Building a Fast and Efficient LSM-tree Store by Integrating Local Storage with Cloud Storage

PENG XU, WNLO, Huazhong University of Science and Technology, China
NANNAN ZHAO, Research and Development Institute of Northwestern Polytechnical University in Shenzhen, and School of Computer Science, Northwestern Polytechnical University, China
JIGUANG WAN, WNLO, Huazhong University of Science and Technology, and Shenzhen Huazhong University of Science and Technology Research Institute, China
WEI LIU and SHUNING CHEN, PingCAP, China
YUANHUI ZHOU, WNLO, Huazhong University of Science and Technology, China
HADEEL ALBAHAR, Virginia Tech and Kuwait University, USA
HANYANG LIU and LIU TANG, PingCAP, China
ZHIHU TAN, Department of Computer Science and Technology, Huazhong University of Science and Technology, China

The explosive growth of modern web-scale applications has made cost-effectiveness a primary design goal for their underlying databases. As a backbone of modern databases, LSM-tree based key–value stores (LSM store) face limited storage options. They are either designed for local storage that is relatively small, expensive, and fast or for cloud storage that offers larger capacities at reduced costs but slower. Designing an LSM store by integrating local storage with cloud storage services is a promising way to balance the cost and performance. However, such design faces challenges such as data reorganization, metadata overhead, and reliability issues. In this article, we propose RocksMash, a fast and efficient LSM store that uses local storage to store frequently accessed data and metadata while using cloud to hold the rest of the data to achieve

Extension of Conference Paper. An earlier version of this article was presented at 2021 IEEE International Conference on Cluster Computing (CLUSTER), 7–10 September, 2021 [58]. In this article, we broaden the design to allow for faster recovery. RocksMash proposes the extended WAL to trade faster recovery time for slightly more storage space. RocksMash searches WAL files in reverse chronological order to reduce data to be processed, logically reducing the size of WAL files to be processed, and thus improving scan efficiency. Evaluation results show that the proposed approach improves the recovery performance by up to 10×.

This work was sponsored in part by the National Natural Science Foundation of China under Grant No.62072196, the Science, Technology and Innovation Commission of Shenzhen Municipality (JCYJ20190808095001781), the Key Research and Development Program of Guangdong Province (2021B0101400003), the Guangdong Basic and Applied Basic Research Foundation 2021A1515110080. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Authors' addresses: P. Xu and Y. Zhou, WNLO, Huazhong University of Science and Technology, Wuhan, Hubei, China, 430074; N. Zhao (corresponding author), Research and Development Institute of Northwestern Polytechnical University in Shenzhen, and School of Computer Science, Northwestern Polytechnical University, Shenzhen, Guangdong, China, 518057; email: nannanzhao@nwpu.edu.cn; J. Wan (corresponding author), WNLO, Huazhong University of Science and Technology, and Shenzhen Huazhong University of Science and Technology Research Institute, Wuhan, Hubei, China, 430074; email: jgwan@hust.edu.cn; W. Liu, S. Chen, H. Liu, and L. Tang, PingCAP, China; H. Albahar, Virginia Tech and Kuwait University, Blacksburg, Virginia, USA, 24060; Z. Tan, Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China, 430074.

1544-3566/2022/05-ART37
https://doi.org/10.1145/3527452

cost-effectiveness. To improve metadata space-efficiency and read performance, RocksMash uses an LSM-aware persistent cache that stores metadata in a space-efficient way and stores popular data blocks by using compaction-aware layouts. Moreover, RocksMash uses an extended write-ahead log for fast parallel data recovery. We implemented RocksMash by embedding these designs into RocksDB. The evaluation results show that RocksMash improves the performance by up to 1.7× compared to the state-of-the-art schemes and delivers high reliability, cost-effectiveness, and fast recovery.

CCS Concepts: • **Information systems** → **Cloud based storage**; **Hierarchical storage management**; **Key-value stores**; **Storage recovery strategies;**

Additional Key Words and Phrases: Log-structured merge tree, key–value, cache, succinct tree

ACM Reference format:
Peng Xu, Nannan Zhao, Jiguang Wan, Wei Liu, Shuning Chen, Yuanhui Zhou, Hadeel Albahar, Hanyang Liu, Liu Tang, and Zhihu Tan. 2022. Building a Fast and Efficient LSM-tree Store by Integrating Local Storage with Cloud Storage. *ACM Trans. Arch. Code Optim.* 19, 3, Article 37 (May 2022), 26 pages.
https://doi.org/10.1145/3527452

1 INTRODUCTION

Log-Structured Merge tree key–value stores (LSM stores), such as RocksDB [20], LevelDB [6], BigTable [21], and Dynamo [29], are widely adopted in database storage systems serving many modern applications, such as online retail, advertisements, online analytics, and instant messaging [32, 36]. **Log-Structured Merge tree (LSM-tree)** is preferred because of its write efficiency performance [21] as it does not perform in-place updates and buffers writes into memory. When the memory is full, writes are flushed to disks and subsequently merged using sequential disk I/Os. LSM-tree organizes data in levels and propagates new data from upper to bottom levels by internal compaction operations. These compaction operations filter out stale data, reducing the data size, and therefore, reducing the required storage space [31] and associated storage costs.

With the advent of cloud technology, cloud storage is increasingly popular for businesses looking to improve efficiency, disaster recovery, and agility [25, 53]. While some organizations have found that using local storage or a hybrid mix is a more effective solution. This is because local storage provides faster data access performance than cloud, and a complete security-oriented control over the data [51]. Therefore, more and more storage systems are deliberately designed to reap the benefits of both local storage and cloud storage by integrating local storage resources with public cloud storage services [61]. However, building a fast and efficient LSM store on a blend of local storage and cloud storage is challenging.

First, due to the imbalance of performance and cost between local and cloud storage, splitting LSM-tree between local and cloud storage to achieve a desirable performance and achieve cost-effectiveness is challenging. Cloud storage provides more space and higher data reliability at lower costs compared to local storage [1]. But we observed a considerable performance degradation, when building an LSM store using cloud storage, compared to using local storage. Moreover, reads are showing a severe (more than 98%) performance degradation than writes (about 40%) when using cloud storage (detailed in Section 2.2).

The second challenge is to develop an efficient cache that compensates for memory's limited size to improve read performance. Mutant [61] and some others approaches [9, 11] try to cache hot files on high performance storage to improve the read performance. They are, however, inefficient for both data and metadata. Their coarse granularity (typically several megabytes) is less space efficient. Besides, they are unable to effectively distinguish compaction invalidated data when managing data blocks. As a result, the cache hit ratio of these approaches is compromised, particularly when the working set data changes over time. Metadata, primarily index and filter, are not preferentially cached in memory or the persistent cache [56], despite the fact that every LSM read request will first visit them before sending an I/O for data blocks to storage. However, caching all **Sorted Strings Table (sstable)** metadata in memory occupies a significant amount of memory for large datasets [40]. Existing sstable metadata optimizations either focus on the index [43, 48] or the filter [62]; they rarely improve both at the same time with a unified structure and has non-zero false-positive rates for filtering, resulting in a suboptimal reduction of metadata size and alleviation of the number of metadata I/Os.

Last, in this extension, we further consider service availability issues for LSM stores that are hybrids of local and cloud storage, as data in local storage is not visible to remote nodes. So quickly resuming accesses to local data becomes crucial. Existing approaches mirror all sstables and **write-ahead-logs (WALs)** to the dedicated cloud storage by background copying [11, 46]. However, frequently writing sstables to cloud storage will cause significant data traffic and incur fees [4]. Besides, naively extending WALs to cover local data will increase its size, thus replaying WALs is essentially conducting the heavy upper level compactions [17], leading to suboptimal LSM store failure recovery performance.

In this article, we address the above issues by presenting the design of RocksMash. RocksMash is designed to exploit both local and cloud storage to achieve high performance and cost-effectiveness while maintaining fault-tolerance. RocksMash comprises a read-centric data layout, an efficient **LSM-aware persistent (LAP)** cache, a space-efficient and fast sstable metadata MashMeta, and a parallel recovery method.

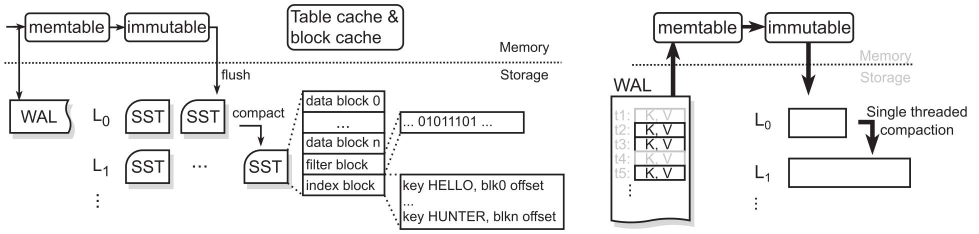
First, the LSM tiering philosophy facilitates RocksMash to constructe the data layout center on read performance. The sstables in the `top-k` levels are stored on local storage for fast access while the rest are stored on cloud for low storage cost (detailed in Section 3.1). In this way, most of the local storage space is spared for caching, improving its space efficiency for read performance.

Second, to speedup the reads to the data stored on the "remote" cloud storage, RocksMash proposes LAP cache to efficiently use the spared local storage space as the high performance persistent cache for hot data blocks and all sstable metadata. LAP cache reduces data block I/Os to cloud and eliminates metadata accesses to cloud storage. Furthermore, RocksMash creates MashMeta, which achieves zero false-positive filtering rates and significantly reduces the size of sstable metadata by encapsulating indexing information in filters.

Third, in this extension, to ensure fault-tolerance and improve data recovery performance for LSM stores that are hybrids of local and cloud storage, RocksMash extends the WAL with negilible overhead and stores them on a dedicated, fast, and small cloud storage to expose local data to remote nodes. During recovery, the proposed extended WAL eliminates time-consuming upper-level compactions and significantly speeds up the construction of sstables, resulting in improved recovery performance.

We make following contributions in this work.

- We make observations that guides the design of integrating cloud storage to LSM stores.
- We propose an LAP that improves the space efficiency and reduces accesses to cloud storage.
- We propose a space-efficient and high performance metadata structure MashMeta for sstables that significantly reduces the metadata space overhead and eliminates unnecessary I/O requests.
- We propose a parallel failure recovery approach that ensures a quick resumption of services remotely. To the best of our knowledge, the proposed parallel recovery in this extension is the first work that remotely recovers LSM stores using loosely coupled data dependencies among sstables.



(a) The LSM store overview

(b) Recover the LSM store with WAL. It replays key-value pairs from the WAL in the chronological order. This causes expensive L_0 to L_1 compactions

Fig. 1. The LSM store overview and recoverying with the WAL.

- We implemented RocksMash based on RocksDB and evaluated its performance by using an instance's physically-attached NVMe SSD as local storage and **AWS Elastic Block Store (EBS)** gp2 as cloud storage. The evaluation results show that the performance of RocksMash outperforms the state-of-the-art scheme Mutant by up to 1.7×, and improves the recovery performance by up to 10×.

In the rest of this article, we first introduce LSM stores and our motivation in Section 2. We present the design of RocksMash in Section 3. In Section 4, we present our experimental results. Related work is presented in Section 5. Finally, we conclude this article in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Background

LSM store architecture. LSM stores persist data in the unit of an sstable and organize them in a multilevel treelike structure as shown in Figure 1(a). The size quota of each level grows larger from L_0 (typically a few hundreds of megabytes) to L_n (up to terabytes).

Sstable. An sstable consists of data blocks that store user key–value pairs and metadata blocks that mainly contain filter blocks and index blocks, as shown in Figure 1(a). All key–value pairs among data blocks are sorted by keys, and a data block usually contains several key–value pairs. So LSM stores use the starting key of each data block as the santry and put these keys and their belonging block offsets in the sstable into index blocks. Binary search on santry keys in index blocks can quickly locate a target key in a data block. To avoid unecessary accesses reading data blocks, LSM stores add filter blocks that use bloom filters [18] to probe the existence of a key in this sstable. A positive probe result indicates the target key is *probably* in this sstable, and a false probe result ensures its non-existence and saves a read request to the data block.

Write. As shown in Figure 1(a), an LSM store first writes data to the WAL to ensure durability of data during system failures. After that, the LSM store buffers the data in an in-memory data structure that is typically a skip-list *MemTable*. When a MemTable is full, it is turned immutable and later flushed to the disk as sstables, then placed at level L_0.

Compaction. When the size of L_i reaches its quota, a number of sstables at level L_i are merge-sorted with a list of overlapping sstables at L_{i+1} and stale key–value items are discarded, this is called *compaction*. As a result, compaction operations ensure that there are no overlapping key-ranges between sstables at level L_i where $i \neq 0$.

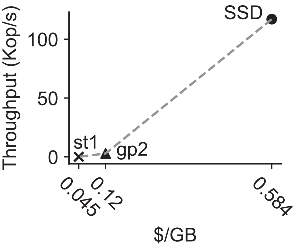


Fig. 2. The cost and performance of different storage services. The SSD is the instance-attached SSD. gp2 and st1 are AWS EBS that use SSD and HDD, respectively.

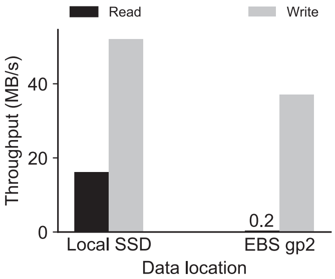


Fig. 3. RocksDB performance on the local SSD (instance attached) or the cloud storage (EBS-gp2) when memory is exhausted.

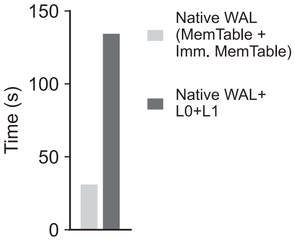


Fig. 4. The recovery time increases when the WAL file coverages change.

Read. When a read request is received, the MemTable and immutable MemTables will be first checked. If the requested value can be found in either one of them, then the request will be served from memory. Otherwise, sstables of all LSM levels are searched from top to bottom until the target key is found.

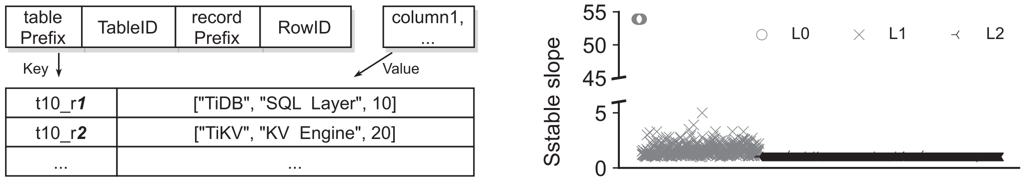
To accelerate locating a key in the LSM-tree, the LSM store records the smallest and largest key of each sstable in memory, to use as boundaries, to quickly narrow down the search to an sstable. Before locating data blocks in the sstable, the sstable's metadata blocks are first accessed. First, filter blocks are used to probe whether a key is stored in the candidate sstable. Note that filter blocks can cause false-positive errors. Upon receiving a positive probe result, index blocks will be used to narrow down the search to a data block that may store this key. This is followed by an in-block search that will return the target key and its value if found. A false-positive probe result causes in-block search misses, and the LSM store continues to search the next level in LSM-tree until all levels are searched. Frequently accessed metadata and data blocks are cached in the *table cache and block cache*. So the LSM store first looks up the table and block cache for requested metadata or data blocks and sending I/Os to storage when needed. Note that metadata blocks use nontrivial amount of memory space [7].

Recovery with WAL: Upon each write request, the LSM store starts by recording it in the WAL and then apply the change to the mutable MemTable as shown in Figure 1(a). Changes are simply recorded in chronological order in the WAL to recover key–value pairs in memory. After flushing immutable MemTables to L_0 sstables, corresponding changes in the WAL could be safely removed. Meanwhile, when inserting or deleting sstables, the LSM store logs these file changes to the *MANIFEST*. As shown in Figure 1(a), the LSM store could be restored to the latest consistent state after checking all sstables with the MANIFEST and restoring MemTable and immutable MemTables by replaying WAL records. Note that the size of file MANIFEST is small and usually in the magnitude of kilobytes.

2.2 Motivating Observations

2.2.1 Imbalanced Cost and Performance between Local and Cloud Storage. Figure 2 shows an example of the performance-cost trade-off for RocksDB built by using local storage (i.e., Amazon EC2 instance store [3]) and cloud storage service (i.e., EBS-gp2 in Amazon [1]).

Local storage and cloud storage exhibit varying performance and costs. For example in Figure 2, using cloud storage (EBS-gp2) can reduce around 80% of the cost compared to the



(a) Mapping SQL table data to key-value pairs. A key is encoded by a sequence of a table prefix (“t”), a table identification (“10”), a record prefix (“_r”), and a row identification (“1”).

(b) Significant key prefixes in bottom level sstables. The slope of an sstable is obtained by dividing the difference between the maximum and the minimum key by the number of keys.

Fig. 5. Strong common key prefixes for database applications.

local server-attached SSD. Beside, the cloud storage is billed in actual resource usages and provides a smooth storage volume expansion. As a result, more and more users move their applications to the cloud [8]. However, cloud storage has a lower throughput for both read and write requests. In Figure 3, the read and write throughput of RocksDB that is built on local storage (300-GB SSD) compared to RocksDB that is built on cloud storage (1-TB EBS-gp2) is 80× and 40% higher, respectively. Hence, designing an LSM store that offers the benefit of both local and cloud storage is non-trivial. An efficient LSM store that uses local and cloud storage shall prioritize improving the read performance in designs.

2.2.2 Similar Prefixes among Keys. To reduce read I/Os to cloud storage, it is important to avoid sstable metadata accesses to it. Reducing the metadata size is a necessary design direction as the metadata occupies a non-trivial memory space [7]. We notice that, as shown in Figure 5(a), strengthening the common prefix of keys is quite common when mapping SQL table data to key–value stores in various databases [5, 12, 48]. Moreover, LSM store compaction operations consolidate the prefix similarity among keys in an sstable, so this makes keys in bottom level sstables tend to share longer prefixes. In this case, indexing keys in an sstable is intrinsically indexing sorted strings with high similarity.

Figure 5(b) shows the common prefix similarity of keys in different level sstables by the term of slopes. We insert 100M random integers and calculate the slope of each sstable. Gentle slopes indicate the longer common prefixes in an sstable and vise versa. In Figure 5(b), bottom levels show flat slopes, which means keys are almost identical except for the last byte in bottom level sstables. So we leverage this strong similarity of keys to reduce both the filter and index size of sstables.

2.2.3 Slow Recovery. Because the native WAL does not record all of the data in these levels, sstables from L_0 to L_i stored in local storage will be lost if the instance fails using the basic recovery approach introduced in Section 2.1. Simply extending WAL to include data from L_0 to L_i sstables is suboptimal for following two reasons. First, naively replaying each valid write request in the WAL to reconstruct L_0 to L_i sstables will repeatedly write and read some key–value pairs several times [47], which is in vain and time consuming. Besides, this replay will result in L_0 to L_1 compactions, and key range overlaps among L_0 sstables will prevent LSM stores from handling them concurrently [17]. Once the size of WAL files are larger, the recovery process will be longer. For example, extending the native WAL to cover L_0 and L_1 sstables increases recovery time by more than 4×, as shown in Figure 4. Second, this basic recovery method cannot generate the exactly same L_0 to L_i sstables as in the failure node. This is because recovery operations build sstables based on empty L_0 to L_i, while those L_0 to L_i sstables on the failure instance are not.

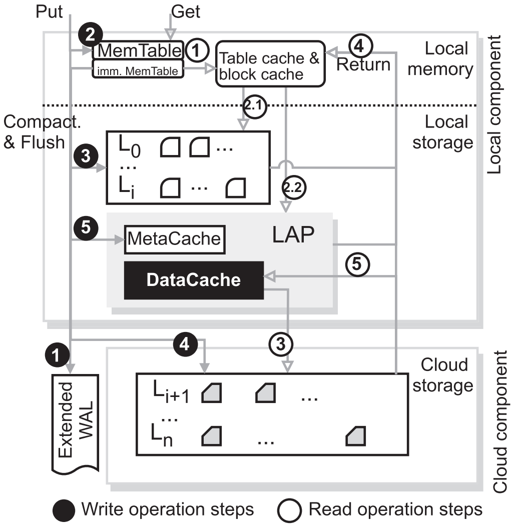


Fig. 6. RocksMash write and read operation dataflows. The local storage is partitioned into two areas, one for L_0 to L_i sstables, and the other for LAP cache.

We notice that sstables are "written once" but "read many times," implying that each sstable can be recovered independently. Except for reading WAL files, this method requires few I/Os for recovery. Thus, directly building top-k level sstables rather than recovering them indirectly through compactions is more efficient, particularly for upper level compactions.

3 DESIGN

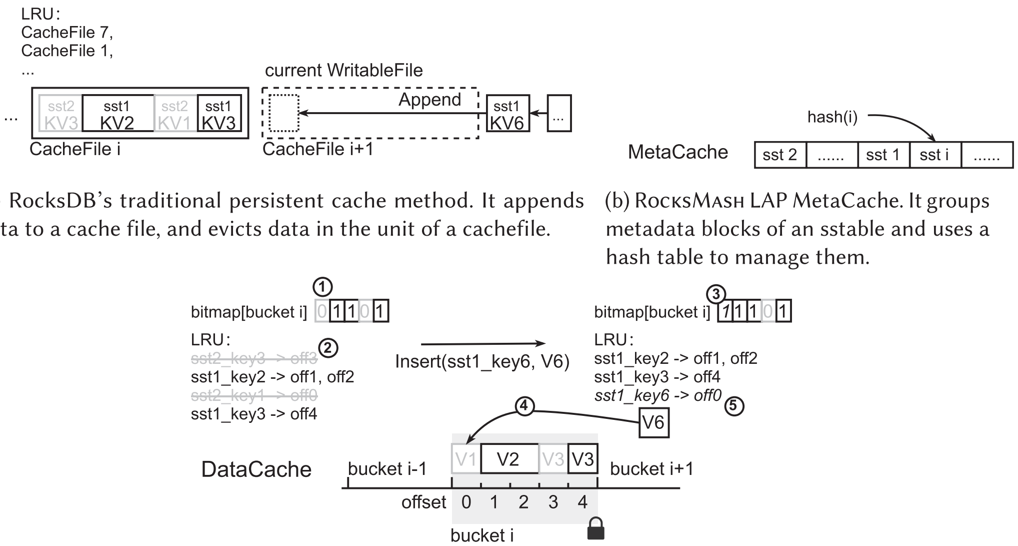
We introduce the design of RocksMash, a fast and efficient LSM store using a mix of local storage and cloud storage. We first provide an overview of RocksMash (Section 3.1). We then describe in detail how it reduces I/Os to cloud storage and improves the cache space efficiency with LAP cache (Section 3.2). After that, we introduce how to reduce the metadata size by MashMeta (Section 3.3). Finally, we introduce how RocksMash quickly recovers from a failed instance (Section 3.4).

3.1 Overview

RocksMash uses the small, fast, and usually expensive local storage for storing the frequently accessed top i levels of the LSM-tree, while the rest of the LSM-tree is maintained by large, slow, and usually cheap cloud storage. Since upper levels are much smaller and hotter than lower levels, we move the top levels of the LSM-tree to the local storage. This relocation offloads a considerable amount of time-consuming operations such as flush and compaction from the slower cloud storage to the faster local storage.

Figure 6 details the storage structure of RocksMash. RocksMash retains the original LSM MemTable in memory that contains one mutable and multiple immutable MemTables.

Moreover, RocksMash uses a fine-grained LAP cache to further mitigate accesses to the data stored in cloud (details in Section 3.2). LAP cache is partitioned into a MetaCache and a DataCache. Since metadata are more frequently accessed than data for LSM stores, the MetaCache stores the metadata blocks of *each* sstable stored in cloud to eliminate metadata accesses to cloud. While the DataCache caches the frequently accessed data blocks in sstables and evicts data in a compaction-aware way. The extended WAL is stored on dedicated and fast distributed storage for fast and parallel recovery of sstables (details in Section 3.4).



(a) RocksDB's traditional persistent cache method. It appends data to a cache file, and evicts data in the unit of a cachefile.

(b) RocksMash LAP MetaCache. It groups metadata blocks of an sstable and uses a hash table to manage them.

(c) RocksMash LAP DataCache. It evicts stale data of sst2 by resetting their bitmap bits (steps 1 and 2), and inserts key-value pair 6 of sst1 (steps 3, 4, and 5).

Fig. 7. The comparison of persistent cache methods.

Note that when level L_i (in local storage) is merged and flushed to the next level L_{i+1} (in cloud), a certain amount of data will be transferred between local storage and cloud storage, we call this process remote compaction. The overhead of remote compaction will not severely affect performance (in Section 4).

Request processing. Figure 6 illustrates the write and read request processing. Upon the arrival of a write request, RocksMash first stores the data into the extended WAL log (❶, detailed in Section 3.4), then updates the MemTable (❷). When the MemTable is full, immutable MemTables will be flushed into local storage, which also triggers compaction (❸) and leads to remote compaction (❹). Note that after remote compaction, RocksMash inserts the new sstables' metadata blocks into LAP MetaCache in the local storage for the future data accesses (❺).

For reads, if the target key cannot be found either in MemTable or immutable MemTables (①), RocksMash proceeds to search sstables from level L_0 to L_n for candidate sstables. RocksMash uses sstable key boundaries to locate each candidate sstable and checks if it contains the key. Before actually reading a data block from a candidate sstable on storage, RocksMash first uses its metadata to probe the existence and get the address of the target key and conducts a data block read if the probing returns a positive result (detailed in Section 3.3).

If the target key is found in L_0 to L_i sstables, then RocksMash will read the data blocks from local storage ((2.1)). Otherwise, RocksMash will check whether the requested data block is in the DataCache within LAP cache ((2.2)). If not, then RocksMash will check whether the requested key is in the cloud storage (③). If yes, then the request will be served (④) by the cloud storage and be forwarded to the LAP DataCache (⑤) for the future accesses.

3.2 LSM-Aware Persistent Cache

The LSM store enables the use of high performance devices (e.g., SSD) as a persistent cache to improve the read performance when the available memory cache is exhausted. However, few schemes optimize the persistent cache for LSM stores. Take RocksDB as the example, its persistent cache usually contains multiple cache files that are organized in a LRU list. As shown in Figure 7(a), a key–value pair (say kv6 from sst1) is loaded from slow disks into RocksDB's persistent cache on a read miss. The key–value pair is appended to the most-recently-used writable cache file CacheFile i+1 as shown in Figure 7(a). If the writable cache file is full, then it will be turned to read-only and a new cache file will be created and inserted to the cache file LRU list. When the persistent cache is full, the least recently used cache file will be deleted.

RocksDB persistent cache is unaware of compaction, which wastes a significant fraction of cache space. This is because compaction leads to invalidating key–value pairs in the sstables that are compacted. For example, sstable sst2 is compacted and invalidated (in gray) as shown in Figure 7(a). However, invalid key–value pairs kv3 and kv1 from sstable sst2 are still kept in cache file CacheFile i, which causes a significant waste of the cache space and also premature evictions of the needed keys (kv2 and kv3 from sstable sst1).

To address these issues, RocksMash uses an LSM-tree-aware persistent cache on local storage, called LAP. LAP is divided into a MetaCache and a DataCache.

MetaCache. MetaCache stores all the metadata blocks of the sstables stored in cloud to eliminate remote metadata accesses. The total size of metadata is non-trivial when compared to the memory size [7], but it is small compared to local storage whose size is typically at least tens or hundreds of gigabytes. We discuss how to efficiently reduce metadata size in Section 3.3. As sstables are created once and read many times, MetaCache groups the metadata of an sstable and manages them with a hash table as shown in Figure 7(b). With MetaCache, RocksMash prevents metadata blocks from been evicted from local storage. This makes RocksMash require only one I/O on local storage to get metadata in the worst case.

DataCache. DataCache is a compaction-aware cache that stores popular data blocks to speedup read requests. As shown in Figure 7(c), key–value pairs stored in DataCache are organized in a LRU list. Key–value pairs of an invalid sstable are marked as invalid in the bitmap, such as sst2_key3 and sst2_key1 in sst2 (steps 1 and 2). When a key–value pair sst1_key6 is loaded into DataCache on a cache miss, the invalid key–value pair will be overwritten by newly added key–value pair sst1_key6 (steps 3, 4, and 5) to improve cache efficiency.

As shown in Figure 7(c), the DataCache is partitioned into multiple buckets, and each bucket holds several blocks from different sstables. A block is assigned to a bucket by hashing its sstable file name. A bitmap is used for each bucket, where each bit in the bitmap corresponds to a block, and if the bit is set, it indicates that the corresponding block holds a valid value. With DataCache, invalid sstable data blocks are removed by resetting their bits in bitmaps. These operations will not affect valid data. This block-level management improves the cache space efficiency.

3.3 Space-Efficient and Fast Indexing and Filtering (MashMeta)

To further improve cache space efficiency, RocksMash uses a space-efficient indexing method, termed as the MashMeta, based on succinct trees [38]. The space-efficient indexing method reduces metadata space consumption that makes room for more data blocks, and thus improving the cache hit ratio. This is because succinct data structures can represent the keys and occupy an amount of space that is closer to the information-theoretic lower bound, while supporting fast queries [38].

Given a key, RocksMash first uses sstable boundaries to locate the candidate sstable that stores the key–value pair. After that, RocksMash uses its metadata to locate a data block for the final search of the key. For each sstable in cloud storage, RocksMash constructs a compressed trie, or a compressed prefix tree, and encodes the compressed trie in a novel compound succinct tree

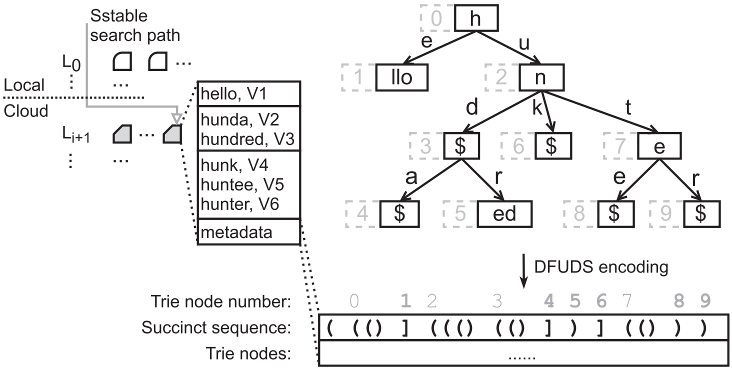


Fig. 8. The space-efficient MashMeta of an sstable in cloud storage. On the left, six key–value pairs are stored in three data blocks. On the right, a trie for theses keys and their succinct encoding. Numbers in dashed rectangles in the trie are the depth-first node number. The bold trie node number indicates this node is a leaf node. A "]" in the succinct sequence indicates a starting key of a data block.

structure as the MashMeta, and uses it as replacement of both index and traditional bloom filter of sstables in cloud. Figure 8 shows an example of a 10-node compressed tree (trie) that represents the following keys "`hello`," "`hunda`," "`hundred`," "`hunk`," "`huntee`," and "`hunter`" in an sstable in the cloud. These key–value pairs are stored in three data blocks. Each node is associated with a distinct node character and number, such as `h` and `0` for root node. The number of a trie node is the depth-first traverse order. While each branch, or edge, is labeled with a branching character, such as branches `e` and `u`. Each path from the root to a leaf node represents a key string. A leaf node indicates the ending of a key.

The compressed trie is encoded in depth-first unary degree sequence (DFUDS) [33]. Figure 8 gives an example of trie DFUDS encoding. During trie depth-first traversal, `i` open parentheses "`(`" are written when a node is visited the first time, and a closing parenthesis "`)`" is written after that node traversal, where `i` is the number of children of that node. This results in a sequence of 20 balanced parentheses as shown in Figure 8. Consequently, the node is represented by the position where its parentheses start. For example, node `0` is represented by a sequence of open and close parentheses"`(()`". While key `hello` is represented by the *path* from root node `0` to leaf node 1. In this way, a close parenthesis "`)`" of a leaf node indicates the ending of a key. Note that the left most "`(`" in the succinct sequence is purposely added to obtain a sequence of balanced parantheses [33].

To locate the data block in the candidate sstable that contains a key with the encoding sequence, for each data block, RocksMash first replaces the closing parenthesis "`)`" of the leaf node in the path of the first key with a close square bracket "`]`". For example, for the first data block, the leaf node (node 1) in the path of its first key `hello` is represented by "`]`" as shown in Figure 8. Since the keys in sstables are sorted, "`]`" marks the boundaries for neighbour data blocks. Consequently, while a key is searched in the DFUDS sequence, the number of "`]`" encountered until the target key is found indicates the data block number. For example, there are 2 "`]`" visited before key "`hundred`" is located in DFUDS order, which means that "`hundred`" is stored in the second data block. Accordingly, RocksMash can achieve precise and fast query for each key and also avoids the filter's false-positive errors with minimal space consumption.

3.4 Parallel Recovery

With RocksMash's data layout, restoring access to the L_0 to L_i level sstables stored on the failure node's local storage quickly is critical for data availability. In this work, we assume that sstables

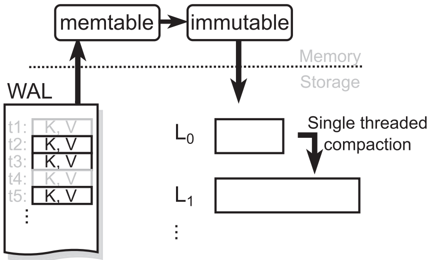


Fig. 9. Recover the LSM store with WAL. It replays key–value pairs from the WAL in the chronological order. This causes expensive L_0 to L_1 compactions.

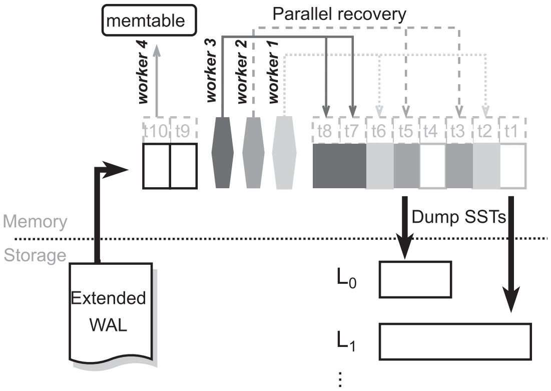


Fig. 10. Recover the LSM store with the proposed extended WAL in parallel. RocksMash adds L_0 to L_1 sstable key lists (hexagons) into the WAL.

on cloud storage are reliable, because cloud storage has the lower annualized failure rate [34] and can be accessed directly from other nodes. To expose local sstables to remote nodes for recovery, RocksMash stores the WAL file on a fast and dedicated cloud store (e.g., EBS-gp3). The data size of top-k level sstables stored on local storage, however, is several gigabytes, resulting in RocksMash WAL files that are much larger than the vanilla RocksDB. As a result, the demand to quickly restore L_0 to L_i sstables from large WAL files makes recovering storage service on a remote node difficult.

For the basic recovery method, illustrated in Figure 9, its insight is to *sequentially* redo write operations, which is inefficient for processing large datasets. Because every recorded key–value pair in WAL files is considered valid, the basic recovery method will conduct many write operations. According to the write path, these key–value pairs will first form Memtable and immutable MemTables in memory, then transferred to sstables and persisted to the recovery node's local storage. This basic method is inefficient in terms of both CPU usage and the number of I/O requests when the WAL size is large and covers key–value pairs for L_0 to L_i sstables. First, it is not necessary to construct the MemTable or immutable MemTable, because doing so generates only L_0 sstables but no L_0 to L_i sstables, wasting CPU cycles. Second, a large number of continuous write operations will inevitably result in L_0 to L_i compactions. As L_0 sstables have key range overlaps, L_0 to L_i compactions typically involve all sstables from these two levels, resulting in a large number of I/Os. Worse, L_0 to L_i compactions are not parallelized, further degrading recovery performance.

RocksMash proposes *parallelizing* the restoration of L_0 to L_i sstables to address the aforementioned inefficiency for large size WAL. However, there are two challenges to overcome.

- How can the construction of sstables be parallelized? Vanilla LSM stores only record key–value pairs to the WAL, and the MANIFEST only keeps track of the list of valid sstables. Each sstable is unaware of the location of the key–value pairs to which it is assigned in WAL files. The basic recovery design lacks the information required to connect key–value pairs in the WAL and valid sstable file names in the MANIFEST, needing time-consuming upper level compactions to correctly match them. As a result, for parallel recovery, RocksMash must establish a connection between key–value pairs in WAL files and L_0 to L_i sstables.
- With the above connection, how can RocksMash maximize parallelism while ensuring that read operations get the correct key–value pair after recovery? If each sstable searches WAL

files in chronological order, as the basic recovery method does, then a large number of out-of-date key–value pairs are picked up if there are many updates. However, due to the read path of LSM stores, these stale key–value pairs will eventually be deleted by compactions and will no longer be visible to users. So RocksMash needs to improve its scan efficiency for sstables.

To address the two aforementioned parallel recovery challenges, as depicted in Figure 10, RocksMash proposes the *extended WAL* (hexagons in Figure 10) as the connection between key–value pairs in WAL files and valid sstable lists in the MANIFEST. The idea is to trade faster recovery time for slightly more storage space. RocksMash searches WAL files in *reverse chronological order* to reduce data to be processed and thus improve scan efficiency. RocksMash logically reduces the size of WAL files by scanning them in this manner, because old WAL files usually contain out-of-data key–value pairs that are not expected for read requests.

Extended WAL: RocksMash extends the WAL to include all data in sstables from L_0 to L_i. When an L_0 to L_i sstable is generated, RocksMash logs its changes to the MANIFEST, then records its *key lists* (hexagons in the extended WAL in Figure 10) at the same time to the WAL for quick identification of sstable keys. Because a compaction job is not considered complete unless all new sstables are successfully persisted, RocksMash can batch key lists of new sstables to reduce the I/O number writing them. This way, RocksMash ensures a safe rolling back to the latest consistent state with the MANIFEST and key lists, and no valid data will be lost, because the only difference between new and deleted sstables are the removed stale data. The key lists of deleted L_0 to L_i sstables in WAL are not visible to RocksMash, because deleted sstables are not included in the latest consistent state in the MANIFEST.

Reverse chronological order search: Each sstable that needs to be restored is assigned a worker by RocksMash. Each worker searches the new WAL files first, followed by the old files. In this way, a worker ensures that the most recent version of a key–value pair is met first, in accordance with the read path of LSM stores. Furthermore, RocksMash enables a worker to retrieve any version of a key for L_x sstables ($x \neq 0$, $x \in [1, i]$). In another word, random versions of a key–value pair in L_i sstables does not affects the correctness of future read request results after resume. Suppose an L_x sstable worker gets a random version of a key–value pair (say, key$_{suspect}$), key$_{suspect}$ would be the only valid verion in all L_x sstables, because there are no key overlaps among L_x sstables. If key$_{suspect}$ is an out-of-date version, then there must be a latest version of it in an L_0 to L_{x-1} sstable, and key$_{suspect}$ will not be first met by the LSM search routine. This pick up rule, combined with the reverse chronological order search, allows RocksMash to logically truncate WAL sizes when there are a lot of updates in WAL files.

Building sstables in parallel: When the instance node fails, RocksMash uses a new instance to recover L_0 to L_i sstables stored in the failure instance. RocksMash reads the MANIFEST and the extended WAL to get sstable file lists of the last consistent state. RocksMash then fetches data for L_0 to L_i sstables from the extended WAL with the help of key lists, and rebuilds these sstables in parallel. Figure 11 shows an example of the recovery steps:

- *step ① read WALs*: RocksMash fetches the L_0 to L_i sstable key lists and all WAL files to the new instance node.
- *step ② find key values*: After getting key lists of valid sstables RocksMash assigns a worker for each sstable to scan WAL files and fetch keys in its list. Scanning should begin from the newer WAL files to older ones to ensure workers meet the latest key first. As L_0 sstables have key range overlaps, the scanning pointer of the older L_0 sstable workers are not allowed to exceed scanning pointers of the younger L_0 sstables workers (step 2.1). This ensures younger L_0 sstables always get the latest key–value pairs for correctness. After L_0 workers complete

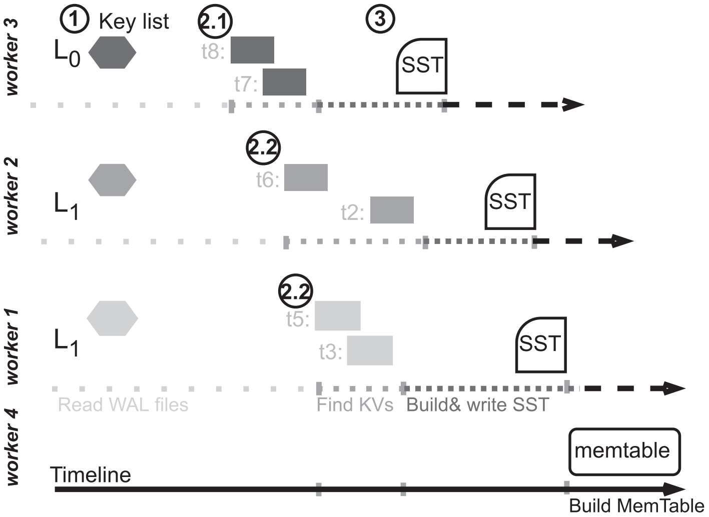


Fig. 11. RocksMash recovers sstables in failure instance node storage from the extended WAL in parallel. It reads sstable key lists from the WAL (step 1), then retrieves key–value pairs from the WAL in parallel for multiple sstables in parallel (steps 2.1 and 2.1). And afterwards, it rebuilds sstables (step 3). It finally replays remaining key–value pairs (belowing the bolded bar) in the extended WAL for building the MemTable (step 4).

scanning, L_1 to L_i sstable construction workers are free to fetch any version of key–value pairs if needed (step 2.2).

- *step ③ build and write sstables*: A worker begins to build the sstable upon getting all its key–value pairs. Recovered sstables are installed to RocksMash after reconstruction.
- *step ④ build MemTable*: After recovering L_0 to L_i sstables, RocksMash starts to restore MemTable and immutable MemTables. RocksMash leverages the information of the latest L_0 sstable to reduce the size of WAL records to replay. Since every change is logged to the WAL in chronological order, the latest key in the latest L_0 sstable indicates a position, where keys are earlier than this time were safely persisted in sstables and keys younger were still in MemTable or immutable MemTables. In this case, RocksMash replays WAL records younger than the latest key of the latest L_0 sstable to restore data in memory.

After above steps, RocksMash completes the recovery for data in the memory and local storage of the failed instance to the latest consistent state.

Simply preserving all data of L_0 to L_i sstables in the WAL will largely increase the size of WAL, so RocksMash periodically flushes the L_0 and L_i sstables to their next levels to shrink the size of WAL files by leveraging the RocksDB built-in **time-to-live (TTL)** mechanism [10]. This flush mechanism flushes sstables of L_0 to L_i older than the predefined time, and we evaluate this effect in Section 4.5.

4 EVALUATION

We answer two questions in the evaluation: How does RocksMash perform compared to state-of-the-art methods? and How effective are RocksMash LAP cache, the MashMeta, and the parallel recovery?

Workload	Operations	Distribution
A	50% R, 50% U	Zipfian
B	95% R, 5% U	
C	100% R	
D	95% R, 5% I	Latest
E	95% S, 5% I	Zipfian
F	50% R, 50% RWM	

Fig. 12. YCSB workload characteristics. R: Read, U: Update, I: Insert, S: Scan, RMW: Read-modify-write.

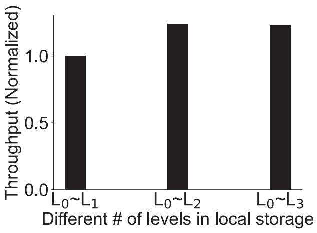


Fig. 13. Performance of putting different number of LSM store levels in local storage under random writes.

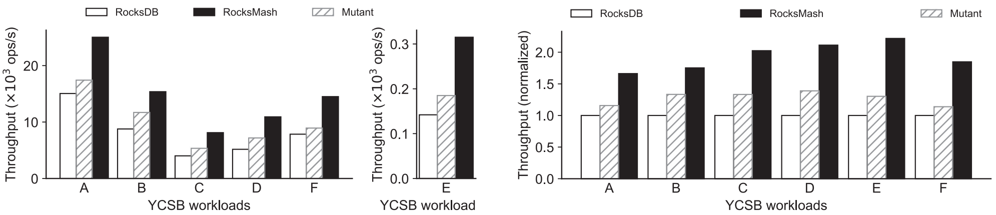
4.1 Evaluation Setup

Testbed. Our testbed consists of a five-node local storage cluster (two management nodes and three storage nodes) and a 3-TB AWS EBS-gp2 cloud storage. Each node in the local storage cluster is a m5d.2xlarge AWS EC2 instance that is equipped with 8 vCPUs, 32 GB memory, and a physically-attached 300 GB NVMe SSD [3]. Each storage node deploys an LSM store as the storage engine that uses the NVMe SSD as local storage, connects a 1-TB EBS-gp2 volume as cloud storage, and connects a 32-GB EBS-gp3 volume with the maximum provisioned performance for the WAL.

Benchmarks. Two benchmarks, YCSB benchmark [24] and db_bench[10] are used to evaluate RocksMash performance compared to the state-of-the-art methods. YCSB is used to simulate real-world workloads. YCSB has six workloads denoted as A to F, and their characteristics are listed in Figure 12. db_bench is used to test RocksMash individual designs. db_bench generates different types of workloads, such as random writes, sequential writes, random reads, range query, marked as "FR," "FS," "RR," and "SR," respectively. Throughout the evaluation, the default key, value, and sstable size are set to 16 B, 256 B, and 8 MB, respectively, and each workload contains 10 M requests. After loading YCSB workloads, the total data size is about 50 GB, and the data size of L$_0$, L$_1$, and L$_2$ is about 1.3%, 2.4%, and 96.3%, respectively. After loading db_bench workload, the total data size is about 5 GB, the data size of L$_0$, L$_1$, and L$_2$ is about 5%, 13%, and 82%, respectively. As a result, unless otherwise noted, the majority of data is stored on cloud storage.

Schemes. We compare RocksMash with two state-of-the-art schemes, RocksDB [10] and Mutant [61]. RocksMash is implemented based on RocksDB, and we embed our sstable placement method by modifying the compaction module. As for the effect of the number of LSM levels stored in the local storage, Figure 13 shows the throughput of varying the i of L$_i$ from 1 to 3 under random write requests. Remote compaction operations happen between L$_i$ and L$_{i+1}$ have a modest effect, especially when i is larger than 2. Considering the workload size of our evaluation, RocksMash puts L$_0$ and L$_1$ in local storage and other levels in cloud storage by default. The LAP cache is mainly implemented in the persistent cache module. The MashMeta replaces metadata structures for sstables in the cloud. The parallel recovery is implemented by modifying the WAL module and recovery procedures.

RocksDB stores all data on cloud storage. This is because the current RocksDB is not designed for a hybrid or a mix of different types of storage infrastructures. We use this RocksDB configuration to simulate the scenarios where most of the data is stored in cloud storage to reduce cost. Mutant caches hot sstables in the local storage, where its cache size on the local storage is controlled by the overall storage cost target per gigabyte data for both local and cloud storage. We borrow its default configuration that sets the target cost to \$0.3G/month. This target cost for Mutant indicates that



(a) Throughput with gp2 and SSD under YCSB workloads.

(b) The normalized throughput with gp2 and SSD under YCSB workloads.

Fig. 14. Performance on a single instance node.

the local and cloud storage size to be 41% and 59% of the workload size, respectively. We believe this configuration for Mutant is fair to unleash its performance. By default, we set the persistent cache size to be 20% of the workload size.

4.2 Overall Performance

We first evaluate the performance of RocksMash and compare it with the performance of RocksDB and Mutant. Next, we integrate RocksMash with TiDB [35], an open source distributed Hybrid Transactional/Analytical Processing database that supports MySQL syntax, to simulate real-world scenarios.

Figure 14(a) and (b) show the *absolute* and normalized throughput (Kops/second) for RocksMash, Mutant, and RocksDB. Note that in this experiment, RocksMash and Mutant are running on the storage node, which has a blend of 1-node local storage (300-GB NVMe instance storage) and cloud storage (1-TB EBS-gp2). As shown in Figure 14(a), RocksMash achieves the highest throughput for all six workloads. For example, RocksMash improves throughput by 2× compared to RocksDB under workload C. This is because RocksDB stores all data on cloud storage. Compared with Mutant, RocksMash's throughput increases by 1.6× for workload F, because Mutant's sstable level cache efficiency is lower than RocksMash's fine-grained LAP cache. We also observed that all three schemes show a better throughput for A, B, and F, because these workloads contain a large proportion of reads and follow a cache friendly Zipfian distribution. Moreover, all three schemes have the lowest throughput for range query workload E, as LSM stores' traversal to levels causes amplified I/Os. However, RocksMash achieves a good improvement by 2.2× and 1.7× compared to RocksDB and Mutant, respectively, for workload E. The reasons for this are the following. First, compared to RocksDB, RocksMash stores the popular sstables at upper levels on the fast local storage, which reduces the need for frequent slower data retrievals from cloud and therefore improves the performance of the whole system. Second, although Mutant caches hot sstables in local storage, its file level migrations to cache are heavier and its cache space-efficiency is suboptimal.

Next, we integrate these three schemes with TiDB as a row store engine. In this experiment, the local storage cluster contains 5 nodes. We first load random key–value pairs into TiDB, and then run six YCSB workloads and measure the performance. Results are given in Figure 15, and the Y axis shows the normalized throughput. Similarly to Figure 14, RocksMash outperforms both RocksDB and Mutant across all six workloads. For example, RocksMash can largely improve the throughput by 1.8× and 1.7× compared to RocksDB for workloads C and E, respectively. Compared to Mutant, RocksMash's throughput increases by 1.3× for workload D. Due to the data replication

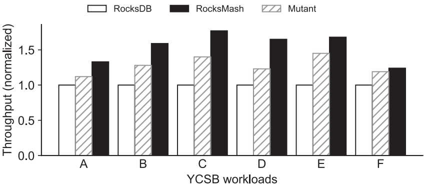


Fig. 15. Performance under the distributed database.

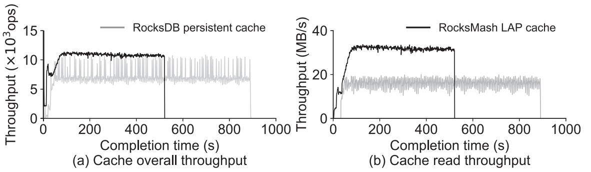
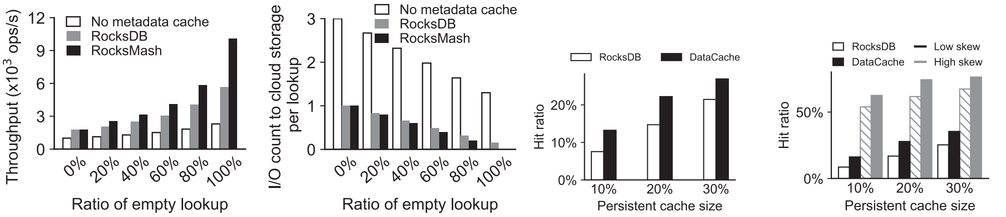


Fig. 16. The persistent cache performance under the read-only YCSB workload C. The line for RocksDB is in gray, and the line for RocksMash is in black.



(a) Random read performance with different metadata caches and varied empty lookup ratios.

(b) I/O count to cloud storage with varied percent of empty lookups.

(c) Persistent cache hit ratios under uniform random read workloads.

(d) Persistent cache hit ratios under skew random read workloads.

Fig. 17. Effects of LAP MetaCache and DataCache.

policy among storage nodes by TiDB, there is more write traffic, but RocksMash still improves the performance.

4.3 Cache Effectiveness

Next, we analyze the effectiveness of RocksMash LAP cache. To understand how LAP cache improves RocksMash's performance, we first evaluate the overall performance of LAP cache and then separately evaluate the performance of each of its components, the MetaCache and DataCache.

Overall cache performance. Figure 16 shows the overall and the read throughput for both LAP cache and RocksDB traditional persistent cache under workload C. Workload C is a read-only workload of YCSB. In Figure 16(a), the LAP cache shows a stable and higher overall throughput than the persistent cache of RocksDB. The throughput of RocksDB persistent cache fluctuates because of its low space-efficiency (detailed in Section 3.2). As shown in Figure 16(b), compared to RocksDB, RocksMash's LAP cache provides a much higher read throughput to users due to its high space efficiency and high cache hit ratio (detailed in Section 3.2). Moreover, we observed that LAP cache almost halves the benchmarking completion time compared to RocksDB because of its higher performance.

MetaCache. When not all metadata can be stored in memory, RocksMash MetaCache is intended to cache all metadata on local storage, thereby reducing metadata I/Os to the cloud. To demonstrate MetaCache's effectiveness, we compare RocksDB with metadata cache (i.e., index and filter block) on local storage and RocksMash with MetaCache. The cloud stores all sstables. To simulate the worst-case scenario, we add a RocksDB configuration that disables all memory

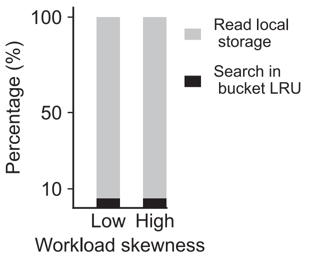


Fig. 18. LAP DataCache lookup latency breakdown.

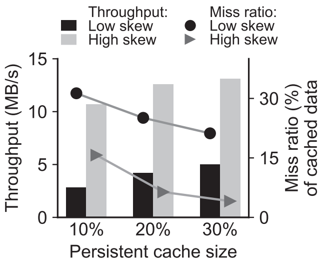


Fig. 19. RocksMash random read throughputs and miss ratios of cached data for various persistent cache sizes.

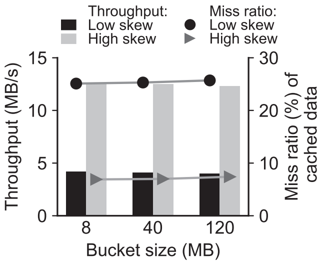


Fig. 20. LAP DataCache miss ratios of cached data for various bucket sizes.

and persistent cache mechanisms, and we label this configuration as "No metadata cache" in Figures 17(a) and (b). Using random read workloads with empty lookups (i.e., search for non-existent keys), we evaluate read throughput (Figure 17(a)) and how many I/Os to the cloud for each lookup (Figure 17(b)).

Overall, as shown in Figure 17(a), caching metadata on the local storage improves the throughput of both RocksDB and RocksMash. When there are no empty lookups (i.e., the empty lookup ratio is 0), the read performance of RocksDB is comparable to that of RocksMash, and this is due to the limited IOPS quota of purchased cloud storage. The poor performance of both RocksDB and RocksMash, however, suggests that improving hits in local storage or memory is critical. When the empty lookup ratio rises, so does the throughput. These performance improvements are due to the reduction of slow I/Os to the cloud, as illustrated in Figure 17(b). Caching metadata on local storage, whether via the RocksDB metadata cache or the RocksMash MetaCache, significantly reduces the I/O count to the cloud for each lookup. RocksMash sends fewer I/Os to the cloud than RocksDB as empty lookup ratios increase, resulting in a larger improvement for RocksMash. When there are only empty lookups (i.e., the empty lookup ratio is 100%), RocksMash has a 78% higher throughput than RocksDB. This is because an LSM store must go through all levels before correctly confirming a non-existent key. RocksMash sends no I/Os to the cloud for empty lookups, because MashMeta in MetaCache has a zero false-positive rate of probing. RocksDB reads more data blocks from the cloud on the contrary, because the false-positive rate of RocksDB metadata is greater than zero. In conclusion, caching metadata on local storage benefits both RocksDB and RocksMash, but MetaCache with MashMeta makes RocksMash more advantageous under empty lookups.

DataCache. We evaluate the DataCache cache hit ratio by comparing it against the persistent cache of RocksDB. We vary the cache size from 10% to 30% of the workload dataset size. We use db_bench to generate two different workloads: a uniform workload where each key–value item is equally accessed and a non-uniform workload with Zipfian distribution. Figure 17(c) shows the cache hit ratio under the workload with uniform distribution. For a cache size of 10%, the RocksDB persistent cache hit ratio is only 7%. DataCache hit ratio is 1.9× higher than RocksDB persistent cache, because DataCache has a higher space-efficiency and more popular data blocks can be cached. As the cache size increases to 30%, DataCache hit ratio increases to 31%, a 1.5× improvement over RocksDB persistent cache.

The cache hit ratio increases for both DataCache and RocksDB persistent cache when the workload is non-uniform (see Figure 17(d)), since the cache works better for skewed workloads. In this experiment, we test two more workloads by tuning the db_bench parameter read_random_exp_range (labeled as exp.). The larger the value of this parameter, the more skewed the read request distribution is. When the workload skewness is medium (i.e., exp. is 5), both DataCache and RocksDB persistent cache improve the hit ratio slightly (bars in black in Figure 17(d)). When we increase the workload skewness (i.e., exp. to 15), both cache methods improve the hit ratio significantly (bars in grey in Figure 17(d)), because the cache is able to preserve a higher number of frequently accessed data blocks compared to less skewed workloads. As RocksDB gathers key–value pairs to a cache file in their arrival order, it is difficult for the cache file level LRU policy to preserve hot data in the persistent cache. RocksMash DataCache manages data in the unit of a block, and the block level eviction policy is able to cache more hot data blocks in the persistent cache. As a result, RocksMash shows increased hit ratio improvements compared to uniform workloads.

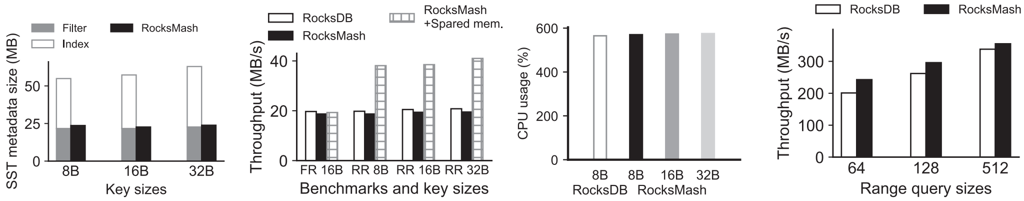
We add the evaluation of the read latency cost split-up in Figure 18 to address concerns about lookup latency within a DataCache bucket. RocksMash uses the bucket by default to reduce the number of elements searched in a linear manner. A bucket and a data block have default sizes of 8 MB and 4 KB, respectively. Each bucket can hold two thousand data blocks and has a maximum of two thousand elements in its LRU list. Because RocksMash caches all LRU lists in memory, linearly searching a LRU list of this magnitude is not as expensive as reading data blocks from local storage. As shown in Figure 18, accessing the local storage dominates the cost, whereas looking up in bucket LRUs costs less than 7%. As a result, LAP DataCache has a low lookup overhead.

To inspect conflicts within a bucket, we vary the workload locality and persistent cache size. The miss ratio of reading cached keys is used as the metric for LAP DataCache in this experiment. Reading cached keys in the LAP DataCache is expected to hit if there are few conflicts in buckets. As a result, the lower this metric is, the fewer the conflicts in buckets. First, we vary the persistent cache size to 10%, 20%, and 30% of the dataset, and then evaluate the miss ratio of reading cached keys in LAP DataCache. Figure 19 shows that skewing the workload distribution (i.e., exp. from 5 to 15) reduces the miss ratio dramatically. Because LAP DataCache distributes hot sstables to different buckets, conflicts between them are mitigated. When the size of the LAP DataCache is increased, the miss ratio decreases. This is primarily due to the capacity. Then, the persistent cache size is then set to 20% of the dataset, and the bucket size is varied. By default, we set the bucket size to 8 MB, and each bucket holds data blocks from several sstables. In this experiment, we reduce the number of buckets, increasing the size of each bucket to 40 and 120 MB, and each bucket can hold data blocks from more sstables. Figure 20 shows that, regardless of workload distribution, larger buckets slightly increase the miss ratio. Because there are more hot sstables in a bucket, the number of conflicts within a bucket increases. This insignificant increase in miss ratio, however, demonstrates that hashing sstables and distributing sstables to buckets is an effective way to avoid hot data block conflicts in LAP DataCache.

4.4 MashMeta

To evaluate the effects of the MashMeta, we use RocksMash that is only equipped with the MashMeta and compare against RocksDB. In this test, we store all the data in local storage to narrow down the variance between metadata structures and to accelerate experiments. We first evaluate the gain in memory space of the MashMeta, then we measure its effects on the write and read performance.

Metadata size. Figure 21(a) shows the total size of all sstable metadata after loading 10M random writes. Compared to the size of traditional block-based sstable metadata, the MashMeta reduces up to 62% of the metadata size and reduces filter false-positive rate to zero. As most traditional optimizations either focus on the index or the filter [48, 62], they cannot improve both of them



(a) The total size of all sstable metadata.

(b) Random write and read performance for varied key sizes.

(c) Average CPU usages of random write with varied key sizes.

(d) Range performance for varied range sizes.

Fig. 21. Effects of the MashMeta.

at the same time with a single structure, and they cannot achieve a false-positive rate of zero for filtering. While the proposed MashMeta halves the metadata size and at the same time enhances the filter functionality in reducing unintended I/Os.

With the MashMeta, RocksMash could save a significant amount of memory. This spared memory space could be used for the block cache. In Figure 21(b), RocksMash uses the spared 32 MB memory (as in Figure 21(a)) to compensate the table and block cache. The random read performance (shown as patterned bars) is therefore increased by 2× compared to RocksMash and RocksDB in cases where the memory is exhausted with no memory space for table and block cache.

Construction overhead and query performance. The MashMeta is built when generating sstables, and so, we use the FR workload to evaluate the overhead of constructing it. We use RocksDB as the comparison, because its index block is built with little overhead by picking up one key from every data block. We use the RR and SR workloads to evaluate the read and range query performance of MashMeta, and use RocksDB as the comparison. RocksDB uses binary searches within index blocks to locate a target data block in an sstable. In Figure 21(b), the FR performance of RocksMash is on a par with RocksDB. Under various key sizes, RocksMash and RocksDB have comparable CPU utilization, as shown in Figure 21(c). This indicates that the MashMeta does not bring significant overhead. This is because the data transfer dominates the compaction overhead and building the MashMeta for sstables only occupies a small fraction of time during compaction operations. Besides, the long shared prefixes among keys in bottom level sstables contribute to a reduction in computational overhead. The slight FR performance drop of RocksMash shows the moderate construction overhead, and it is due to encapsulating all keys into the MashMeta. The MashMeta also shows a moderate (less than 5%) RR overhead across different key sizes in Figure 21(b). However, considering the significant reduced metadata size (Figure 21(a)) and the benefits gained from leveraging the spared memory space for caching (Figure 21(b)), this construction and query overhead of MashMeta is tolerable. For range query performance, we vary range sizes and show the performance in Figure 21(d). RocksMash improves the range query performance for all range sizes, and the largest gain, by 1.2×, is achieved when the range size is 64. As the range size grows, the data block transfer overhead dominates and the benefits on throughput of the MashMeta gradually decreases.

4.5 Recovery Performance

In this part, we first assess whether putting the WAL on a cloud volume affects the write performance. After that, we analyze the proposed parallel recovery performance.

A write request first logs its data in the WAL. Even configured with the maximum provisioned performance [2] (IOPS 16,000 and throughput 1,000 MB/s at the cost of $119), the EBS-gp3 has a

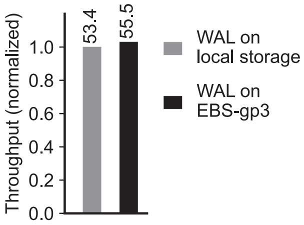


Fig. 22. Write performance of using the local storage or EBS-gp3 for WAL.

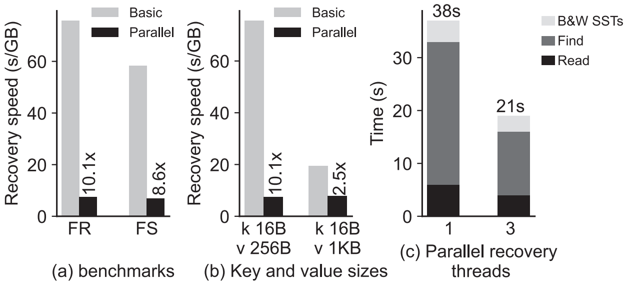


Fig. 23. Recovery speed. "FS" is fillseq. "Read" is reading WAL files; "Find" is sstable worker finding belonging key–value pairs; "B&W SSTs" is building and writing sstables.

much lower IOPS performance when compared to the local storage. So we evaluate whether using the EBS-gp3 for the WAL degrades the write performance. In this experiment, we configure L_0 and L_1 sstables in the local storage, and L_2 sstables in the EBS-gp2 for RocksMash. We set eight threads for db_bench to simulate heavy random write traffics. Figure 22 depicts RocksMash's random write throughput when using local storage or the EBS-gp3 for the WAL. The performance difference is due to two factors. First, for writing WAL of the vanilla RocksDB, the IOPS performance is insignificant. The RocksDB groups WAL writes by default [13, 23], which significantly reduces I/Os when writing key–value pairs to WAL files. We do not see any IOPS performance saturation when using either local storage or EBS-gp3 for WAL files. Second, local storage has a lower bandwidth than EBS-gp3. We evaluate the local storage's IOPS and bandwidth performance, and instance m5d.2xlarge limits its IOPS by 80 K and bandwidth by 260 MB/s. Even though local storage outperforms EBS-gp3 in terms of IOPS, its bandwidth is limited due to instance limitations. RocksMash allocates L_0 to L_1 levels on local storage; heavy write workloads cause constant compactions, occupying many bandwidths of local storage and throttling RocksMash's overall write performance. If RocksMash uses local storage with high IOPS and bandwidth, then we recommend using high-performance cloud storage for WAL files. We use EBS-gp3 for WAL in the following experiments.

Parallel recovery performance. Figure 23 shows the recovery speed in terms of the time spent per gigabyte of WAL files. The proposed parallel recovery method is ~8.6–10.1× faster than the basic recovery method for both FR and FS workloads (Figure 23(a)). This gain drops to 2.5× when the FR *value* size is larger (Figure 23(b)) because of faster L_0 to L_1 compactions. Figure 23(c) also shows how the parallelism (number of threads) affects the recovery performance. When RocksMash allocates three threads for recovery, the time for finding sstable belonging key–value pairs is shortened. Most of the recovery time is spent on finding key–value pairs for sstables (detailed in Section 3.4). Using three threads helps RocksMash reduce about 45% of the recovery time. This benefit is primarily due to improvements on preparing key–value pairs for sstables in parallel and thus reducing the time of putting key–value pairs to their corresponding sstables. In summary, the proposed parallel recovery significantly improves RocksMash's recovery performance by up to 10× compared to the basic recovery.

WAL sizes. The recovery performance largely depends on the size of WAL files. To evaluate how the preserved WAL size changes, we perform 10 M random writes to RocksMash and RocksDB, and configure TTL to 20 seconds (choice explained further below) for RocksMash and the default 60 seconds for RocksDB. Figure 24 shows the real-time total size of WAL files and upper level

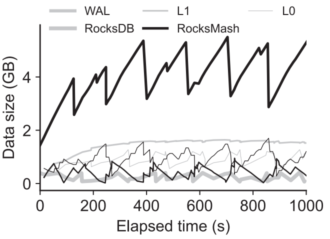


Fig. 24. Realtime sizes under random writes. RocksMash lines are in black, and RocksDB lines are in grey. Lines of different widths represent different LSM store data locations.

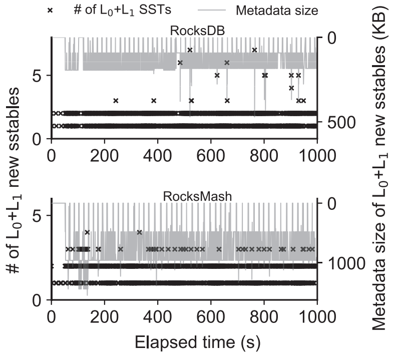


Fig. 25. The number of new L_0 and L_1 sstables and their metadata sizes.

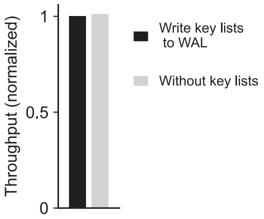


Fig. 26. Random write performance with and without key list writing to the extended WAL.

sstables L_0 and L_1. The real-time size shows the possible amount of data to be recovered. The WAL real-time size of RocksMash (black and bold line) fluctuates between 3 and 5 GB, which is always larger than the WAL size of RocksDB (grey and bold line). This is because RocksMash keeps all L_0 and L_1 key–value pairs in WAL files, while RocksDB only keeps mutable and immutable data in WAL files. Besides, data in the WAL is in chronological order while data in sstables is in key-order. This mismatch in the order scatters L_0 and L_1 data to different WAL files, which amplifies the number of WAL files that need to be preserved and makes the WAL sizes of RocksMash much larger than the sum size of these two level sstables. Note that, the data size of reserved WAL is at the size of about ten gigabytes, therefore it is usually three orders of magnitude smaller than user data. Besides, EBS-gp3 and EBS-gp2 have a similar storage cost, hence the WAL storage cost on EBS-gp3 will not cause many cost increases.

Figure 25 shows the effects of TTL on writing upper level sstable key lists to WAL files. Since RocksMash flushes L_0 and L_1 sstables to their next level more frequently than RocksDB, this reduces the number of L_1 sstables, and thus lowering the chances of heavy L_0 to L_1 compactions. In this case, L_0 to L_1 compaction jobs involve fewer sstables and no compaction jobs generate more than five new sstables in RocksMash. This lowers the number of written sstable key lists to WALs. The real-time total size of RocksMash new upper level sstable metadata (which includes their key lists) is larger than RocksDB's new upper level sstable metadata (which does not have key lists). This is because RocksMash adds key lists of these sstables to WALs. But writing key lists to WALs does not incur significant write traffic as shown in Figure 26. This is due to the fact that a compaction job is not considered complete unless all new sstables are successfully persisted, so RocksMash can batch key lists of new sstables to reduce the I/O number writing them. Second, the majority of compaction jobs involve multiple sstables [55], and Figure 25 shows that the data sizes of their key lists are typically less than 1,000 KB, implying that writing data of this magnitude to the WAL will not result in significant I/O pressures.

To evaluate the overhead of TTL induced compactions, Figure 27 shows the overall compaction data size after loading 10M random writes to RocksDB and RocksMash with different TTL. A shorter TTL helps RocksMash reduce the WAL file size. When we set TTL to 20 seconds, RocksMash has 8% more compaction read data and 9% more compaction write data compared to setting

	TTL	WAL size	Compact data (GB)	
	(s)	(GB)	Write	Read
RocksDB	20	0.16	11.2	10.1
	60	0.15	9.9	9.0
	100	0.15	10.3	9.3
RocksMash	20	3.3	12.5	11.6
	60	3.5	10.4	9.2
	100	5.2	10.7	9.5

Fig. 27. WAL file sizes and cumulative compaction data sizes under different TTLs.

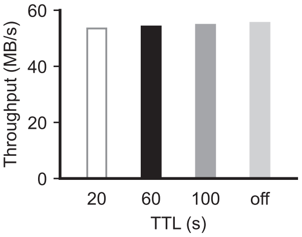


Fig. 28. RocksMash random write throughputs under different TTLs.

TTL to 60 seconds. As these upper level compaction operations read from and write to the local storage, the high performing local storage could mitigate these moderate overhead. In Figure 28, periodic flushing operations merely affect the write performance due to the following two reasons. First, the majority of L_0 to L_1 sstables have a lifetime of about 10 seconds due to write operations [26]. Because the time between two TTL flushing operations is two to 10 times longer than the lifetime of the majority of L_0 to L_1 sstables, extra compaction data sizes caused by periodic flushing operations are greatly reduced. Second, the time between two periodic flushing operations is sufficient, which reduces the number of periodic flushing operations and dilutes the impact of induced compactions even further. In summary, setting TTL to 20 seconds helps reduce the WAL size and does not incur significant compaction overheads.

5 RELATED WORK

LSM with heterogeneous storage. Heterogeneous storage provide different opportunities for LSM stores. The major motivation of adopting cloud storage services into LSM stores is cutting the overall storage cost. Mutant [61] prioritizes the overall storage cost rather than the performance, and takes advantage of workload localities to cache hot sstables for cloud storage. However, its coarse-grained caching methods is suboptimal for space efficiency on expensive storage. Aurora reduces network traffics for cloud databases [53]. The high performing Non-Volatile Memory is mostly used for improving the flush performance [39, 41, 60]. SpanDB [23] tiers LSM levels into the high performance SDPK-capable SSDs and capacity-advantaged SSDs, and it further exploits the parallelism of SPDK to improve WAL performance. PrismDB [14] uses CLOCK algorithm strive to pin hot objects to upper levels. However, the LSM-tree structure makes it challenging to efficiently pin the majority hot objects in small-sized upper levels.

LSM index optimizations. SlimDB [48] exploits the semi-sorted application scenarios and leverages the share key prefixes to reduce index sizes by the binary-trie based entropy-encode tries [43]. However, the entropy-encode trie is not directly applicable to string keys. LSM-trie [57] reduces read and write amplifications for small-sized value stores, and it uses hash keys to aggressively organizes all sstables as a single level. Jungle [15] uses B+ tree to index overlapped sstables in LSM that uses tiering merge policies, and this ensures one read to sstables to get the target data block. RemixDB [63] proposes a globally sorted view index structure REMIX to manage sstables, and this global view helps improve the range lookup performance. Bourbon [26] integrates learned index to LSM and uses it in place of sstable indices to improve read performance.

LSM filter optimizations. Stacked Filters [30] absorb frequently queried non-existing keys into the filter and compose a stacked filter structure to get lower false-positive rates and space overheads. Rosetta [45] puts a key's all binary prefixes into bloom filters and organizes them in implicit segment trees to improve point and range lookup performance. SuRF [62] proposes an optimized succinct tree based range filter that is tunable for space overheads or false-positive rates. MashMeta differs from SuRF in terms of the design philosophy, methods, and benefits. First, SuRF focuses solely on the filter, and even with more memory space, it cannot achieve a zero false-positive rate for filtering. Second, SuRF employs Level-Ordered Unary Degree Sequence [37], which differs from MashMeta's DFUDS and makes it difficult to encapsulate indexing information without incurring significant space overhead. Third, it does not reduce the index size, which is close to the size of the filter. Monkey [27] proposes to allocate more memory for upper level sstable filters to reduce their false-positive rates. ElasticBF [42] extends Monkey by dynamically changing sstable filter settings according their hotness rather than their locations.

LSM cache optimizations. AC-Key [56] aims at LSM cache mechanisms in the memory and hybrids different kinds of cache object and dynamically adjusts their sizes to improve cache efficiency. There are two differences between our work and AC-Key. First, AC-Key focuses on memory cache management optimization. Because it does not prioritize caching metadata for sstables, it is unable to efficiently eliminate metadata read I/Os to cloud storage. Second, the local storage uses block interfaces, so their benefits of caching small objects (e.g., key–value pointers) are diminished. Internal compactions cause non-trivial cache invalidations, which in turn harms the LSM read performance. LSbM-tree [52] uses a compaction buffer to minimize these cache pollutions. Leaper [59] leverages machine learning methods to predict hot data and actively prefetches them into the cache to ease this problem. Kassa et al. [40] extend the persistent cache with **Storage Class Memories (SCM)**. However, there are two significant differences with RocksMash. First, they focus on combining DRAM and SCM, but their design does not assign top-k level sstables to the high performance SCM, thereby missing out on the opportunity to saturate the high performance of SCM devices. Second, their work does not optimize the metadata of sstables, which takes up a significant amount of DRAM space.

SSD as the cache and general caching methods. A large body of work use SSDs as the cache for slower storage. These SSD caching methods either take advantage of the high random performance [19, 22, 28, 44] or absorb device and workload characteristics into cache managements [16, 50]. LeCaR [54] introduces machine learning methods to improve the cache replacement policy. Cacheus [49] extends LeCaR to automatically be more adaptive to various workloads.

LSM recovery approaches. MyRocks mirrors all sstables and WALs to cloud-managed storage by background copying [46]. However, writing sstables to cloud storage on a regular basis, however, will result in significant data traffic and additional fees [4]. Both Aurora [64] and Rockset [11] choose the WAL to transfer the most recent updates to other nodes, but details are not disclosed. RocksMash extends additional metadata for sstables and adds it to WAL for recovery, significantly improving recovery speed while consuming little storage space.

6 CONCLUSION

In this article, we examine how integrating cloud storage affects the performance of LSM stores. We found that the read performance is severely affected by the capped performance of cloud storage due to storage cost concerns. We also found that the key patterns in many applications could be leveraged to help with the LSM store read performance that uses cloud storage. Taking these findings into consideration, we have presented RocksMash, a fast and efficient LSM store that efficiently splits LSM-tree between local storage and cloud storage to achieve cost-effectiveness. Moreover, to reduce the memory footprint for metadata and improve read performance, RocksMash uses an LSM-aware persistent cache that stores metadata in a space-efficient way and caches popular data blocks by using compaction-aware layouts. Furthermore, RocksMash extended WALs for fast parallel data recovery. The evaluation results show that RocksMash delivers a better performance compared to RocksDB, high reliability, and cost-effectiveness.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] [n. d.]. Amazon EBS Pricing. Retrieved from https://aws.amazon.com/pricing.

[2] [n. d.]. Amazon EBS volume types. Retrieved from https://aws.amazon.com/ebs/volume-types/?nc1=h_ls.

[3] [n. d.]. Amazon EC2 instance store. Retrieved from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html.

[4] [n. d.]. Amazon EC2 On-Demand Pricing—Data Transfer. Retrieved from https://aws.amazon.com/cn/ec2/pricing/on-demand/.

[5] [n. d.]. CockroachDB: Data mapping between the SQL model and KV. Retrieved from https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[6] [n. d.]. LevelDB. Retrieved from https://github.com/google/leveldb.

[7] [n. d.]. Memory usage in RocksDB. Retrieved from https://github.com/facebook/rocksdb/wiki/Memory-usage-in-RocksDB.

[8] [n. d.]. Netflix on AWS. Retrieved from https://aws.amazon.com/solutions/case-studies/netflix.

[9] [n. d.]. Persistent Read Cache. Retrieved from https://github.com/facebook/rocksdb/wiki/Persistent-Read-Cache.

[10] [n. d.]. RocksDB. Retrieved from https://github.com/facebook/RocksDB.

[11] [n. d.]. RocksDB-Cloud: A Key-Value Store for Cloud Applications. Retrieved from https://github.com/rockset/rocksdb-cloud.

[12] [n. d.]. TiDB Computing. Retrieved from https://docs.pingcap.com/tidb/v5.0/tidb-computing.

[13] [n. d.]. Write Ahead Log File Format. Retrieved from https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-File-Format.

[14] 2020. PrismDB: Read-aware log-structured merge trees for heterogeneous storage. arXiv:2008.02352. Retrieved from https://arxiv.org/abs/2008.02352v2.

[15] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. 2019. Jungle: Towards dynamically adjustable key-value store by combining LSM-tree and copy-on-write B+-tree. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19).*

[16] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. 2016. CloudCache: On-demand flash cache management for cloud computing. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. Santa Clara, CA.

[17] 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 USENIX Annual Technical Conference (ATC'19).*

[18] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[19] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.* 3, 2 (2010), 1435–1446.

[20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'20).*

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementati1on (OSDI'06).*

[22] Feng Chen, David Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing (ICS'12).*

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

[24] 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 (SoCC'10).*

[25] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The snowflake elastic data warehouse. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data (SIGMOD'16).* San Francisco.

[26] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (USENIX OSDI'20)*. Virtual.

[27] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD'17)*. Chicago, IL.

[28] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.* 3, 2 (September 2010), 1414–1425.

[29] DeCandia, Giuseppe and Hastorun, Deniz and Jampani, Madan and Kakulapati, Gunavardhan and Lakshman, Avinash and Pilchin, Alex and Sivasubramanian, Swaminathan and Vosshall, Peter and Vogels, Werner. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th Symposium on Operating Systems Principles (SOSP'07)*.

[30] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2021. Stacked filters: Learning to filter by structure. *Proc. VLDB Endow.* 14, 4 (2021), 600–612.

[31] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. 2017. Optimizing space amplification in RocksDB. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'17)*. Chaminade, CA.

[32] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'21)*. Virtual.

[33] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. 2008. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th ACM Symposium on Principles of Database Systems (PODS)*.

[34] Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. 2021. An in-depth study of correlated failures in production SSD-based data centers. In *Proceedings of the 19th Conference on File and Storage Technologies (FAST'21)*. Virtual.

[35] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084.

[36] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 38th ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*.

[37] Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS'89)*.

[38] Guy Joseph Jacobson. 1989. *Succinct Static Data Structure*. Ph. D. Dissertation. Carnegie Mellon Univesity.

[39] 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 USENIX Annual Technical Conference (ATC'17)*. Boston, MA.

[40] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving performance of flash based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. 1–18.

[41] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. 2020. HiLSM: An LSM-based key-value store for hybrid NVM-SSD storage systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*.

[42] YongKun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proceedings of the Annual Technical Conference (ATC'19)*.

[43] Hyeontaek Lim, Bin Fan, David G. Andersen, and Micheal Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'11)*.

[44] Xin Liu and Kenneth Salem. 2013. Hybrid storage management for database systems. *Proc. VLDB Endow.* 6, 8 (June 2013), 541–552.

[45] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the Annual Conference on Special Interest Group on Management of Data (SIGMOD'20)*.

[46] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving facebook's social graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230.

[47] 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 Symposium on Operating Systems Principles (SOSP'17)*.

[48] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB'17)*.

[49] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning cache replacement with cacheus. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'21)*.

[50] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2017. DIDACache: A deep integration of device and application for flash based key-value caching. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'17)*.

[51] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a cloud DBMS: Architectures and tradeoffs. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB'19)*.

[52] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'17)*.

[53] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the ACM Annual Conference on Special Interest Group on Management of Data (SIGMOD'17)*.

[54] Vietri, Giuseppe and Rodriguez, Liana V. and Martinez, Wendy A. and Lyons, Steven and Liu, Jason and Rangaswami, Raju and Zhao, Ming and Narasimhan, Giri. 2019. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*.

[55] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*.

[56] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and Du. David H.C. 2020. AC-key: Adaptive caching for LSM-based key-value stores. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*.

[57] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*.

[58] Peng Xu, Nannan Zhao, Jiguang Wan, Wei Liu, Shuning Chen, Yuanhui Zhou, Hadeel Albahar, Hanyang Liu, Liu Tang, and Changsheng Xie. 2021. Building a fast and efficient LSM-tree store by integrating local storage with cloud storage. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'21)*.

[59] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proc. VLDB Endow.* 13, 12 (2020), 1976–1989.

[60] 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 USENIX Annual Technical Conference (ATC'20)*.

[61] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in LSM-tree data stores. In *Proceedings of the IEEE International System-on-Chip Conference (SoCC'18)*.

[62] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the ACM Annual Conference on Special Interest Group on Management of Data (SIGMOD'18)*.

[63] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient range query for LSM-trees. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. Virtual.

[64] Alexandre Z. Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon aurora: On avoiding distributed consensus for I/Os, commits, and membership changes. In *Proceedings of the ACM Annual Conference on Special Interest Group on Management of Data (SIGMOD'18)*.

Received October 2021; revised January 2022; accepted March 2022