertion, 1 stands for update, and 2 stands for deletion. The role of OP is to determine whether the corresponding KV needs to be retained or ignored during recovery. To reduce the space used by KeySize and ValueSize, they are encoded into integers of variable length. The most significant bit of each byte in the coded word indicates whether the coded word is ended, and 0 indicates that current coded word is ended while 1 indicates that the next byte still belong to current coded word. Once KeySize and ValueSize are correctly decoded, the variable-sized Key and Value can be read out.

4.2.2 Writing to and Reading from PMTable. Figure 6 shows the process of writing PMTable. There are four steps required, and step 5 is optional. The first step is *Compose Log Entry*. To reduce the number of copy operations in PM Log, we pack Log Entries in pre-allocated DRAM buffers. The second step is *Write Log*. In this step, Log Entries will be written to Buffer Log and PM Log, respectively, according to their sizes. When Log Entry is larger than 256B, it is written to PM Log, the algorithm of which is shown as Algorithm 1. Otherwise, the log entry is written to Buffer Log, the algorithm of which is shown as Algorithm 2. If the `UsedSize` exceeds the capacity, Buffer Log needs to be extended. *BufferLogExtend()* does this by allocating a new 256MB block and then copying the current Log Entries between `flush_ptr` and `alloc_ptr` to the new block. Then the new incoming Log Entries will be appended to the new block. The third step is *Construct the Index*. Once the data is allocated in Log, the corresponding index node should be created and inserted into the skiplist. Since the skiplist is volatile, there is no need to consider its consistency. It should be noted that the index node points to the address of KeySize because OP does not need to be included. If the index is updated successfully, the requester will be informed and KVs can be accessed through the index. The fourth step is *Flush Buffer Log to PM Log*. The process of Buffer Log flush is shown

ALGORITHM 1: Write to PM Log

Require:
`SourcePtr`: The address of Log Entry
`Size`: The size of Log Entry
Ensure:
`RetPtr` return address of PM Log
1: **function** PMLog_AllocAndCopy(`SourcePtr`, `Size`)
2: `RetPtr` ← `AllocPtr`
3: **pmem_memcpy_persist**(`AllocPtr`, `SourcePtr`, `Size`)
4: `AllocPtr` ← `AllocPtr` + `Size`
5: `LogCurrSize` ← `UsedSize` + `Size`
6: **pmem_memcpy_persist**(`UsedSizePtr`, `LogCurrSizePtr`, **sizeof(LogCurrSize)**)
7: **return** `RetPtr`
8: **end function**

as Algorithm 3. Function FlushBuffer synchronously flushes the accumulated log entries between `flush_ptr` and `alloc_ptr` to PM Log when a newly arrived write request causes the accumulated size to be larger than 256B. The reason we do not adopt asynchronous flush is that the latency of flushing to PM is in an acceptable range of 200ns to 400ns due to small write batches. If an asynchronous strategy is adopted, the overhead of background threads sleeping and waking up is unbearable. The fifth step is *Update the Index* (optional). Once Buffer Log is successfully flushed to PM Log, we can modify the corresponding index node to point to the new location of the log entry in PM or not. If we choose to update the index, the usage of DRAM will be reduced but the overhead of updating is high. Otherwise, the index node still points to the old DRAM location, which decreases the delay at the expense of higher DRAM consumption. We choose the latter, as the DRAM consumption is acceptable: first, the KVs smaller than 256B are few, and second, for few KVs smaller than 256B, the DRAM consumption is the same as that in common LSM-Tree KV stores like RocksDB [17]. With this design choice, a small amount of redundant DRAM space is used, and the overhead of updating the index is reduced.

Reading from PMTable. Reading from PMTable is basically the same as reading from memtables. They both need to search the KVs through the volatile index. Since PMTables build the index after writing KVs to PM Log, it has no effect on the concurrent scenario of one writer and multiple readers.

Garbage Collection in PMTable. Since both PM Log and Buffer Log are append-only, garbage collection is of great necessity. The update frequency is the key point to be considered in the design of garbage collection strategy. If keys are updated frequently, the size of PM Log (called *PM_Log_Size*) will be much larger than the actual size of KVs (called *Total_KV_Size*) because the log is append-only. We believe that $\frac{\texttt{PM\_Log\_Size}}{\texttt{Total\_KV\_Size}} < 8$ indicates that updates are not frequent. Therefore, after PMTable is flushed to SSD, the space of PM Log and Buffer Log will be recycled. Duplicate keys from different PMTables are recycled by compactions between SSTables. When $\frac{\texttt{PM\_Log\_Size}}{\texttt{Total\_KV\_Size}} > 8$, we believe that there are frequent updates. Then we read all KVs from PMTable and write them to the new PM Log and Buffer Log, then update the index. The space of old PM Log and Buffer Log will be freed. Asynchronous collection can also be considered to reduce the latency caused by garbage collection.

4.2.3 Crash Consistency and Data Recovery. In this section, we discuss crash consistency and data recovery of PMTable.

Crash Consistency. Since the main persistent component in PMTable is PM Log, crash consistency will be discussed first, and the crash consistency after adding Buffer Log will be discussed

ALGORITHM 2: Write to Buffer Log

Require:
SourcePtr: The address of Log Entry
Size: The size of Log Entry
Ensure:
RetPtr The address of Log Entry in PM Log

```
1: function BufferLog_AllocAndCopy(SourcePtr, Size)
2:     if BufTotalUsed + Size > BufCurrCapacity then
3:         BufferLogExtend()
4:     end if
5:     RetPtr ← AllocPtr
6:     memcpy(AllocPtr, SourcePtr, Size)
7:     AllocPtr ← AllocPtr + Size
8:     BufUsedSize ← BufUsedSize + Size
9:     BufTotalUsed ← BufTotalUsed + Size
10:    if BufUsedSize >= 256 then
11:        FlushBuffer(FlushPtr, AllocPtr, BufUsedSize)
12:    end if
13:    return RetPtr
14: end function
```

ALGORITHM 3: Buffer Log Flush into PM Log

Require:
FlushPtr: The flush beginning of FlushPtr
AllocPtr: The address of Log Entry in Buffer Log
Ensure:

```
1: function FlushBuffer(FlushPtr, AllocPtr, BufUsedSize)
2:     FlushSize ← AllocPtr − FlushPtr
3:     PMLog_AllocAndCopy(FlushPtr, FlushSize)
4:     FlushPtr ← AllocPtr
5:     BufUsedSize ← 0
6: end function
```

later. The crash consistency of PM Log needs to be explained in terms of persistence and consistency. The persistence of PM Log is guaranteed by *pmem_memcpy_persist()* of PMDK [12]. The `NT Store` and `mfence` instructions in *pmem_memcpy_persist()* ensure the persistence. When writing KVs and modifying `UsedSize` in PM Log, we employ *pmem_memcpy_persist()* to ensure crash consistency. The consistency of PM Log is mainly supported by the atomic modification of `UsedSize`. For PM Log, the entire KV is considered successfully written to PM Log only after `UsedSize` is modified, because the `UsedSize` determines the total amount of KV pairs that need to be recovered after crash. Since the size of `UsedSize` is 8B, the atomic modification of `UsedSize` can be guaranteed. Thus, PMTable has the characteristics of crash consistency. When Buffer Log is added, KVs are persistent only if they are accumulated into batches larger than 256B and then flushed to PM Log.Although the crash consistency is reduced after Buffer Log is added, 256B batches are still much smaller than 4KB page cache blocks. Therefore, the persistence of Buffer Log is still better than that of WAL in RocksDB [17]. Moreover, to solve the problem of long batching time, we force flushing every second.

Data Recovery. During recovery, we need to traverse PM Log to read KVs one by one. The end condition for recovery is `currSize == usedSize`, which means that KVs are successfully written to

PM Log only after `usedSize` is modified. KVs exceeding `usedSize` are considered failed. Moreover, since PM is byte addressable, it is not necessary to copy KVs from a PM Log to a new one during recovery. We only need to create a new index and link the index to the address of Log Entries in the old PM Log.

4.3 The Flat Constructure of FlatLSM

In this section, we illustrate the flat design of FlatLSM from the perspectives of the size and number of PMTable and the low-level design, respectively.

The Size and Number of PMTable. The size of original memtables is limited by WAL and DRAM. However, with the help of migrating tables to PM, the size of each PMTable can be increased. The default size of each PMTable is 2GB. Because we merge L_0 into PMTable level, the design of one mutable PMTable and multiple immutable PMTables is used. Immutable memtables will not flush to SSD until a number of them reaches the threshold `flush_immutable_num`. The default value of `flush_immutable_num` is 4. Multiple immutable PMTables can delay the flushing and improve read performance. The parameter `max_write_buffer_number` denotes the maximum amount of PMTables. When the total number of PMTables reaches `max_write_buffer_number`, the write requests will be blocked. To prevent frequent blocking and wake-up, `max_write_buffer_number` is usually set to be `flush_immutable_num` + 2 (an empty buffer will be reserved). Once immutable PMTables are flushed to SSTables, Manifest will be modified.

Low-Level Design. Too many levels of LSM-Tree will make the compactions between the deep levels preempt the I/O bandwidth of L_0 flush and L_0-L_1 compaction, which causes write stall. With the newly designed PMTables, the extra copy of L_0 is reduced, and the width of each level can be increased. The design of FlatLSM only retains two levels, and each level is widened. To provide better read performance, the first level is ordered. Through the flat two-level design, the I/O bandwidth can be concentrated on the process of flushing PMTable to $Ordered - L_1$. Additionally, write amplification can be effectively reduced with this flat design.

However, flushing large PMTables to $Ordered - L_1$ still causes write stall because of the large amount and serial processing of KV rewriting. To solve this problem, we design a parallel flush/-compaction algorithm based on KV separation, which will be explained in the next section.

4.4 Parallel Flush/Compaction Based on KV Separation

To solve the problem of write stall caused by flushing large PMTables into $Ordered - L_1$, we propose a parallel flush/compaction algorithm based on KV separation. This algorithm is different from the traditional KV separation method used in WiscKey [29]. It solves the problem that L_0-L_1 compaction cannot be performed through intra-memtable parallelism. Moreover, the strategy of separating KVs reduces the amount of values rewritten brought by read-modify-write. This section describes two algorithms: parallel flush and parallel compaction based on KV separation.

4.4.1 Parallel Flush. When $Ordered - L_1$ is empty, parallel flush based on KV separation will be executed. Algorithm 4 shows the details of parallel flush, and Figure 7 shows an example. The main steps of parallel flush based on KV separation are as follows. The first step is to divide tasks according to the number of threads. Assuming that the total number of KVs stored in the PMTable is N and the number of threads is K, each thread is responsible for N/K KVs. The second step is to determine which KVs each thread processes. We traverse the candidate PMTable to divide the KVs into K groups. Then we determine the minimum value, the maximum value, and the size of each group to ensure that the key ranges of threads do not overlap. In the third step, threads flush their own groups in parallel. Each thread constructs an iterator to traverse its group from the minimum value to the maximum value. Each thread writes keys to SSTables and KV pairs to Value Log. In the

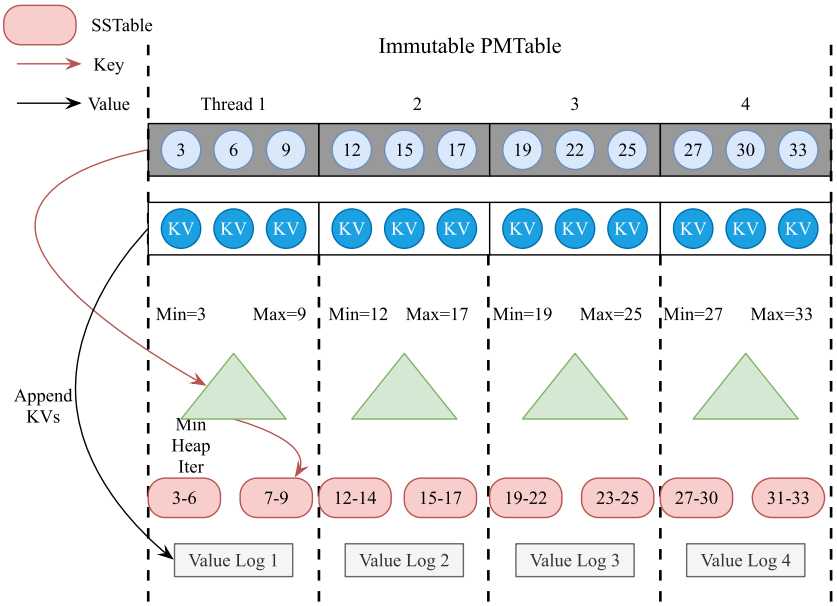


Fig. 7. Parallel flush algorithm.

ALGORITHM 4: Parallel Flush Based on KV Separation

Require:
`PMTable`: Candidate Immutable PMTable
`ThreadNum`: The Number of Thread
Ensure:

```
function ParallelFlush(PMTable, ThreadNum)
    Traverse PMTable and divide tasks according to the number of threads
    [Mem_min, Mem_max, MemIter] ← Each thread's range and iterator.
    for Each thread do in parallel
        IterMin ← Mem_min
        IterMax ← Mem_max
        MemIter.Find(IterMin)
        while MemIter.Key() <= IterMax do
            [Key, Value] ← MemIter.get()
            Let Key write into LSM-Tree to construct SSTable
            LetKey+Value append into Value Log
            Link Key of SSTables to Key+Value of Value Log
        end while
    end for
end function
```

fourth step, after all threads finish, they update the manifest and delete the immutable PMTable to release PM space. If there is any blocked thread in the foreground, wake it up.

4.4.2 Parallel Compaction. If there are SSTables in $Ordered - L_1$, parallel compaction based on KV separation will be performed. The detailed algorithm is shown in Algorithm 5. Figure 8 shows an example of the algorithm. The main steps are as follows. First, divide tasks according to the

ALGORITHM 5: Parallel Compaction Based on KV Separation

Require:
PMTable: Candidate Immutable PMTable
ThreadNum: The Number of Thread
SSTableArray: Array of all SSTables on $Ordered - L_1$
Ensure:
1: **function** Parallel Compaction(PMTable, ThreadNum, SSTableArray)
2: Traverse PMTable and divide tasks according to the number of threads
3: $[Mem_{min}, Mem_{max}, MemIter]$ ← Each thread is responsible for processing the smallest Key, the largest Key, and the KV iterator.
4: **for** Every thread parallel process **do**
5: MinHeapIter ←Create an iterator through the minimum heap
6: MinHeapIter.**Add**(MemIter)
7: **for** SSTable in SSTableArray **do**
8: $[SST_{min}, SST_{max}, SSTIter]$ ← the smallest Key, the largest Key, and the KV iterator if each SSTable
9: **if** $SST_{max} \geq Mem_{min}$ and $SST_{min} \leq Mem_{max}$ **then**
10: MinHeapIter.**Add**(SSTableIter)
11: $TotalSST_{min} \leftarrow \mathbf{min}(TotalSST_{min}, SST_{min})$
12: $TotalSST_{max} \leftarrow \mathbf{max}(TotalSST_{max}, SST_{max})$
13: **end if**
14: **end for**
15: $PreMem_{max} \leftarrow Mem_{max}$ of previous thread. If it is the first thread, the value does not exist
16: **if** First Thread **then**
17: IterMin ← $\mathbf{min}(Mem_{min}, TotalSST_{min})$
18: IterMax ← Mem_{max}
19: **else if** Last Thread **then**
20: IterMin ← $PreMem_{max} + 1$
21: IterMax ← $\mathbf{max}(Mem_{max}, TotalSST_{max})$
22: **else**
23: IterMin ← $PreMem_{max} + 1$
24: IterMax ← Mem_{max}
25: **end if**
26: MinHeapIter.**Find**(IterMin)
27: **while** MinHeapIter.**Key**() <= IterMax **do**
28: [Key, Value] ← MinHeapIter.**get**()
29: Let Key write into LSM-Tree to constructSSTable
30: **if** MinHeapIter is MemIter **then**
31: LetKey+Value append into Value Log
32: **end if**
33: Link Key of SSTables to Key+Value of Value Log
34: **end while**
35: **end for**
36: **end function**

number of threads. Second, select SSTables and determine the key range that each thread is responsible for. To prevent the key ranges of different SSTables from overlapping, this step consists of two substeps. First, determine the candidate SSTables for compaction. Since the manifest usually can be loaded into cache, the overhead of selecting SSTable is relatively small. If $SST_{max} \geq Mem_{min}$ and $SST_{min} \leq Mem_{max}$, the SSTable is a candidate SSTable. As shown in Figure 8, thread 1 needs to flush KVs with keys in the range of 2 to 5 in PMTable. The SSTables with key ranges 1 to

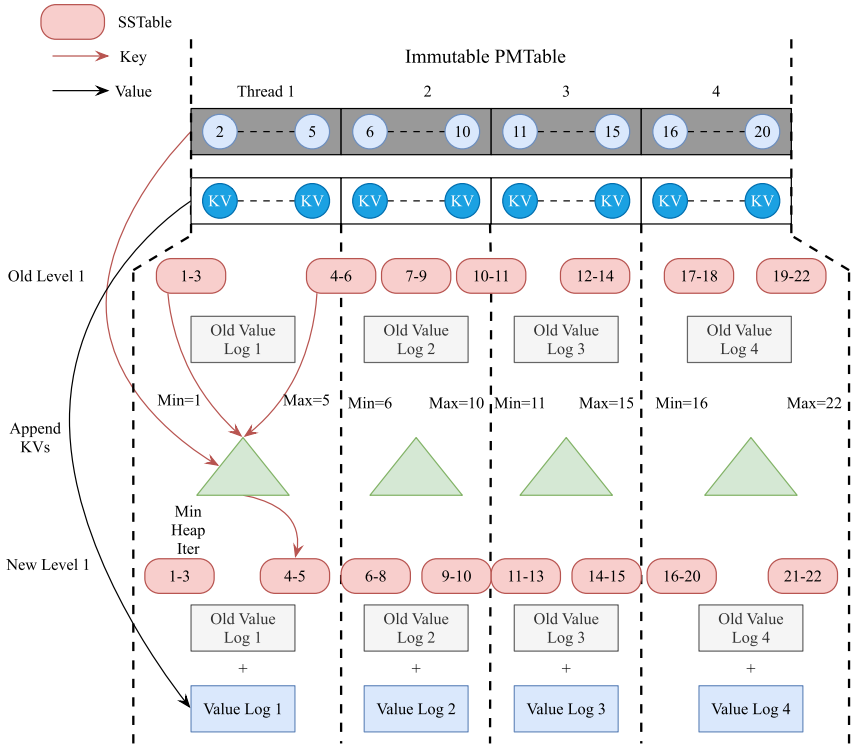


Fig. 8. Parallel compaction algorithm.

3 and 4 to 6 are in the range of 2 to 5, so the two SSTables are both candidate SSTables. Second, determine the key range that each thread is responsible for. To calculate the key ranges, it is required to obtain the minimum value $TotalSST_{min}$ and the maximum value $TotalSST_{max}$ from all candidate SSTables. Then all the working threads are divided into boundary threads and non-boundary threads. For the first thread, like the first thread in Figure 8, its key range is from min(Mem_{min}, $TotalSST_{min}$) to Mem_{max}. For the last thread, like the fourth thread in Figure 8, its key range is from $PreMem_{max}$ + 1 to max(Mem_{max}, $TotalSST_{max}$). For other non-boundary threads, their key range is from $PreMem_{max}$ + 1 to Mem_{max}. Third, all the threads write KVs in parallel. To ensure that the KVs are put into SSD orderly, each thread maintains a minimum heap composed of the PMTable iterator and all candidate SSTable iterators. During the traversal process, different types of iterator will have different operations to make. If it is a MemIter, we should write the KV to Value Log. Otherwise, if it is an SSTIter, only the key needs to be written to SSTable. Finally, the keys in SSTables are linked to the corresponding KVs in Value Log. Fourth, free the PM space consumed by PMTables.

Different Value Granularity. When values are relatively small like 128B and 256B, FlatLSM does not employ the strategy of KV separation.

Can PM Log Be Reused? In Titan [2] and DiffKV [27], there is optimization that uses WAL and Value Log. The values in SSTables are directly linked to WAL. But in FlatLSM, PM Log cannot be reused as Value Log, because the former occupies SSD space and the latter occupies limited PM space. Immutable PMTables will be written to SSD in FlatLSM, as the PM space has run out and needs to be freed up. Therefore, PM Log cannot be reused. Otherwise, foreground write requests will be blocked when most space in PM is occupied and cannot be freed up.

Garbage Collection. The garbage collection of Value Log in FlatLSM is similar to WiscKey [29].

5 EXPERIMENT

In this section, we evaluate the overall performance of FlatLSM in terms of throughput and tail latency. Then we analyze memtable performance, instant throughput, and latency breakdown to explain performance improvement of FlatLSM.

5.1 Experimental Setup

experiments, we evaluate FlatLSM on a dual-socket x86 server, which is equipped with two Intel Xeon Gold 5220 processors. Each processor runs at 2.2GHz frequency and has 18 cores with 32KB L1i cache, 32KB L1d cache, and 1MB L2 cache in each core. The shared last-level cache of each processor is 25MB. We use 128GB DRAM. The NVM devices include four Intel Optane DC PMMs in socket 0, and we run experiments on socket 0. Each PM has a size of 128GB, so the total size of NVM is 512GB with an EXT4-DAX (without page cache) file system. We use a 480GB Intel SATA SSD and a 480G NVMe SSD with the EXT4 file system. The OS is CentOS Linux release 7.8.2003 (Core).

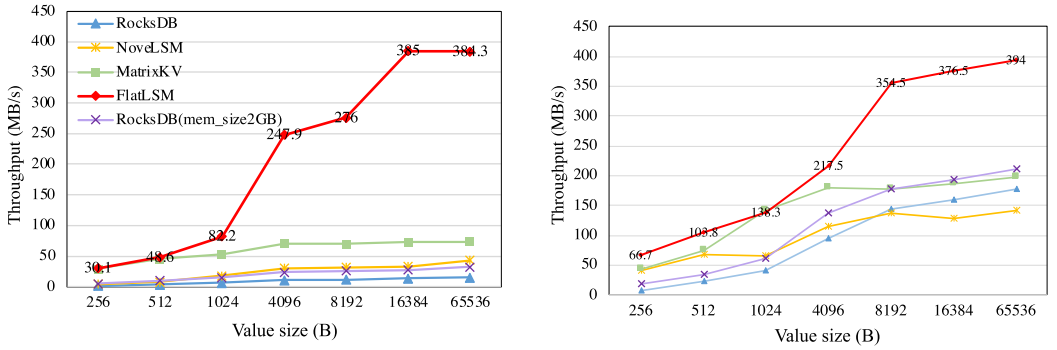
System Configuration. We use RocksDB, MatrixKV, and NoveLSM as baselines to compare with FlatLSM. The version of RocksDB is v6.11. RocksDB's WAL is on SSD and memtables are 256MB, and the other configuration is default. The PM size of NoveLSM is 12GB. The PM size of MatrixKV is 8GB. The reason for assigning 8GB PM to MatrixKV is that the write performance of MatrixKV under this configuration is better than that under other configurations. WAL and L_0 of MatrixKV are on PM. NoveLSM has two 6GB memtables on PM. The size of PMTable in FlatLSM is set to 2GB, and the max number of PMTables is 6. It will trigger flush when the number of PMTable in use reaches 4. The $Ordered - L1$ is 40GB. In our experiments, we turn off the switch *level_compaction_dynamic_level_bytes*. We will use *sync* and *echo 3 > /proc/sys/vm/drop_caches* in OS to clear the cache every minute to avoid the impact of large DRAM cache.

Dataset. We use the NoSQL YCSB [10] dataset to get YCSB-A to YCSB-F results of all workloads. The key size is 16B in the experiment, and the default value size is 4096B. We mainly evaluate the throughput, latency, and tail latency (P90, P99, and P999).

5.2 Overall Performance

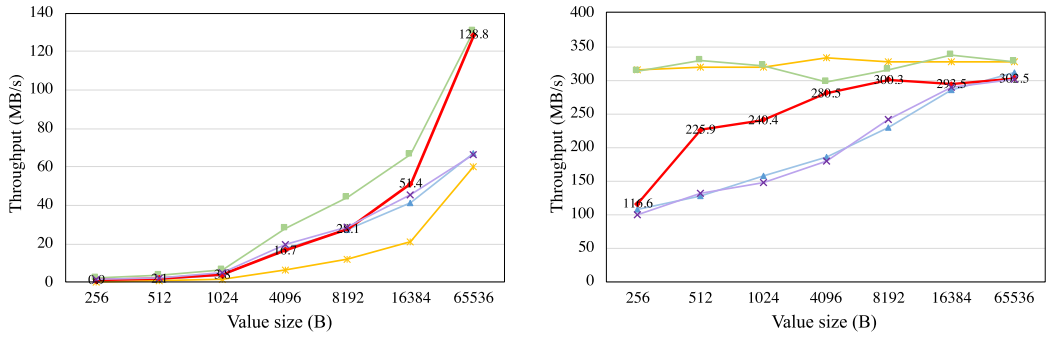
5.2.1 Micro-Benchmark. In this section, we evaluate the random/sequential read/write performance of FlatLSM, MatrixKV, NoveLSM, and RocksDB. The amount of data fed to these KV stores is 80GB. The tested value sizes are 256B, 512B, 1KB, 4KB, 8KB, 16KB, and 64KB. The test tool is *db_bench*, a benchmark tool from RocksDB [17].

Write Performance. From Figure 9, we can see that the write performance of FlatLSM is better than that of RocksDB, NoveLSM, and MatrixKV under all value sizes. Figure 9(a) shows that the random write throughput of FlatLSM is 6.54× to 27.73× that of RocksDB. When the memtable size of RocksDB is 2GB and the number of RocksDB's memtable and immutable memtable is 6 (same as PMTable of FlatLSM), the random write throughput of FlatLSM is 3.96× to 16.27× that of RocksDB. FlatLSM is still better than RocksDB. The poor performance of RocksDB might due to its low PM usage, the overhead of WAL, and a serious write stall problem. For NoveLSM, throughput increased by 3.65× to 10.6×. The write stall problem in NoveLSM is more serious due to its slow L_0 flush and L_0-L_1 compaction. The throughput of FlatLSM is 1.01× to 5.2× that of MatrixKV. This is because FlatLSM has significantly fewer write stalls and better foreground write performance than MatrixKV. It can be observed that in FlatLSM, as the value size increases from 512B to 64KB, the throughput is gradually improved from 1.61× to 12.67× compared with that under the 256B value size. However, RocksDB, NoveLSM, and MatrixKV can only achieve up to 3.04×, 5.77×, and



(a) Random Write (b) Sequential Write

Fig. 9. Write performance with different value sizes.



(a) Random Read (b) Sequential Read

Fig. 10. Read performance with different value sizes.

2.47× improvement, respectively. Considering the results mentioned previously, FlatLSM has the best granularity scalability.

Figure 9(b) illustrates the sequential write throughput. FlatLSM's advantage is less significant than that in the context of random write because write stalls and SSTable compactions are effectively reduced in the context of sequential write. Compared to MatrixKV, FlatLSM improves throughput by 1.55× to 2.06×. The improvement is mainly attributed to the introduction of PMTable, wheras the optimization on the write stall contributes little. FlatLSM still shows much better scalability than other methods.

Read Performance. Figure 10(a) shows the random read performance. The capacity of RocksDB's capacity has little impact on performance. Compared with RocksDB, NoveLSM, and MatrixKV, FlatLSM has no significant reduction in read performance. The throughput of FlatLSM is at most 35.58% lower than MatrixKV, because of the cross-row hint search method used by MatrixKV. From Figure 10(b), we can see that the sequential read performance of FlatLSM is also close to that of others.

From the micro-benchmark results, we can conclude that FlatLSM significantly increases random/sequential write performance compared with RocksDB, NoveLSM, and MatrixKV and has no obvious degradation on the performance of read.

5.2.2 Macro-Benchmark. Figure 11 shows the performance of FlatLSM, MatrixKV, NoveLSM, and RocksDB under six YCSB workloads (YCSB-A to YCSB-F). The key size is 16B, and the value

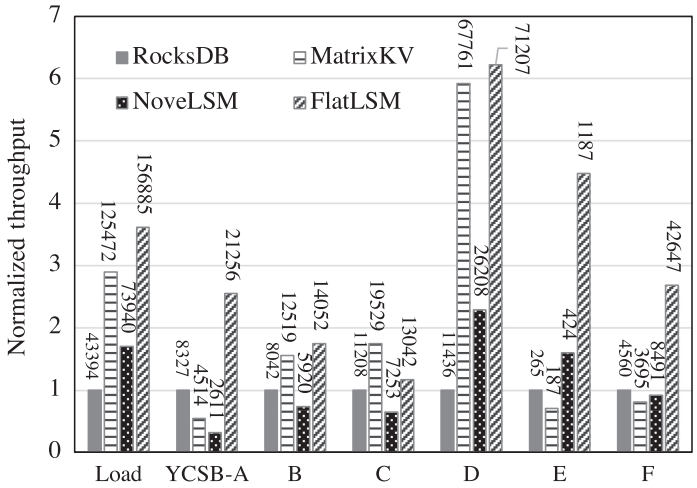


Fig. 11. YCSB performance. The *y*-axis shows the throughput normalized to RocksDB. The number above each bar is the raw value. Load means sequential write. YCSB-A is composed of 50% updates and 50% read. B contains 95% updates and 5% read. C is 100% read. D contains 95% read and 5% write (latest distributed new keys). E contains 95% range queries and 5% write. F contains 50% read and 50% read-modify-write. The key distribution is Zipf, except D.

Table 1. Random Write Latency

Latency (μs)	Avg.	P90	P99	P999
RocksDB	875.21	1,502.04	3,038.03	4,452.84
NoveLSM	1,050.07	1,818.7	3,268.97	4,259.84
MatrixKV	286.84	500.94	755.52	858.90
FlatLSM	**17.22**	**18.5**	**22.39**	**539.64**

size is 4KB. All workloads use sequential writing to load data except YCSB-E. Since YCSB-E is a workload dominated by scan query, it adopts random writing to load data. For a write-dominated workload, FlatLSM performs well. For YCSB-A and F, the performance improvement is 2.55× and 2.68× compared to the best result of baselines. For a read-dominated workload, such as YCSB-B, C, D, and F, FlatLSM still performs close to the baselines. On E, FlatLSM performs well because it is in a flat design with fewer layers. The flat design can improve the write performance by completing most scans in the same layer to significantly reduce read amplification.

5.2.3 Tail Latency. To evaluate the effect of FlatLSM on write stall, we compare the average, P90, P99, and P999 latency of random write and YCSB-A, respectively. Table 1 shows the latency of random write. We can see that FlatLSM remarkably reduces the long tail latency compared with RocksDB, NoveLSM, and MatrixKV. For P999, it reduces the average latency by 75.70%, 76.39%, and 53.77%, respectively. Table 2 shows the results of YCSB-A latency. FlatLSM still outperforms the baselines but with smaller gap. The reason for the smaller tail latency reduction under YCSB-A is that the workload of YCSB-A is so skewed that the write stalls are less than random writes.

5.2.4 The Relationship Between PMTable and Ordered-L1 Capacity and Write Performance. In Figure 12, the size of a single PMTable is 2GB. The value size used in the test is 4KB. We adjust the amount of PMTables to make the total capacity of PMTables from 8GB to 14GB. We can see that increasing the total size of PMTable will improve performance, but the improvement is slight. The

Table 2. YCSB-A Write Latency

Latency (μs)	Avg.	P90	P99	P999
RocksDB	45.88	48.8	53.14	57.54
NoveLSM	47.22	69.9	140	428.9
MatrixKV	24.12	31.4	35.8	**38.72**
FlatLSM	**11.15**	**12**	**12.7**	83.93

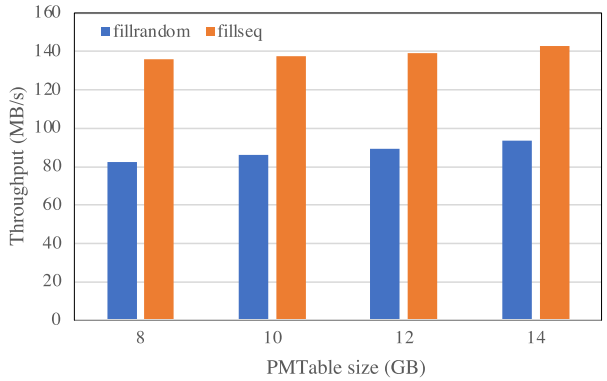


Fig. 12. Write performance with different PMTable sizes (value size = 4KB).

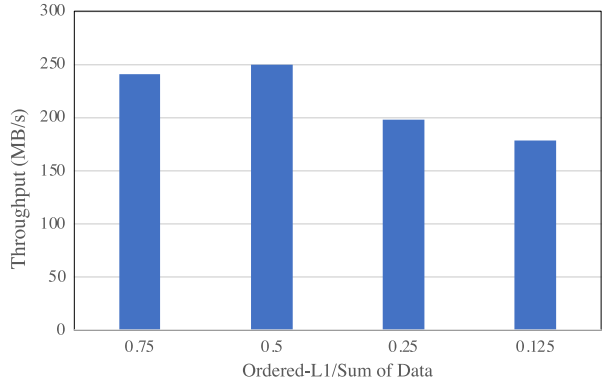


Fig. 13. Random write performance with different Ordered-L1/Sum of Data (value size = 4KB).

capacity of PMTable has little effect on write performance. Because of parallel flush/compaction, the write stall of flush between PMTable and Ordered-L1 has little impact. In the micro-benchmark experiment, the ratio is 0.5.

From Figure 13, it can be seen that when the Ordered-L1 is too small and only accounts for one-eighth of the total data, the performance degradation is slightly obvious due to the increased write amplification. When the Ordered-L1 is too large, the impact is relatively small. But the large Ordered-L1 can still cause certain blocking with more data involved in compaction.

5.2.5 Performance with NVMe SSD. We compare the performance between SSTables stored in SATA SSD and NVMe SSD. Figure 14 shows the results. Figure 14(a) and (b) show the random write and read performance, respectively. We can see that FlatLSM does not improve random write

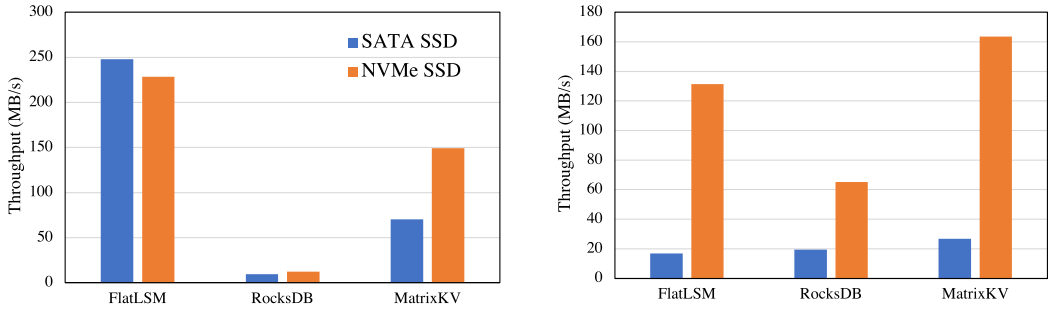


Fig. 14. Micro-benchmark of FlatLSM, RocksDB, and MatrixKV on SATA SSD and NVMe SSD.

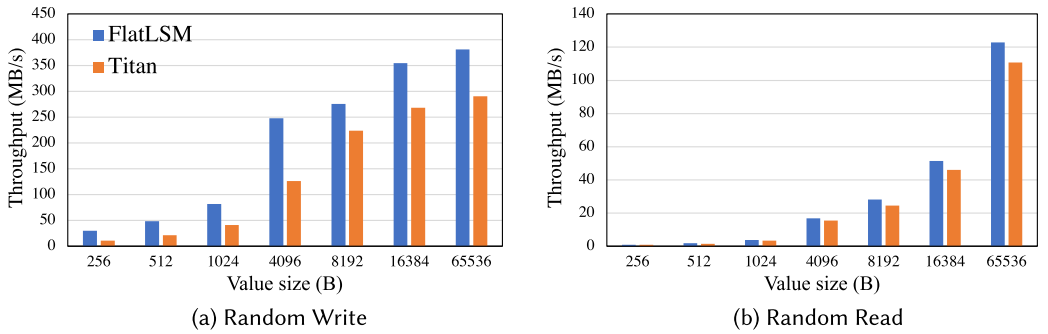


Fig. 15. Performance comparison between FlatLSM and Titan.

performance when SSTables are put in NVMe SSD because the write stall problem is solved by fast NVMe SSD. But it improves random read performance obviously because read performance of FlatLSM is dominated by SSTables. The read operation of SSTables on NVMe SSD is faster than that on SATA SSD. The faster NVMe SSD has little impact on write performance, but it causes obvious improvement on read performance. The performance improvement that NVMe SSD brings to RocksDB and MatrixKV is obvious.

5.2.6 Compared with Titan. Because FlatLSM uses a KV separation design, this section compares the performance of FlatLSM with Titan [2], a KV separation plugin designed for RocksDB. The design of Titan refers to KV separation WiscKey [29]. In Titan, the key and value pointers are stored in LSM-Tree, and value is stored in the append-only log. In this experiment, WAL of Titan is written to PM. The key size is 16B. The random write experiment involves 80GB KVs. The random read experiment is conducted by writing 80GB KVs sequentially first, then reading 100,000 KVs randomly.

Figure 15 shows the random write/read performance compared with Titan [2]. Because of PMTables, the random write throughput of FlatLSM is 3.44/2.82/2.09/1.27/1.47/1.48/1.39× higher than that of Titan as the value size increases from 256B to 64KB. The performance gap becomes narrower when the value size becomes larger. The overall performance improvement is dominated by the contribution of KV separation strategy (which benefits both methods) with large value size. For random read throughput, FlatLSM is up to 14% higher than Titan because read performance of PM is better than SSD.

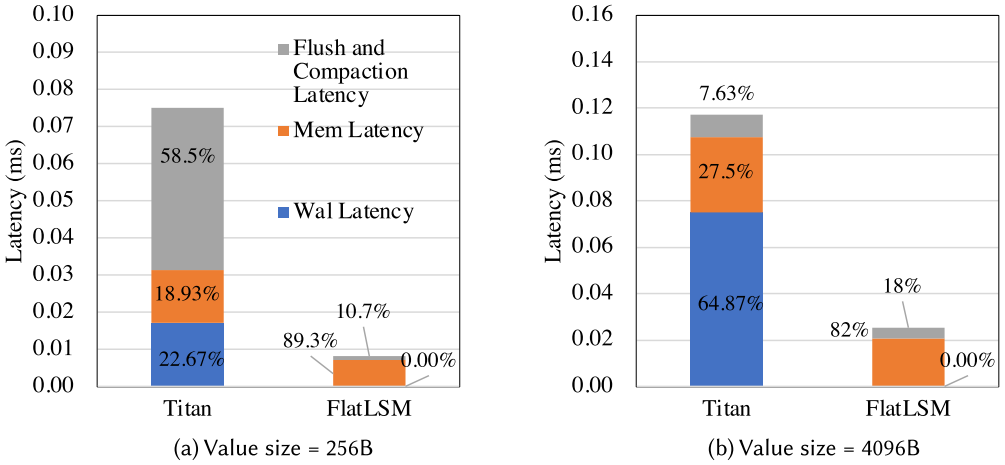


Fig. 16. Latency analysis between FlatLSM and Titan.

Figure 16 shows the latency comparison and analysis. When the value size is small, the performance of Titan is similar to RocksDB, because small values are not separated into Vlogs. And the latency of FlatLSM is much lower than Titan. When the value is relatively large, due to KV separation, the write amplification of those two is reduced, and the impact of compaction of Titan is relatively small. Additionally, because PM is used instead of memory, MEM latency of FlatLSM is higher than Titan.

5.2.7 PMTable Recovery. We evaluate two recovery strategies. The *Copy-Free* strategy means only setting pointers to log entries when rebuilding the index. The *Copy-Required* strategy means have to copying log entries themselves like RocksDB. Figure 17 shows the results. As expected, the Copy-Free strategy spends much less time on recovery compared with the Copy-Required strategy. It can be seen from the figure that both methods have a significant decrease in recovery time as the value size increases, because the larger the value size, the less the KV pairs.

5.3 Analysis of Write Performance

In this section, we mainly analyze the reasons for the write performance improvement of FlatLSM, including the performance of PMTables, instant bandwidth analysis, and delay composition analysis.

5.3.1 The Performance of PMTable. This section mainly discusses the write performance of PMTable. In this experiment, we compare three kinds of memtables. *PMTable* is the memtable with index and data separated in FlatLSM. *DRAM Memtable + PM WAL* means memtables stored entirely in DRAM and WAL in PM like (no sync) RocksDB [17] and MatrixKV [43]. *PM Memtable* means the persistent skiplist designed by NoveLSM [24]. The full memtable is stored in PM without WAL. In this experiment, the size of memtable is set to 6GB to prevent memtable performance deterioration caused by flush and compaction on background. Additionally, 4GB KVs are randomly written to each memtable. Through these strategies, it is guaranteed that all KVs will be written to memtables, not SSTables.

Figure 18 shows the throughput of three different strategies under small and large value sizes. From Figure 18(a), with small-sized values, PMTable achieves throughput 1.87/1.87/1.84/1.87/1.15× higher than DRAM Memtable + PM WAL and 4/4.53/3.89/3.75/2.0× higher than PM Memtable. In PMTable (sync mode), value in all sizes will be directly written to PM Log to ensure its persistence.

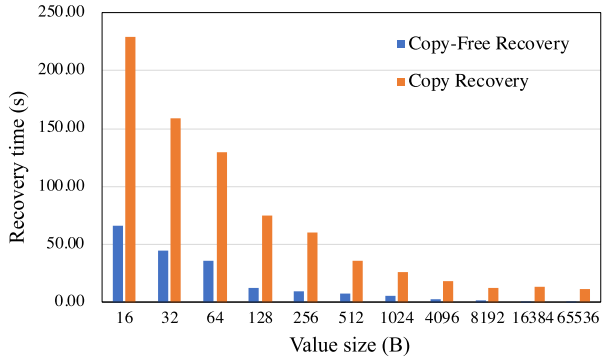
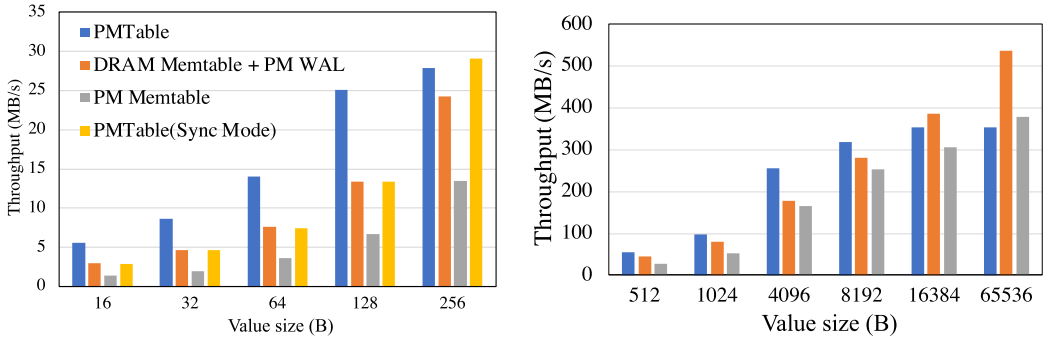


Fig. 17. PMTable recovery performance comparison. The Copy-Free strategy only needs to rebuild the skiplist index, and each node points to the address of Log Entry in PM Log. The Copy-Required strategy has to copy each entry in PM Log to the new PMTable, which is the WAL recovery method of RocksDB. This figure shows the recovery time of PMTables, which loads 4GB KVs randomly.

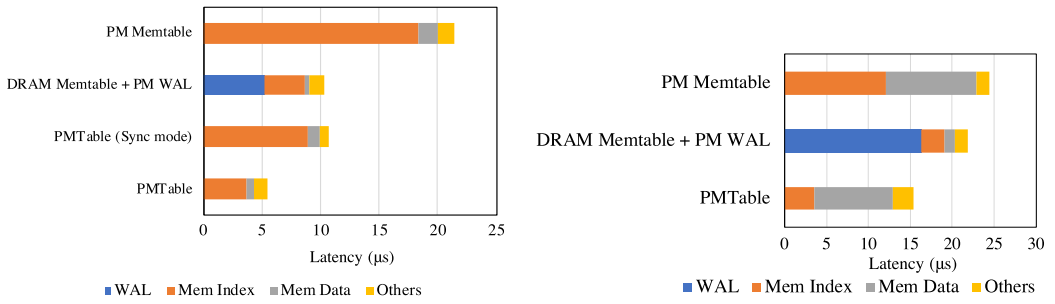


(a) 16B to 256B (b) 512B to 64KB

Fig. 18. The throughput of memtables under different value sizes. PMTable represents the memtables of the separation of index and data in FlatLSM. DRAM Memtable + PM WAL represents that memtables stored entirely in DRAM and WAL stored in PM. This scheme is used by RocksDB. PM Memtable represents a persistent skiplist designed by NoveLSM. With large-sized values, there is no need for Buffer Log. Thus, PMTable is in sync mode.

But in PMTable, only values in size that exceeds 256B will be directly written to PM Log since Buffer Log is used. When the value size is smaller than 256B, the throughput of PMTable (sync mode) will be reduced to half of PMTable (close to DRAM Memtable + PM WAL), due to the absence of Buffer Log. There are two main reasons for the high performance of PMTable. First, the separation of index and data reduces the overhead of fine-grained index updates in PM. Compared to PM Memtable, PMTable achieves better index write performance. Second, the design of Buffer Log reduces the small KV write in PM.

From Figure 18(b), it can be seen that when the value size is large, the performance improvement of PMTable is not as obvious as that of small value sizes. From 512B to 8KB, PMTable improves throughput by 22.69%, 21.91%, and 42.54% compared to DRAM Memtable + PM WAL. For 16KB and 64KB, PMTable shows a reduction of throughput by 8.25% and 34.17% compared to PM Memtable. The reason PMTable loses its advantage is that under large value size, memory and file interfaces have similar performance. Additionally, under large value size, PMTable sacrifices some



(a) Small value size (128B) (b) Large value size (4KB)

Fig. 19. Memtable latency breakdown. Key size is 16B. WAL stands for the latency of WAL. Mem Index stands for the latency of memtable index. Mem Data stands for the latency of memtable KV data.

performance to ensure the persistence, whereas PM WAL does not execute strict sync operation to ensure persistence in every insertion. Although there is write performance degradation under large value size, PMTables provide a more solid persistence guarantee than PM WAL.

From Figure 19(a), in PM Memtable, the Mem Index step takes 85.66% of overall latency. Compared with PM Memtable, the latency of PMTable is reduced mainly from Mem Index Latency. Compared with DRAM Memtable + PM WAL, the latency reduction of PMTable is mainly from WAL. The strategy of Buffer Log and PM Log only constitutes 11.52% of the WAL latency, so it does not have much impact on the overall latency of PMTable. For PMTable (sync mode), due to the poor performance of small-grained writing on PM, the latency of each part has increased. Figure 19(b) shows the latency breakdown of different memtables under large value size. The latency of PMTables is reduced by 29.63% and 36.92% compared with PM Memtable and DRAM Memtable + PM WAL, respectively. Compared with PM Memtable, the latency reduction of PMTable still comes from Mem Index. Compared with DRAM Memtable + PM WAL, the latency reduction comes from WAL. Mem Data of PMTable is 46.99% lower than WAL + Mem Data of DRAM Memtable + PM WAL. The reason is that the latency of memory access is less than file access.

5.3.2 Instant Throughput Analysis. In this section, we compare the instant throughput of FlatLSM, MatrixKV, NoveLSM, and RocksDB to explain how FlatLSM improves write performance.

Figure 20 shows that the running time of FlatLSM is 11.46% and 28.40% of NoveLSM and MatrixKV, respectively. The overall throughput of FlatLSM is large, and the number of throughput troughs caused by write pauses is small. The lowest throughput of FlatLSM is much higher than that of the baseline, which means that FlatLSM can resume immediately after the write stall. And the parallel flush/compaction algorithm based on KV separation can make the write stall problem occur as infrequently as possible.

Figure 21 shows that the overall running time of FlatLSM is only 3.83% of RocksDB, because the write amplification and write stall problems caused by RocksDB's multi-level structure are serious.

5.3.3 Latency Analysis. The average write latency is mainly composed of three parts: Write Stall Latency represents the blocking latency caused by the write stall in the foreground, Mem Latency represents the write latency of the memtable, and Wal Latency represents the write latency of WAL. Figure 22(a) shows the write latency breakdown of four methods under 256B value size. For FlatLSM, the write stall latency only accounts for 8.6% in the overall latency, meaning that FlatLSM effectively solves the write stall problem. The latency of Mem Latency in FlatLSM accounts for 91.4%, which is similar to Mem Latency + WAL Latency of MatrixKV. And both MatrixKV and FlatLSM maintain similar overall latency, which is much lower than RocksDB and NoveLSM.

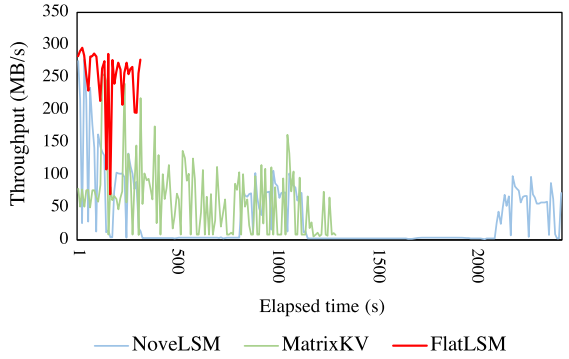


Fig. 20. Instant throughput of NoveLSM, MatrixKV, and FlatLSM. This experiment is evaluated under 16B key size, 4KB value size, and random write. The throughput is sampled every 10 seconds. The red line represents the instant throughput of FlatLSM.

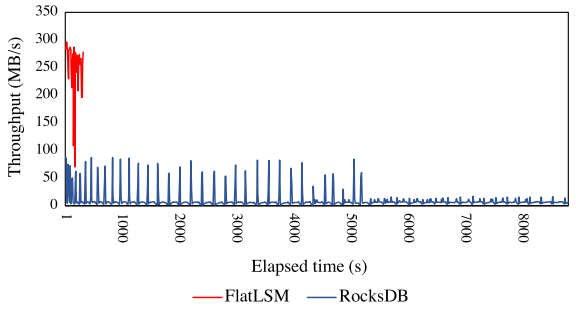
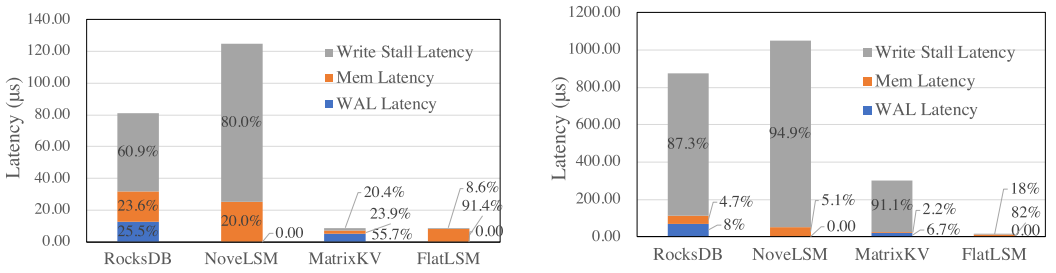


Fig. 21. Instant throughput between RocksDB and FlatLSM. The experiment is performed in the same way as in Figure 20.

Figure 22(b) shows the write latency breakdown of four methods under 4KB value size. The overall latency of FlatLSM is lower than RocksDB, NoveLSM, and MatrixKV. FlatLSM's write stall latency accounts for only 18% and is much lower than other designs.

6 RELATED WORK

KV Stores Based on LSM-Tree for NVM. Prior works like SLM-DB [23], RangeKV [28], NoveLSM [24], MyNVM [15], and MatrixKV [43] design KV stores for NVM. SLM-DB [23] utilizes PM and SSD. It uses PM memtables and immutable memtables. Its single-level structure can reduce write amplification, and global B+-Tree on PM can improve read performance. Selective compaction relies on B+-Tree. But the global B+-Tree can be detrimental to SLM-DB performance due to its costly synchronization operations. PM and NVM are the same class of devices. PM used in SLM-DB is emulated by DRAM. NoveLSM [24] utilizes NVM memtables to scale DRAM memtables. But large NVM memtables can only delay the write stall and might even harm overall performance. MatrixKV [43] utilizes the L_0 matrix container on NVM and fine-grained L_0-L_1 compaction to mitigate write stall. MyNVM [15] uses PM as a block device to reduce SSD usage and speed up throughput. NVMRocks [26] is similar to NoveLSM. It uses a persistent allocator for PM memtables and PM SSTables to decrease the usage of SSD. RangeKV [28] utilizes RangeTab in PM to increase the capacity of L_0, which reduces the number of levels and compactions. RangeTab utilizes a hash index and double buffer to reduce the number of PM accesses and the overhead of compaction. Yan et al. [40] design a PM allocator for LSM-Tree and Reorder Ring



(a) 256B value size (b) 4KB value size

Fig. 22. The latency breakdown of different methods under different value sizes. The key size is 16B. The value sizes are 256B and 4KB. Write Stall Latency represents the latency caused by the write pause in the foreground. Mem Latency represents the latency of the memtable. Wal Latency represents the latency of WAL.

to implement transactions log-free. DiffKV [27] utilizes a semi-separated structure to reduce the write amplification and improve performance of scan query.

Optimization for LSM-Tree. LSM-Tree [37] is specially optimized for small KVs. The prefix tree is used to replace the level structure to reduce write amplification. SSTables are replaced by a hash index to reduce index and metadata space cost. However, SSTables with a hash index are not conducive to scan. Silk [6] analyzes the root cause of write stall. The write stall problem is solved by increasing the write bandwidth and thread priority of L_0-L_1 compaction and L_0 flush. WiscKey [29] separates key and value and compacts only for key to reduce the amount of data for compaction. PebblesDB [35] makes fully ordered SSTables become partially ordered. It reduces write amplification by sentinel ordering. But there are still write stall problems after continuous writing. KVell [25] analyzes the problem that data ordering brings write amplification and excessive CPU consumption. KVs do not have to be ordered under high throughput of NVMe SSD. EvenDB [18] considers the spatial locality of cluster data. Many key prefixes are the same. It utilizes munks with the same key prefix to ensure the spatial locality.

KV Stores for PM. Because of persistent and byte-addressable characteristics, PM is often used to replace the traditional DRAM index structure to build KV stores. CCEH [32] redesigns a persistent hash table to guarantee crash consistency. FPTree [33] utilizes PM to store leaf nodes and DRAM to store non-leaf nodes to reduce PM fine-grained writing. It uses ordered leaf nodes to ensure crash consistency. HiKV [38] is a KV store for a hybrid index. There is a hash index with crash consistency for point query and a global B+-Tree index without crash consistency for range query. FlatStore [9] uses asynchronous group commit to batch KVs that are smaller than 256B to improve performance because larger than 256B writing is good for the Intel Optane DC PMM [22].

CONCLUSION

In this article, we proposed FlatLSM, which takes advantage of byte-addressable, persistent PM to combine memtable and L_0. We analyzed the latency composition of PM-based memtables and proposed PMTable, which separates index and data. Additionally, PMTables use PM Log and Buffer Log to reduce the number of less than 256B writing on PM. To solve the write stall problem, we flattened the structure of LSM-Tree and removed the redundant L_0. We designed $Ordered - L_1$. To solve the write stall because of PMTable flushing into $Ordered - L_1$, we proposed a parallel flush and compaction algorithm based on KV separation. FlatLSM is implemented on the real system and uses the latest NVM device (Intel Optane DC PMM) to conduct experiments. The evaluation shows that FlatLSM achieves stable throughput and low tail latency. FlatLSM achieves better write performance compared with RocksDB, NoveLSM, and MatrixKV.

REFERENCES

[1] SNIA NVM Programming Technical Working Group. 2017. *NVM Programming Model* (Version 1.2). SNIA NVM Programming Technical Working Group.

[2] 2018. Titan: A RocksDB Plugin to Reduce Write Amplification. Retrieved April 25, 2019 from https://pingcap.com/blog/titan-storage-engine-design-and-implementation.

[3] Apache. 2014. HBase. Retrieved January 30, 2023 from https://hbase.apache.org/.

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

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 363–375.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'19)*. 753–766.

[7] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 1–8.

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

[9] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.

[11] Intel Corporation. 2019. Intel Optane DC Persistent Memory Product Brief. Retrieved January 30, 2023 from https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[12] Intel Corporation. 2019. Persistent Memory Development Kit. Retrieved January 30, 2023 from https://pmem.io/pmdk/.

[13] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.

[15] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference*. 1–13.

[16] Facebook. 2008. Cassandra. Retrieved January 30, 2023 from https://cassandra.apache.org/.

[17] Facebook. 2013. RocksDB. Retrieved January 30, 2023 from https://rocksdb.org/.

[18] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: Optimizing key-value storage for spatial locality. In *Proceedings of the 15th European Conference on Computer Systems*. 1–16.

[19] Google. 2011. LevelDB. Retrieved January 30, 2023 from https://github.com/google/leveldb.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 187–200.

[21] Intel. 2019. What Is Intel Optane DC Persistent Memory? Retrieved January 30, 2023 from https://www.boston.co.uk/blog/2019/07/10/intel-optane-dc-persistant-memory.aspx.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, et al. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[23] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 191–205.

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

[25] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
[26] Jianhong Li, Andrew Pavlo, and Siying Dong. 2017. NVMRocks: RocksDB on non-volatile memory systems.
[27] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated key-value storage management for balanced IO performance. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. 673–687.
[28] Zhan Lin, Lu Kai, Zhilong Cheng, and Jiguang Wan. 2020. RangeKV: An efficient key-value store based on hybrid DRAM-NVM-SSD storage structure. *IEEE Access* 8 (2020), 154518–154529.
[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage* 13, 1 (2017), 1–28.
[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 183–196.
[31] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*. 477–489.
[32] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 31–44.
[33] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.
[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.
[35] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
[36] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Y.-C. Chen, Robert M. Shelby, Martin Salinga, et al. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.
[37] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.
[38] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 349–362.
[39] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, Los Alamitos, CA, 476–488.
[40] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the design of LSM-Tree based OLTP storage engine with persistent memory. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1872–1885.
[41] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 169–182. https://www.usenix.org/conference/fast20/presentation/yang.
[42] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. 2019. GearDB: A GC-free key-value store on HM-SMR drives with gear compaction. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 159–171.
[43] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-Tree based KV stores with matrix container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 17–31.
[44] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 461–476.

Received 26 April 2022; revised 26 September 2022; accepted 22 November 2022