LSM-Tree Managed Storage for Large-Scale Key-Value Store

Fei Mei, Qiang Cao, *Senior Member, IEEE*, Hong Jiang, *Fellow, IEEE*, and Lei Tian, *Member, IEEE*

Abstract—Key-value stores are increasingly adopting LSM-trees as their enabling data structure in the backend block storage, and persisting their clustered data through a block manager, usually a file system. In general, a file system is expected to not only provide file/directory abstraction to organize data but also retain the key benefits of LSM-trees, namely, sequential and aggregated I/O patterns on the physical device. Unfortunately, our in-depth experimental analysis reveals that some of these benefits of LSM-trees can be completely negated by the underlying file level indexes from the perspectives of both data layout and I/O processing. As a result, the write performance of LSM-trees is kept at a level far below that promised by the sequential bandwidth offered by the storage devices. In this paper, we address this problem and propose LDS, an LSM-tree Direct Storage system that manages the storage space based on the LSM-tree objects and provides simplified consistency control by leveraging the copy-on-write nature of the LSM-tree structure, to fully reap the benefits of LSM-trees. Running LevelDB, a popular LSM-tree based key-value store, on LDS as a baseline, comparing that to LevelDB running on three representative file systems (ext4, f2fs, btrfs) with HDDs and SSDs, respectively, we evaluate and study the performance potentials of LSM-trees. Evaluation results show that the write throughputs of LevelDB can be improved by from 1.8× to 3× on HDDs, and from 1.3× to 2.5× on SSDs, by employing the LSM-tree friendly data layout of LDS.

Index Terms—LSM-tree, key-value store, file system performance, application managed storage

1 Introduction

Log-Structured Merge-trees (LSM-trees) have been applied to both local and distributed environments for large-scale key-value stores, such as LevelDB [1], RocksDB [2], HBase [3], BigTable [4], Cassandra [5], PNUTS [6], InfluxDB [7], etc., because LSM-trees are capable of performing sequential writes to persistent storage, which is the best expected access pattern for both hard-disk drives and solid-state devices [8], [9], [10], [11]. To benefit from these potential advantages of LSM-trees, other popular database systems that traditionally organize data in B-trees have also begun to use LSM-trees in their new releases, such as MongoDB [12] and SQLite4 [13].

Ideally, an LSM-tree expects to store its data object to the underlying block storage contiguously, but an intermediate block manager, such as a file system, can prevent this from happening by virtue of block management indexing, as demonstrated by an example depicted in Fig. 1. Usually a block manager stores the management index in the forms of object location metadata (e.g., file inodes) and resource allocation map, to respectively locate the object data blocks and find free blocks for storing the object data. Maintaining these index blocks incurs more non-sequential I/Os, which harms the performance for both HDDs and SSDs [8], [9], [10], [11], [14], [15]. Moreover, all these index blocks must be updated synchronously with the data blocks for data consistency, which requires significant extra work to carry out [16], [17], [18], [19], [20], [21].

To address these problems and fully reap the benefits of LSM-trees, we present LDS, an LSM-tree conscious key-value storage system that maps the LSM-tree data directly onto the block storage space to preserve the intended sequential write access patterns, and manages data consistency by exploiting the inherent index mechanism and the copy-on-write nature of the LSM-tree structure to avoid the overhead associated with consistency enforcement. As a result, LDS completely eliminates the complicated and expensive file-level index operations in storing LSM-tree data, and substantially reduces the number of I/Os and strongly preserves the disk write sequentiality. These advantages of LDS are applicable to both HDDs and SSDs.

We implement a prototype of LDS based on LevelDB to assess the benefits of directly storing the LSM-trees, in comparison with storing the LSM-trees through three representative file systems with different block management stategies on the data layout and I/O processing, namely, ext4 (update-in-place) [22], f2fs (log-based without wander-tree updating) [23], and btrfs (B-tree structured and copy-on-write) [24]. Experimental results show that LDS consistently and substantially improves the write performance under all workloads examined. On HDDs, the write throughput is improved by at least 1.8×, and up to 3×. On SSDs, the write throughput is improved by at least 1.3×, and up to 2.5×. The read performance also benefits from the LDS design due to the shortcut in indexing the LSM-tree data.

The rest of the paper is organized as follows. In Section 2 we present the background and analysis of LSM-trees

- *F. Mei and Q. Cao are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {meifei, caoqiang}@hust.edu.cn.*
- *H. Jiang is with the University of Texas at Arlington, Arlington, TX 76019. E-mail: hong.jiang@uta.edu.*
- *L. Tian is with Tintri, Inc., Mountain View, CA 94043. E-mail: leitian.hust@gmail.com.*

Manuscript received 18 Nov. 2017; revised 8 July 2018; accepted 25 July 2018. Date of publication 7 Aug. 2018; date of current version 16 Jan. 2019. (Corresponding author: Qiang Cao.) Recommended for acceptance by F. Qin. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2864209

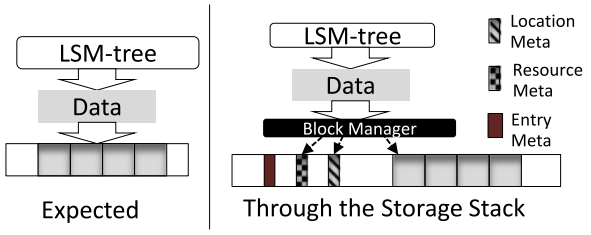


Fig. 1. An example of how a sequential pattern on the block storage expected from LSM-tree (left) is broken up by a block manager (right).

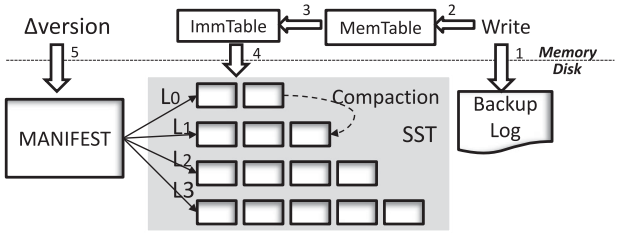


Fig. 2. LevelDB implementation of LSM-tree. The in-memory component, C_0, is composed of two sorted skiplists (the MemTable and ImmTable), and each on-disk component (C_1, C_2, etc.) is a level (L_0, L_1, etc.). There are three types of disk writes throughout the runtime, as indicated in the figure: write-ahead log for backing up the memory tables (Step 1); writing the newly generated chunks (SSTs) in compaction (Step 4); and updating the LSM index when compaction finishes (Step 5).

running on file systems to motivate our LDS research. The design and implementation of LDS are detailed in Section 3. We evaluate LDS in Section 4. Related work is presented in Section 6. Finally, we conclude our work in Section 7.

2 Background and Motivation

2.1 LSM-Tree and LevelDB

A standard LSM-tree comprises a series of components C_0, C_1,..., C_K with exponentially increasing capacities [14], where C_0 resides in memory while all other components reside on disk, and all the keys in a component are kept sorted for fast retrieval. Every time the amount of data in C_x reaches its capacity limit, the component's data will be rolling merged to C_{x+1}.

To amortize the cost of merge, state-of-the-art LSM-tree based key-value stores split each on-disk component into *chunks* [1], [2], and only partially merge a component (one or several chunks) when its size reaches the limit, a function called *compaction*.[1] Specifically, a compaction process on C_x will select a target chunk there in a round-robin manner [1], and merge it into C_{x+1}. All chunks in C_{x+1} overlapped with the target chunk are read out to participate in the merge sort process. After the compaction, new chunks generated from the merge are written to C_{x+1} and obsolete chunks participating in the merge will be deleted. The chunk write operation can be regarded as a copy-on-write process from the LSM-tree's viewpoint: a copy of each of the key-value pairs in a new chunk exists in the obsolete chunks. In other words, interrupted write operations of chunks can be recovered and redone, because the original chunks are not to be deleted before all the new chunks have been safely persisted. Besides the chunk writes, LSM-tree generates write-ahead logs to an on-disk backup for the in-memory component C_0.

Let us take LevelDB, a widely used LSM-tree based key-value store implementing the partial merge introduced above, as a concrete example. Fig. 2 demonstrates the structure of LevelDB, in which C_0 consists of two sorted skiplists (the MemTable and ImmTable), and each on-disk component is referred to as a level ($L_0 \sim L_3$ in the figure) that contains multiple chunks (sorted string tables, or SSTs). An SST includes a body of sorted key-value pairs and a tail that indexes a read request to the body. Decoding the tail always begins from its last bytes (the *footer* in the LevelDB terms). A write request is first appended to the *Backup Log* and then inserted to the *MemTable*, which will be marked as immutable (*ImmTable*) if its size reaches its capacity limit. Compaction on C_0 dumps the *ImmTable* onto disk as an L_0 chunk, and compaction on an on-disk component L_n merges one of its SST to L_{n+1}. A separate structure is maintained to keep track of the metadata of all the SSTs, called *LSM index* (i.e., the version) that is backed by the *MANIFEST*. Each SST has a unique ID that is recorded in the *MANIFEST* along with its metadata. A compaction process that changes a level's structure must update the *MANIFEST*, which is also implemented in a logging manner, called a *version edit* or Δ*version*.

2.2 LSM-Tree on File Systems

Generally, local LSM-tree based key-value stores, such as LevelDB, persist data to the storage through a file system, referred to as LSM-on-FS in this paper. In an LSM-on-FS implementation, all data (e.g., chunk, log, etc.) are stored in the form of files. Intuitively, file systems are supposed to enable LSM-tree data to be stored in large, sequential I/Os, a desirable property for the low-level storage devices such as HDDs and SDDs. Unfortunately, such expected large, sequential I/Os are actually broken into non-sequential, small I/Os due to the need to access the file system index (*FS index*), as explained next.

Generally, FS index includes the file metadata (e.g., inode) and the resource allocation map (e.g., bitmap), which are all stored in the file system blocks of a fixed size (e.g., 4 KB). Updates to an LSM chunk file or LSM index file come into force by non-sequential, small writes to update the FS index, causing various degrees of I/O amplification, i.e., increased number and intensity of I/Os that are often small and non-sequential, depending on the nature of a given file system.

First, for an update-in-place file system, such as ext4, the metadata of the files are usually aggressively stored in one place, and the resource allocation map is stored in a different place, while the user data are stored in other places where sufficient free space is available, within or without a group ("block group" in ext4), as indicated in Fig. 3 (the upper one). Writes to the data blocks of a file inevitably lead to I/Os for updating the FS index. Failed writes in a crash can bring the file system to an inconsistent status [25], resulting in space leakages or file corruptions. To maintain the data consistency, update-in-place file systems usually employ a journal to obtain atomic updates to the inodes and bitmap. That is, writing the journal becomes an integral part of and in addition to the usual FS index updates. Therefore, we regard the journal also as FS index for such file systems. Since the update-in-place file systems store the user data and the FS index in separate places, the amplified I/Os are usually random accesses to the storage device.

1. In a special case, a target chunk in component x is directly pushed to component $x + 1$ or lower components (including from the in-memory component to the first on-disk component) without the actual merge sort. This is also regarded as a compaction because at least one component has its structure changed.

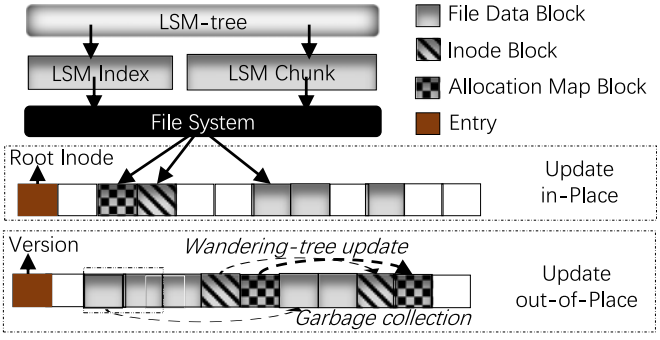


Fig. 3. Both the LSM index and LSM chunk are re-organized as file data and file index through a file system, be it an update-in-place file system (upper) or an update-out-of-place file system (lower).

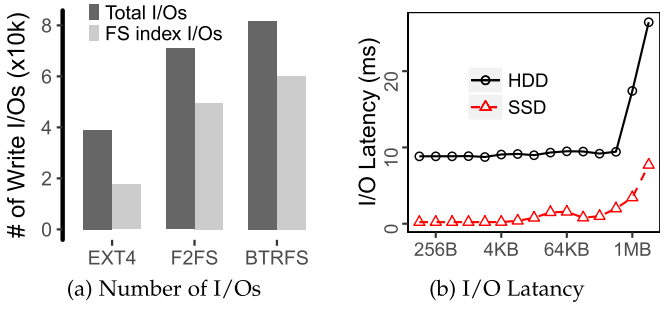


Fig. 5. (a) I/O distributions when storing LSM-trees through three representative types of file systems. (b) I/O latencies of persisting different sized requests on HDD and SSD.

Second, for a log-based file system [26], an instance that updates out of the place, although writes for updating the FS index can be contiguous to writes for updating the user data, it must update the entire metadata link from the root inode to the user data as well as the allocation map, a phenomenon known as the *wandering-tree update* [23], [24], [26]. In addition, measures must be taken to reclaim the obsolete blocks, a process known as *garbage collection*. How to implement an efficient garbage collection is crucial for a practical log file system, especially in an LSM-tree environment that produces a very high volume of garbage in the compaction process. In fact, we have experimented on running LevelDB on NILFS2 [27], a log-structured file system implementation on Linux, and found that after writing data in the amount of only one-tenth the size of the file system's volume, the file system ceases to work because no space is left. F2fs [23] is a practical log-structured file system that resolves the wandering-tree update problem by introducing a *Node Address Table* for the metadata blocks and storing the resource allocation map in an update-in-place manner, at the cost of losing the benefits of logging feature and resulting in more random I/Os. Another problem of the log file system is that the clustered user data (such as LSM-tree log) can be fragmented across the storage space [28].

The third type of file systems are the copy-on-write (CoW) file systems, another instance that updates out of the place, such as btrfs [24], which also update the index in a wandering-tree fashion, except that they do not guarantee that the updates are physically contiguous.

In both types of the update-out-of-place file systems, as the FS index is written to a new place in each update, an anchor in a definite place must be periodically updated to record the latest location of the FS index in order not to lose track of the updated data. Each update to the anchor represents a new version of the file system, as depicted in Fig. 3 (the lower one). Successful updates to the file data but without checking the version can result in an actual failed file update if a crash happens. Accordingly, we regard the anchor also as an integral part of the FS index for update-out-of-place file systems.

As introduced in the previous subsection, an LSM-tree has its own index to locate and describe the data chunks. When running on a file system, the LSM index and the LSM chunks are all organized in files by the underlying file system. A single update to an LSM-tree chunk in effect entails multiple physical updates (write I/Os) of the following two types: (1) updates to FS blocks for the LSM-tree chunk data (e.g., 4 I/Os for a 4 MB chunk); (2) updates to FS blocks for the FS index (at least 2 I/Os, depending on the file system organization). The same processes are repeated in updating the LSM index, as: (1) updates to FS blocks for the LSM index; and (2) updates to FS blocks for the FS index. Fig. 4 illustrates the write patterns of LSM-tree through the three representative file systems and LDS respectively. The experiments for this figure are run with the backup log disabled under sequential workloads. It shows that there is a significant amplification of the numbers of I/Os in that the expected large, sequential write I/Os from LSM-trees are actually converted into larger numbers of small, and potentially non-sequential write I/Os on the storage device through file systems.

To provide additional insight, Fig. 5a shows how large the fractions of total I/Os are actually FS index I/Os when LSM-trees run through the three representative file systems. We also show in Fig. 5b the I/O latencies of persisting different sized requests on raw HDD and SSD devices. In our experiments for Fig. 5a, we identify the FS index I/Os by analyzing the block trace results of sequentially writing the LevelDB with the backup log disabled, so that only chunk files and the MANIFEST file are persistently updated. For

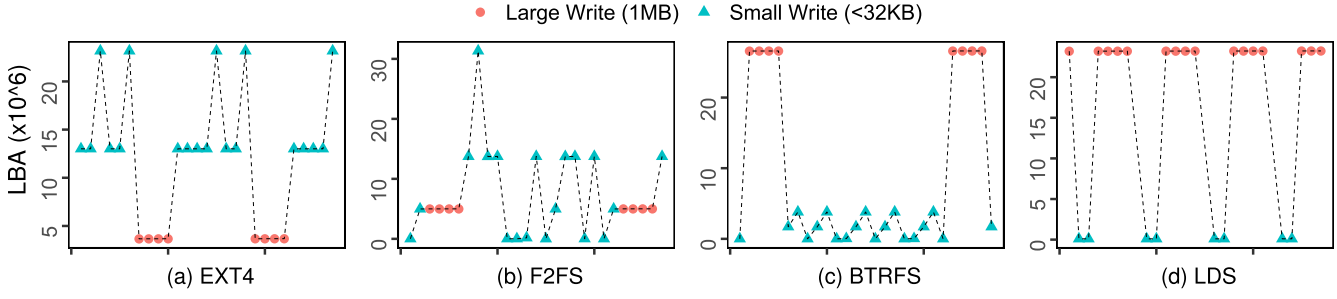
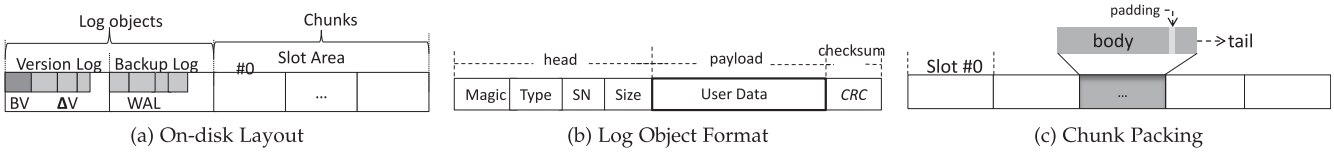


Fig. 4. Figures (a), (b), and (c) show the write patterns of LSM-trees through three representative file systems, in which the large writes are updates to the LSM chunks while the small writes are updates to the FS index and LSM index. Figure (d) shows that the FS index writes are completely eliminated by LDS, leaving on the LSM index writes.



(a) On-disk Layout (b) Log Object Format (c) Chunk Packing

Fig. 6. On-disk layout of LDS for the LSM-tree data and the internal data format.

Fig. 5b, each result is obtained by sending a sequence of *write-fsync* request pairs to the raw device sequentially for ten seconds, and the average response time of requests is used as the latency measurement. The experiment results clearly show that *file-system induced index I/Os, while small in size, far outnumber the actual user data I/Os and substantially degrade the performance of LSM-tree applications*.

With a larger write buffer, applications can accordingly adopt larger chunks that incur lower ratio of FS index I/Os in the LSM-on-FS implementations, hence the amount of improvement of LDS over the LSM-on-FS implementation can be degraded. Nevertheless, LDS still exhibits considerable improvements over the LSM-on-FS implementation in a wide range of chunk size, as will be presented in Section 4.2.4.

2.3 Motivation

The above analysis of LSM-trees running on file systems reveals a significant disconnect and discrepancy between LSM-tree's intended sequential, aggregated write I/Os and the resulting file-system non-sequential (random), small write I/Os due to the two-level indexes (LSM index and FS index). This discrepancy not only adversely impacts the performance of LSM-trees, but also harms the performance and endurance (e.g., SSDs) of the underlying storage devices in the stressful LSM-tree environments.

One of the benefits of file system is its support of objects (files) with dynamically varying sizes, an abstraction that enables a database store to provide whatever higher level objects it wishes [29]. However, with the popularity of key-value stores and the wide utilization of LSM-trees, the uniform-size and immutable data objects from the LSM-tree applications benefit little from the file system abstraction while still suffering from overheads of the file system indexes. Therefore, we propose and study the LSM-tree Directly managed Storage, LDS, to understand the overheads induced by the file system, so as to develop a high-performance and reliable LSM-tree storage system.

3 Design and Implementation

The purpose of LDS is to perform direct mapping between LSM-tree data and its physical location to eliminate extra FS indexes. While the typical on-disk structure of LSM-tree has been illustrated in Section 2.1 (Fig. 2), in this section we introduce how to map the LSM-tree objects (i.e., Backup Log, Version/MANIFEST, SST Chunk) to the storage space and maintain the consistency enforcement.

3.1 Disk Layout

To perform direct mapping, LDS manages the storage space based on the LSM-tree data objects, instead of dividing the storage space to unified size (e.g., 4 KB block in common file systems). As illustrated in Fig. 6a, the entire volume is divided into three areas: version log area, backup log area, and slot area. All the slots in the slot area have identical size and are numbered sequentially by their offsets (slot IDs) to the first slot. A slot can contain an LSM-tree chunk that also has an ID (chunk ID) derived from the offset, thus LDS can immediately locate the physical position of a given chunk. The two log areas contain continuous log objects with the legal format illustrated in Fig. 6b, in which the *magic* and *CRC* fields are used to ensure the integrity of the log object, the *type* and *SN* (sequence number) fields are used to identify live objects in recovery (described in Section 3.3), and the *size* field tells how many bytes the payload contains. Live objects are objects that should be taken into consideration in a recovery. The opposite are the obsolete objects that should be ignored.

The version log consists of two main types of objects. (1) *Base Version* (BV) contains a complete description of the LSM-tree at the time the base version is generated: the metadata of all the chunks etc. A chunk's metadata includes the chunk ID, the level it belongs to, the smallest and largest keys of the chunk. Besides, LDS replicates the storage format information in the base version. (2) Δ*Version* (ΔV) describes the result of a compaction, e.g., the obsolete chunks that should be deleted and the new chunks that should be added. All the Δversions and their corresponding base version can be merged to a new base version, a process called *trim*. The backup log mainly contains the *Write-Ahead-Log* (WAL) objects that provide a backup for the in-memory key-value pairs that have not been persisted to the on-disk structure (i.e., MemTable and ImmTable in Fig. 2).

The slot area stores the LSM-tree chunks, and each for one chunk. As introduced before, a chunk consists of a body that contains sorted key-value pairs, and a tail section to index the requested keys in the body. Since LSM-trees cannot always generate a chunk with the exact length of its hosting slot, the body and tail of a chunk may not fully occupy a slot, a small padding area is used to make the tail right-aligned with its slot boundary, as shown in Fig. 6c, therefore right end of the tail can be immediately located.[2] This storage solution does not cause external fragmentation in the storage space and deleting a chunk immediately frees its hosting slot for re-allocation. Thus, LDS does not have the garbage collection and defragmentation problems of general CoW systems [23], [24]. However, the padding area within a slot can lead to *internal fragmentation*, which is discussed in Section 3.4.

Based on the above on-disk layout, we further explain how a compaction operation from the LSM-tree application (i.e., LevelDB) works in LDS. Assume that the compaction operation will merge a chunk with ID 100 (chunk-100) from L_i to L_{i+1}, and three chunks with IDs 200, 201, 202 respectively from L_{i+1} are overlapped with chunk-100. LevelDB performs the compaction by reading the key-value pairs from the four chunks and merge-sorts them to generate new chunks. Specifically, to read the key-value pairs of chunk-100, LDS first locates the hosting slot with the chunk ID (one-to-one mapping), and then decodes the chunk's internal index from the

2. Searching a key in a chunk begins from the last bytes of the tail, i.e., the SST footer in LevelDB.

last byte stored at the end of the slot. The key-value pairs of the chunk are read according to the index. When LevelDB generates a new chunk, LDS allocates a free slot for this chunk. The ID for the new chunk is derived from the allocated slot (in this case, assume that four new chunks with IDs 500, 501, 502, 503 are generated). Finally, after all the new chunks are stored, a Δversion is committed, which indicates that (1) chunk-100 from L_i, chunk-200, chunk-201, chunk-202 from L_{i+1} are deleted, and (2) chunk-500, chunk-501, chunk-502, chunk-503 are added to L_{i+1}.

3.2 Consistency Enforcement

LDS leverages the copy-on-write nature of LSM-trees to maintain the data consistency.

A complete LSM-tree includes the version (LSM-tree index), the backup log (on-disk backup of the in-memory LSM-tree component), and the chunks (self-indexed key-value groups). A version represents a snapshot of the backup state and the chunk organization, which changes after each compaction process. When a compaction finishes, the metadata of the chunks that participate in the compaction should be deleted from the version, and the metadata of the chunks generated from the compaction should be added to the version. The version is not updated in-place, instead, its change (Δversion) is committed to the version log and a group of Δversions with the base version can be merged to generate a new base version. Recall that the trivial move of a chunk from one level to another is also regarded as a compaction operation, including the memory table dump. If the memory table is compacted to a disk chunk, LDS resets the backup log by recording the start point of live objects in the version that represents this memory compaction. Only a successfully committed Δversion validates the compaction result. Otherwise, any result of the compaction is discarded as if nothing had happened. Since the new chunks are always written in free slots, a corrupted compaction has no impact on the original data. For the memory compaction, as the reset start point of the backup log is recorded in the Δversion, a failed Δversion simply cancels that reset. In other words, the committing of a Δversion ① deletes the old chunks from the version, ② adds the new chunks to the version, and ③ resets the start point of the backup log, in an atomic way.

Slots usage status (allocated or free) can be obtained by inspecting the version. On a restart the recovery process will construct a bitmap online when executing the *trim* for tracking the slots usage. We refer to this bitmap as *online-map*, to distinguish it from the traditional bitmap that is separately stored in the persistent storage and must be regarded for consistency control. Allocating slots in the runtime immediately flips their status in the *online-map* (from free to allocated) to prevent them from being allocated again. However, only a successfully committed Δversion will maintain the flipping outcome if a crash happens. Flipping a slot's status from allocated to free to delete a chunk must be performed after the Δversion is committed; otherwise, data can be corrupted because the LSM-tree may be recovered to the previous committed Δversion, while the slot has been re-allocated to store the wrong data. For example, considering a slot allocated in Δversion x, and freed in Δversion y, if the slot's status in the *online-map* is flipped to free before Δversion y is committed, it is possible that the slot is allocated to store a new chunk when a crash happens. As a result, on a restart the system recovers

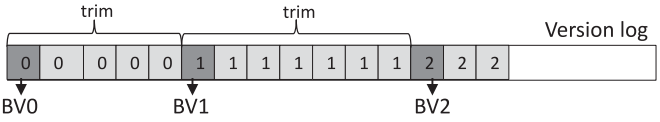


Fig. 7. Trim to generate a new base version. The Δversions share the same SN with their base version.

the slot to Δversion x but the slot has stored the wrong data. On the other hand, not freed slots in the online-map when crash happens never lead to space leak since the trim process in the recovery will construct the online-map according to the committed version.

If a compaction generates new chunks, LDS must allocate free slots from the slot area to persist these chunks. The allocation can be implemented in any way that is based on the online-map. A new chunk will be assigned an ID according to the offset of its allocated slot, so that from the chunk ID recorded in the version we can directly know the chunk's hosting slot. The default allocation implementation in LDS is similar to the threaded logging in LFS [23], [26], but with the slot as the primary unit. That is, LDS always advances in one direction in scanning the online-map for free slots and wraps around when the end is reached. If a complete round of a scan fails to find sufficient free slots, a "space full" status is reported.

3.3 Log Write

Logging is an important component in an LSM-tree. Besides the backup log for supporting the memory tables, the version is also updated in a logging manner (i.e., Δversion). In this subsection we introduce the efficient log mechanism in LDS and how to recover a consistent state from a crash with the logs.

3.3.1 Strict Appending and Crash Recovery

The version log and backup log are both used in a cyclical way, a common usage for logging or journaling. A log stored via a file system generally must update file index after an appending operation. For the file system itself that adopts a log, e.g., journaling in ext4, a super block is set at the beginning of the journal area (journal super) [25] and is updated afterwards to identify live and obsolete journal items. In contrast, LDS updates the log with only one physical appending operation, without the need to update any other identification data. LDS achieves this by using some special fields in the log object, such as *type* and *SN* (Fig. 6b), to identify live objects.

For the version log, the latest base version is used as an separation for live and obsolete objects, as shown in Fig. 7. A base version is generated each time when the trim process is performed. We now assume that the version area begins with a legal object[3] and no wrapping-around happens. In a recovery, LDS scans all the objects in the version log area from the beginning until garbage (i.e., illegal format) or SN value smaller than the latest scanned one is encountered, and identifies the latest base version (with the largest SN value) and all its subsequent Δversions that have the same SN value, to recover the version structure and online-map. For the backup log, as has been introduced in Section 3.2, the start point of the live objects can be obtained from the

3. A data stream is regarded as a legal object if it matches the format shown in Fig. 6b.

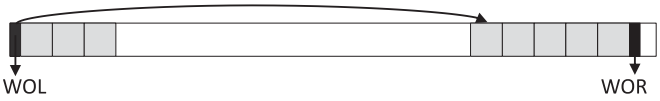


Fig. 8. The wrapping object at the left (WOL) and the wrapping object at the right (WOR). chunk size.

version. Recovering the memory table is achieved by scanning the backup log from the start point.

In practice, both the version log and the backup log will wrap around when the remaining space in the end of their log area is not enough to hold the requested object. To make the wrapping status identifiable to the recovery process, LDS introduces two special objects. As shown in Fig. 8, one is appended adjacently to the last object when the log wraps, called the *wrapping object at the right* (WOR), and the other is placed at the beginning of the log area when the log wraps, called the *wrapping object at the left* (WOL). The WOL includes a pointer to the first live object when the log wraps around, while the WOR is only a boundary identifier that informs the recovery process to return to the beginning of the log area. In practice, for the version log only the WOL takes effect since LDS always scans objects from the beginning of the log area and it only needs a trace of the right-end live objects, while for the backup log only the WOR takes effect because the Δversion has specified the start point but it needs to know the wrapping boundary if the log has wrapped. Nevertheless, LDS does not immediately turn to the right portion pointed by the WOL when scanning the version log, because they probably have been obsolete (by the *trim* process). As the WOL has the same SN as the right-end object when the log wrapped around, LDS knows that the right-portion objects are obsolete if a base version following the WOL with a larger SN is encountered. LDS turns to the right portion if it does not find a live base version from the legal objects in the left: this implies the objects in the left are all live Δversions and their base version exists in the right portion. The size of the wrapping objects are fixed to be the sector size of the block device, ensuring that appending the wrapping object is an atomic operation [30]. With the wrapping objects, the log area always starts with a legal object, enabling the recovery to proceed correctly.

Since the log areas are generally not large in size, scanning the log area will not be as costly as one may think. We evaluate the recovery cost in Section 4.4.

3.3.2 Commit Policy

In LDS, each compaction result (i.e., Δversio) is committed to the version log to provide a consistent state of the latest on-disk structure, while the Write-Ahead-Logs (WALs) are committed to the backup log for the purpose of recovering the in-memory key-value pairs. Not-committed Δversion before a crash would invalidate all work the compaction process has done, and delayed committing prevents other compactions from operating on the chunks related to this not-committed Δversion. In LevelDB, the Δversion is committed immediately.

On the other hand, not-committed WALs of recent insertions before a crash will cause the insertions lost. However, committing each WAL is extremely expensive because the I/O latency is several orders of magnitude longer than the memory operations. Low-latency and byte-addressable NVM technologies are promising for the WAL committing [31], but have not been widely used. As a result, users must make their own trade-off between the performance and durability. For example, some applications make frequent *fsync* calls to commit recent writes in order to ensure high durability [32], [33], at the cost of throughput degradation, while others may flush accumulative logs to OS cache or disable the backup log for high throughput by sacrificing durability [34], [35]. LDS inherits the LevelDB policy that flushes each WAL to the OS cache. Users should explicitly set the synchronizing option of an insertion request if they want the insertion to be durable.

The commit policy described above raises a problem of how to identify the start point of the backup log in a recovery, since the reset object pointed by the version may not have been persisted. To resolve this problem, LDS also records the SN of the reset object in the version. If the recovery process finds that the SN of the reset object does not match the one recorded in the version, the backup log is simply ignored.

3.4 Internal Fragmentation

The padding between the body and tail sections of a chunk (Fig. 6c) can cause internal fragmentation that leads to some wasted storage space in LDS.

In the merge sort of a compaction, the merge process traverses multiple chunks and sorts their key-value pairs in the body of a new chunk. The tail of the new chunk is updated along with the increasing of the body. The chunk is packed after examining the size of the body and tail. If the merge process finds that adding one more key-value pair to the body would cause the package to overflow the slot size, it will not add this key-value pair, but turns to perform the packing. In this case internal fragmentation can occur in that the slot has some free space but insufficient for the next key-value pair of the merge process. This kind of fragmentation also exists in file systems because it is hard to generate a chunk file exactly aligned with the file-level block size (e.g., 4 KB). As long as the sizes of key-value pairs are less than the file-level block size, which is a common case in LSM-tree key-value stores [14], [36], [37], [38], the internal fragmentation in LDS will not be more detrimental than that in file systems.

However, at the end of the merge sort, in particular, the last chunk must be packed no matter how little data it contains. Such a chunk in LDS is called an *odd chunk* that has a variable size, and too many odd chunks existing in a level can cause significant internal fragmentation. To reduce the internal fragmentation caused by odd chunks, we make a small change to the processing of the odd chunk in each compaction from L_n to level L_{n+1}. Instead of placing it in L_{n+1}, the odd chunk is retained in L_n and has two possibilities in the future operations. One is that it is picked by the next compaction of L_{n-1} as the overlapped chunk. The other is that it is attached to its next adjacent chunk that will participate in the next compaction of L_n. In both cases the odd chunk is assimilated and absorbed. By doing so, each level has at most one odd chunk, regardless of the size of the store. As the odd chunk does not overlap with any chunks in L_n and L_{n+1}, placing it in L_n does not break the tree structure. Another alternative way is preferentially selecting the odd chunk as a victim for compacting, instead of selecting in a round-robin manner.

3.5 Implementation

We implemented a prototype of LDS based on LevelDB 1.18 to evaluate our design.

The first task is to manage the storage space, called the *I/O layer*. We use *ioctl* to obtain the properties of the storage partition and initiate the space by writing an initial version in the version log. The I/O operations on the physical space are implemented by `open/write/read/mmap` system calls. The number of synchronous requests from LevelDB are not changed by LDS. However, unlike the file systems that must update the FS indexes besides the user data, a synchronous request in LDS only involves a specific range of the user data. For a synchronous request, LDS calls the `sync_file_range` [39] with all the three flags set[4] to ensure that the data is persisted. We do not use fsync because it syncs once all buffered data of the raw device (e.g, syncing the chunk or version data also causes syncing the backup log, which is unexpected).

Although LDS can take control of the buffer management work and choose proper flush opportunities to achieve better performance without losing data consistency, such as enabling concurrent merging and flushing in the background process, we currently does not activate this feature. Instead, we implement the function to work in the same way as in LevelDB, so as to provide unbiased evaluation results. In general, LDS distinguishes three types of write requests, as chunk write, version logging and backup logging, and handles them separately according to the design described above. A total of 456 lines of code is written to implement the I/O layer.

The second task is to modify LevelDB to make it runnable on LDS's I/O layer, primarily a new implementation of the *env* interface, on which totally 32 lines of code is written.

3.6 Scalability

In this subsection we discuss the scalability of LDS, including features that are not implemented in the prototype LDS but can be easily realized based on the LDS design.

3.6.1 Self-Described Logging

Unlike the common logging systems that define a global metadata to identify the valid log [1], [25], LDS does not require the metadata. Instead, the validity of each log object is self-contained within the object so that updates in the log area are implemented in a strict appending manner. This strict appending mechanism in LDS enables flexible optimization for FLSM-trees on the flash-based storage with a specialized FTL layer [40]. In addition, the logging mechanism of LDS also enables to implement an efficient synchronous logging system on the emerging byte-addressable NVM, because LDS syncs each log to the persistent media without the need to write extra data.

3.6.2 Pre-Allocated File

LDS can be used as the storage engine to manage the low-level storage devices by providing the key-value interfaces (*put*/*get*/*delete*). Nonetheless, running LDS on a file space pre-allocated by a file system is readily feasible. For users, using a pre-allocated file space is exactly the same as using the raw device, but LDS internally can not use *sync_file_range* directly on the pre-allocated file space because of the usage limitation of *sync_file_range* [39] and the potential file-system interference on the allocation of the physical space.

4. SYNC_FILE_RANGE_WAIT_BEFORE | SYNC_FILE_RANGE_WRITE | SYNC_FILE_RANGE_WAIT_AFTER

Generally speaking, the address space (in bytes) of a pre-allocated file space starts from 0 and is statically mapped to LDS. The file system that allocates the file space maintains the mapping between the file space and the storage space in an inode. *sync_file_range* only ensures that data in the range of the file space is synced to the corresponding storage space [39], but does not ensure that the inode data (mapping information) is synced. If the file system updates in-place [22], the mapping information does not change when writing and syncing data in the file space, and there is no problem to retrieve the synced data after a crash. However, if the file system updates out-of-place [23], [24], writing and syncing in its file space always results in the data being persisted in a new storage location, and the mapping data in the inode should be updated to keep track of the new location. In such a case, synced data in the file space is lost after a crash if the inode data is not synced. Hence, fsync/fdatasync must be used to inform the file system of syncing the inode information.

The backup log in the pre-allocated file space needs to be specially processed for update-out-of-place file systems. As stated in Section 3.3.2, the backup logs usually are not flushed to disk at the same pace as the slot or version data. However, fsync on the pre-allocated file space applies to the whole file space, leading to a performance drop in the case where users choose to commit the backup log lazily. A practical solution is to designate a separate allocated file space for the backup log. Actually, separately storing the backup log in a different storage space has been a practical way to improve logging efficiency [41], which we will not elaborate any further in this paper.

3.6.3 Storage Size Adjustment

Being applicable to both raw device space and filesystem-allocated space, LDS enables flexible space adjustment according to user requirements of either expanding or shrinking the storage space by setting a special field in the version to describe the storage space it manages.

Expanding the storage space, i.e., joining a new device or requesting more space from the file system, is achieved by re-constructing the online-map to embody the expanded slots of the new space, and a new version is generated subsequently to persist the information of the joined space. To shrink the storage space, i.e., removing a device or giving back some space to the file system, LDS first copies the chunks in the shrunk slots to other free slots, then trims the versions and re-constructs the online-map to exclude the shrunk slots. In the trim process, the chunks that are originally stored in the shrunk slots are assigned new IDs according to their new hosting slots.

3.6.4 Multiple Instances

The above introduction of the LDS design assumes that only one LSM-tree instance runs on LDS. However, LDS is scalable to support multiple LSM-tree instances by applying a mechanism to distinguish their log objects. The instances can share the same log area and use a special field in the header of the log object to identify them, or employ separate independent log areas. Taking the latter as an example, LDS preserves a space to split log areas for a new LSM-tree instance. Each instance has its own version log and backup log, but shares the slot area with other instances. An instance that requests a chunk write gets a slot ID from LDS, and the ID is recorded in the version of the instance. Each logging request

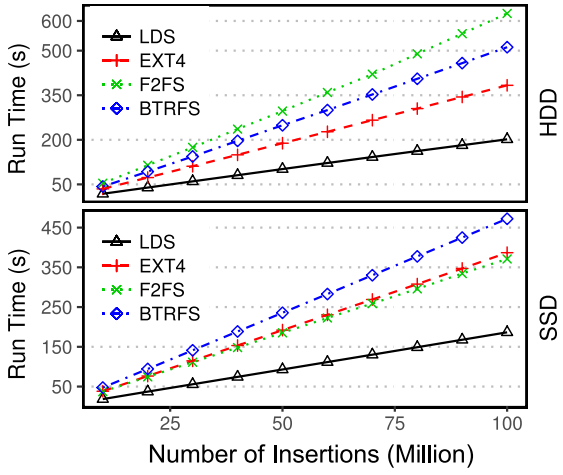


Fig. 9. Sequential insertion performance.

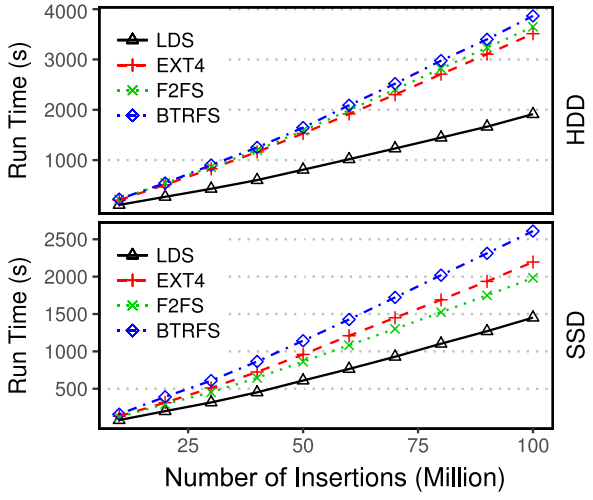


Fig. 10. Random insertion performance.

from an instance are appended to its dedicated log area. In restart, the recovery process will inspect the version logs of all instances to construct a correct online-map.

4 Evaluation

This section presents the experimental results that demonstrate the benefits of LDS.

4.1 Environment Setup

The experiments were conducted on a machine equipped with two Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00 GHz processors and 32 GB RAM. The operating system is 64-bit Linux 4.4. The HDD used, Seagate ST2000DM001, has a 1.8TB capacity with a 152 MB/s sequential write speed, and the SSD used, Intel SSD DC S3520 Series 2.5in, has a 480 GB capacity with a 360 MB/s sequential write speed. Note that HDDs have a slightly faster speed on the outer cylinders. Our experiments select the partitions starting from 800 GB of the HDDs for each system to minimize such hardware impacts on individual experiment. The write caches of the drives are disabled, to ensure the data being safely stored.

We compare the performance of LDS with that of LevelDB (1.18) running on three typical file systems, ext4 (update-in-place) [22], f2fs (log-based) [23], and btrfs (copy-on-write) [24]. All the file systems are mounted with the *noatime* option to eliminate potential overheads irrelevant to our evaluations. The chunk (SST) size is configured to be 4 MB in LevelDB. The version log and backup log in LDS are configured to be 64 MB and 16 MB respectively, and the slot size is 4 MB. This configuration does not trigger the trim process on the version log, which is the practice in LevelDB. The cost of the trim process is evaluated together with the recovery process (Section 4.4). Data compression in LevelDB is disabled in all experiments.

4.2 Write Performance

In this subsection we use the default benchmarks in LevelDB (*db_bench*) to evaluate the insertion performance of LDS and LSM-on-FS under the sequential and random workloads respectively. We also evaluate the insertion performance in the synchronous mode. The average key-value pair size is 116 Bytes (i.e., 16B key, values range from 1B to 200B with uniform distribution).

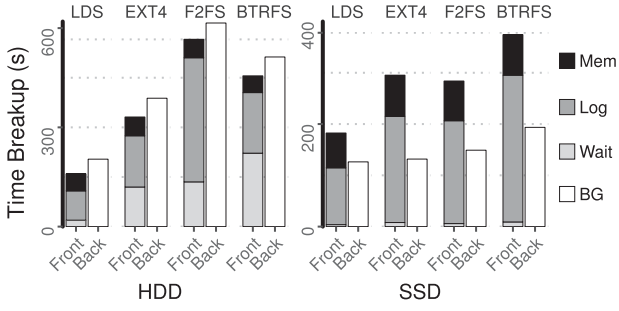


Fig. 11. A breakdown of run time for sequential insertions.

4.2.1 Sequential Workload

Fig. 9 shows the performance under the sequential workload in terms of run time as a function of the number of insertions. From the figure we can see that LDS performs the best on both HDDs and SSDs. To further analyze the results, we take a closer look at the time cost in Fig. 11 by examining the contributions to the run time by different types of operations/events.

LSM-tree has a foreground thread (Front) for the write-ahead log (Log) and MemTable inserting (Mem), and triggers the background thread (Back) to do compaction when the MemTable is converted to ImmTable. The foreground operation is slowed down (Wait) if the background thread does not finish the process in time. Under sequential workload, there is no merge sort in the compaction and chunk writes only happen in dumping the ImmTable to L_0. Compaction on an on-disk level is a trivial moving operation that only updates the LSM-index.

On HDDs, the background process in LSM-on-FS is slow because of the frequent FS index updates for both LSM chunk and LSM index, and there are waiting times in the foreground. On SSDs, as all the systems can quickly finish the background processing due to the low latency of flash, the foreground costs (mainly the logging cost) dominate the overall performance. Nevertheless, different file systems incurs respective overheads for the log requests because they have their own processing mechanisms for the write system calls (translated from *fflush* of LevelDB) that push the WALs to OS cache. For example, they will check there are enough free blocks for the write in order to guarantee the future flush will not fail [42].

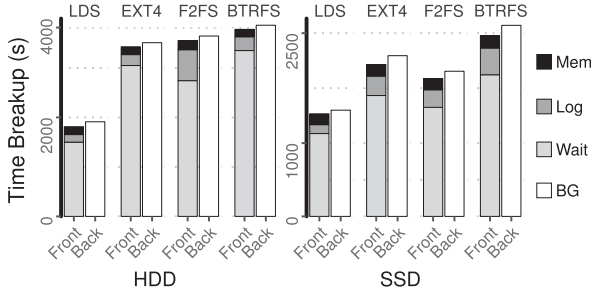


Fig. 12. A breakdown of run time for random insertions.

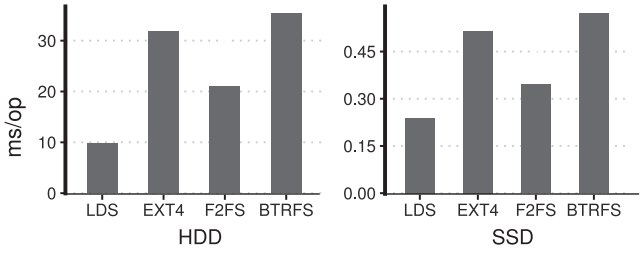


Fig. 13. Insertion latency in the synchronous mode.

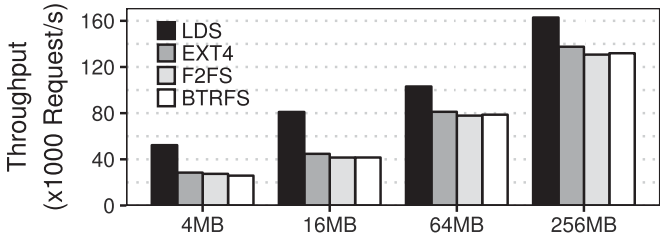


Fig. 14. Write throughput as a function of chunk size.

4.2.2 Random Workload

The performance results under random workloads are shown in Fig. 10, in terms of run time as a function of the number of insertions. Random insertions incur frequent compaction merge operations in the background and need a long time to perform the merge sort and chunk writes. As a consequence, the foreground process waits for the background process most of the time and the system with the best efficiency in chunk writes performs the best, as shown in the cost distributions of the run time for random insertions in Fig. 12.

The background process can block the foreground process because each level of the LSM-tree has a capacity limit as well as the two memory tables, as introduced in 2.1. When the memory tables are full and the room in L_0 is under pressure, the foreground process must slow down the insertion operations or wait until the background compaction has produced enough room in L_0.

4.2.3 Synchronous Insertion

The above two workloads are run with the backup log in the default setting, i.e., only flushing each write-ahead log to the OS buffer. However, users sometimes want the insertions they have issued are durable once the insertion request returns successfully. We use the synchronous mode provided by LevelDB to evaluate the performance in such case. In the synchronous mode, the insertion throughput is completely determined by the write efficiency for the backup log, whether the workload is sequential or random. The insertion efficiencies measured as insertion operation latencies are shown in Fig. 13.

For an insertion request in synchronous mode, LDS can achieve an efficiency equivalent to writing the same size of data on the raw storage device (refer to Fig. 5b). This is because LDS only incurs one I/O in the backup log area by using the strict appending mechanism. In LSM-on-FS, there are several FS index blocks that need to be updated together with the backup file update, in order to guarantee the request persisted both in the storage and in the file system. F2fs is optimized for small synchronous requests by implementing a roll-forward mechanism [23], which eliminates many of the FS index updates, therefore it performs better than ext4 and btrfs. However, f2fs still has to update one block for the FS index (i.e., *direct node* in f2fs), and results in longer latency than LDS.

4.2.4 Large Chunk

LSM-tree applications can configure write buffer and chunks of larger sizes to optimize write performance. For example, RocksDB stores the data in 64 MB-chunk files by default. We evaluate the performance of LDS under different chunk sizes with HDD as the underlying storage [5], and present the result in Fig. 14. The write buffer size in each experiment is set to the same as the chunk size. From the figure we can see that larger chunk and write buffer improves the write throughput for each system. Nevertheless, LDS consistently outperforms the LSM-on-FS implementations, and is able to improve the write throughput by 18 percent even when the chunk is set to an extremely large size, i.e., 256 MB.

4.2.5 Size-Tiered Compaction

While LevelDB implements the LSM-tree with the leveled compaction strategy, there is another LSM-tree family implementing the size-tiered compaction that maintains multiple stages each of which contains SSTs with predefined size [43], such as BigTable [4], Cassandra [5], HBase [3], etc. We implement the size-tiered compaction with the framework code of LevelDB (referred to as ST) to evaluate the efficiency of LDS for key-value stores based on the size-tiered compaction strategy on HDDs. The SST size in the first stage is configured to 4 MB, and the minimum compaction threshold is 4 (default configuration in Cassandra), meaning that when the number of SSTs in a stage reaches to 4, they are compacted to a larger SST that is moved to the next stage. For LDS, the slot size is fixed to 4 MB, and SST chunks larger than 4 MB are sliced to sub-chunks to fit the slot. The version data maintains the sub-chunks' IDs for each sliced chunk.

Fig. 15 shows the evaluation results from experiments that insert 100 million KV pairs to ST. We can see that LDS keeps performing at a significantly higher throughput level than all the LSM-on-FS implementations. Although ST writes large SST chunks in higher stages, all the data must go through the lower stages in which the FS overheads are more significant. In fact, even for the large chunks, LDS is proven to be more efficient than the LSM-on-FS implementation in Section 4.2.4.

4.3 Read Performance

We load 1 billion random key-value pairs with a fixed size (16B key and 100B value) to set up a 100 GB dataset to evaluate the read performance. The available OS cache is limited to

5. Note that using SSD as the storage media exhibits a similar performance trend.

Fig. 15. Write throughput of size-tired compaction.

1GB to emulate a 100× storage/memory configuration. A larger storage system can have even higher storage/memory configuration ratio [44]. The number of concurrent threads for read are set to 4 on HDDs, and 16 on SSDs. We measure the throughput and read amplification in cold cache and warm cache respectively, as shown in Fig. 16.

For a read request of a key, it is sent to a chunk after looking up the LSM index that resident in memory. Then, different systems translate the chunk ID to the on-disk location of the chunk data. Unlike LSM-on-FS where the FS index (file metadata) must be read to locate the chunk data, LDS can determine the chunk location directly from the LSM index. As different systems organize the storage space and design data indexing mechanism with their own ways, they induce different read amplifications, which not only influence the cache efficiency, but also impact the read performance. For example, ext4 clusters multiple inodes in one block, while f2fs exclusively allocates a block for each node object [23]. However, it is still interesting to see that the performance of btrfs is particularly low. A further analysis of the trace shows that btrfs has significantly higher read traffic than others as shown in Fig. 16c.

Read amplification is accounted for by the average I/O traffic of the processed requests. In cold cache, almost all requests are processed with I/Os to load the storage blocks that potentially contains the requested key-value data to the memory cache, so there are high read amplification. With the caches warmed up, a fraction of I/Os are avoided due to cache hits, resulting in lower read amplification. Especially, for a single read request btrfs incurs 8 I/Os in cold cache, of which half are larger than 512 KB, much more than other systems both in terms of the number and size of I/Os. As we use the same *mmap* system call to read chunk data from the underlying storage, the difference in read amplification can be only caused by the internal data layout of the different systems. The high read amplification of btrfs is also observed by Mohan etc. [45].

4.4 Recovery

In this subsection we evaluate the cost of recovering the in-memory version from the on-disk version log. Recovering the

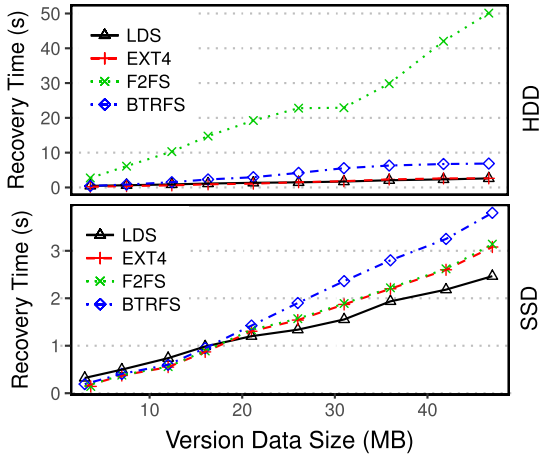


Fig. 17. Recovery time as a function of the size of the accumulated version data.

memory table from the backup log is a similar procedure but without the need to scan the entire log area to locate valid objects. Our evaluation on LDS always assumes the worst case, that is, we always scan the entire version log area (64 MB) even when we have determined all the live objects, and only after the scan finishes we begin to perform the *trim*. The total time cost on the LevelDB recovery process is used to measure the recovery performance. Experiments on LSM-on-FS were executed after the file system has been prepared, and file system consistency check during the mounting time [46] is not taken in account. We use the random workload in Section 4.2 to insert 100∼1000 million key-value pairs to generate different sizes of version data (from 3 MB to 47 MB).

Fig. 17 shows the recovery time for different sizes of accumulated version data. The recovery cost mainly comes from I/O cost of loading the version data and CPU cost of performing the trim. While performing trim is a similar process for all systems that costs time proportional to the version data size, which takes 0.1 second for a 3 MB version data and 2 seconds for a 47 MB version data, the variance in recovery time is attributed to the I/O cost. As we know, log-structured file systems always allocate blocks for all the files at the logging head on the block address space. While the version file (a logical log to the application) is periodically appended mixed with SST file writes, it becomes fragmented by the file-system log, a problem similar to the known log-on-log phenomenon [28]. As a result, the recovery time required for f2fs is significantly longer than other systems on HDD. Fragmenting the version file can also happen in the general CoW file system [47] (e.g., btrfs). Moreover, btrfs has a high read amplification as shown in Section 4.3, therefore, reading the version file of btrfs is a costly operation on both HDDs and SSDs.

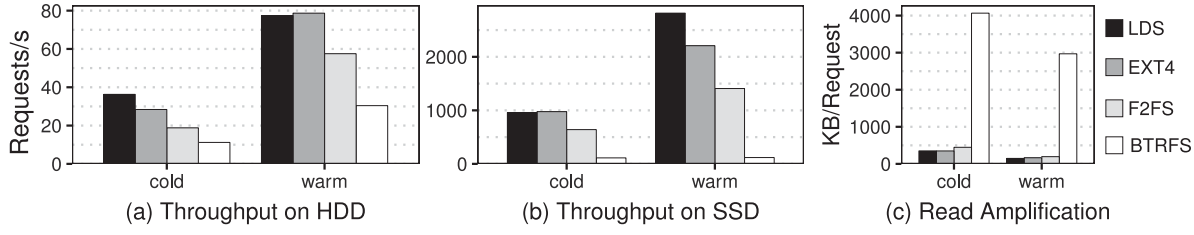


Fig. 16. Read throughput and amplification on cold cache and warm cache. The number of threads for read is 4 on HDD, and 16 on SSD. The first 10× Number-of-Threads requests are accounted for cold cache results, and results on warm cache are achieved after the free cache has been filled up and the throughput reaches the steady state. Read amplification is measured by averaged IO traffic (KB) of the requests.

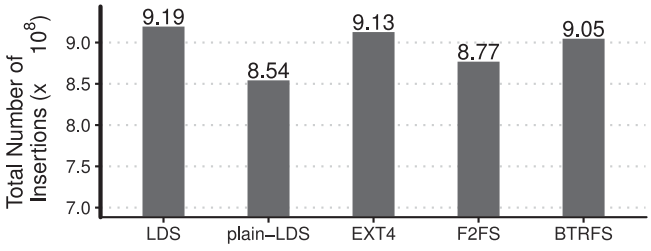


Fig. 18. Total key-value pairs inserted when the system reports 'space full' on a 100 GB storage device.

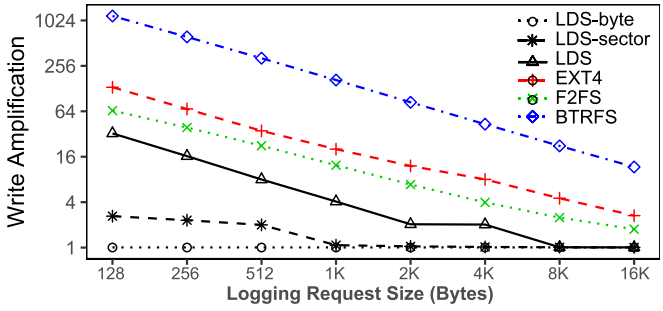


Fig. 19. Write amplification of synchronous logging.

LDS spends slightly more time than others when the version data is small. This is because in the worst-case scenario LDS must load the entire log area and perform a thorough scan to find the live objects regardless of the version data size, which takes a constant time of 0.35 second on HDD and 0.18 second on SSD. With this small trade-off, LDS can perform efficient log updates in the runtime.

4.5 Space Utilization

We compare the space utilizations of different systems in this subsection to study the impact of the internal fragmentation in LDS. We define the utilization as the number of fix sized key-value pairs a system can accommodate on a storage device with a given capacity. The experiment for each system is done by using the random workload to fill up a 100 GB storage device with key-value pairs of 116 Bytes (16B key and 100B value), until "space full" is reported by the system. In order to examine the effectiveness of the LDS optimization to reduce the odd-chunk induced fragmentation, we also run a test on LDS without this optimization (labeled as plain-LDS).

A comparison of the space utilization is shown in Fig. 18. From the figure we can see that, without taking any measures to reduce fragmentation, plain-LDS has the lowest utilization, accommodating an amount of key-value pairs that is about 97 percent of f2fs, 93 percent of ext4 , and 94 percent of btrfs. Our investigation shows that the space wastage mainly comes from odd chunks. With the aforementioned fragmentation reduction optimization, LDS achieves the best space utilization among all the systems. The inefficiency in file systems mainly comes from the FS-index induced space overhead, which is more obvious in f2fs because it needs quite a few blocks to store the node address table.

4.6 Write Amplification of Logging

In this subsection we evaluate the write amplification of synchronous logging with varying logging size, to show the friendliness of LDS to NVM storage, since lower write amplification helps extending the NVM's lifetime.

We first perform experiments in the normal way, in which each log request is synced to the underlying media through the OS page cache in the unit of 4 KB, to have an general comparison of the logging write amplification between LDS and the LSM-on-FS implementations. After that, we specifically evaluate the logging write amplification of LDS on two kinds of storage media with write unit of 8B (labeled as "LDS-byte") and 512B (labeled as "LDS-sector") respectively. The 8B write unit represents the access unit of the byte-addressable NVM [48], [49], while 512B is the atomic write unit of the legacy block storage. The evaluation results are demonstrated in Fig. 19.

In the normal way, we can see that when the logging size is larger than 8 KB LDS incurs no write amplification, while the LSM-on-FS implementations still induce high write amplifications (from 1.7 to 22) because of the FS metadata updating. When the logging size is small, the LSM-on-FS implementations induce even higher write amplification since they still need to update the FS metadata block/blocks that are much larger than the logging data. For example, when the logging size is 128B, f2fs must update one 4 KB metadata block besides a 4 KB file data block, resulting a write amplification of 64 theoretically, which is consistent with the experimental result. For the storage media supporting fine-grained write unit (e.g., byte addressable NVM), LDS is flexible to reduce the write amplification because there is no other data needing to write except the logging data itself. In general, LDS induces no write amplification of logging when the write unit of the media is smaller than the logging size. This feature makes LDS promising to achieve both performance and durability advantages on the byte-addressable NVMs [48], [49].

4.7 Overheads on Pre-allocated File Space

Although LDS is designed as a raw device storage manager for LSM-trees, it readily supports the feature of using a pre-allocated file space with some limitations as stated in Section 3.6.2. In this subsection we conduct experiments to evaluate the limitations when LDS uses a pre-allocated file space from the three representative file systems, i.e., the impact of the file-system interference. While ext4 is known as an update-in-place file system[6], the other two are out-of-place-update file systems. We hence keep using *sync_file_range* on the ext4-allocated file space, and disable the journaling of ext4 since mapping data in the inode is not updated when writing the LSM-tree data. On the file space pre-allocated by f2fs or btrfs, fsync is used to guarantee that the updated mapping data is synced together with the LSM-tree data. Evaluation results show that there are two kinds of overheads induced from the file system interference on the pre-allocated file space.

The first kind is the system call overhead of flushing the WAL from user space to OS cache in the default commit policy, in which each of the LevelDB's logging requests is translated to a *write* system call and is eventually processed by the corresponding file system that allocates the file space. This kind of overhead is almost the same as the one when LevelDB runs directly on file systems, as the *Log* cost shown in Fig. 11. Employing *mmap* to implement the flushing can significantly minimize the WAL overheads, since flushing

6. We assume that ext4 updates in-place strictly.

the log to the OS cache will be a *memcpy* operation that does not lead to system calls after the page table entries of the log area have been established.

The other kind is the I/O overhead induced by the file system interference, which mainly exists on the out-of-place-update file systems because they always allocate new storage blocks for any LDS writes and need to sync the file metadata when the LDS data is synced. For example, on HDDs, the overhead of using btrfs-allocated file space is about 2.5× higher than using raw space, which is equivalent to the conventional way of running LevelDB on btrfs because the wandering-update can not be avoided. This value is 1.4× for f2fs-allocated file space, or half the overhead of running LevelDB on f2fs since the NAT has been established for the pre-allocated file space and most of the time one indirect node needs to be synced [23] when the LDS data is synced. For the ext4-allocated file space, while it does not need to update the file metadata, the overhead is equivalent to using raw space. For example, in the pre-allocating process, ext4 determines all the physical blocks that will belong to the pre-allocated file, and creates an inode that maps the file space (file offsets) to the physical block space (i.e., LBAs). Since the file system updates in-place, subsequent writes from LDS to the file space directly go to the corresponding physical blocks, and only syncing the data written in the file space is enough to guarantee the data consistency because the mapping information in the inode does not change. Therefore, LDS can work on the ext4-allocated file space the same way as on the raw space.

5 Discussions

5.1 Log Durability in LSM-trees

In LSM-trees, the backup log is used to recover the KV pairs buffered in memory, so as to provide the durability for users. Here we associate the durability measure to three separate levels, called device-level, OS-level, and application-level.

Device-level durability requires that a write request be written to the device before returning success to the user, which provides the highest measure of durability. Under the device-level durability, users can always retrieve the written data if the write request is returned, even though the OS is crashed after that. However, maintaining this measure of durability bears great overhead, because the operation of writing the log lies in the critical path and I/O latency on the block device is much longer than memory access latency. New storage technologies such as byte-addressable NVM with low latency [48], [49] are promising to reduce the high I/O cost in support of device-level durability [31], [50].

OS-level durability guarantees that the written data can be retrieved so long as the OS does not fail. To provide OS-level durability, each write request needs to be pushed to the OS cache, which may cause system call overheads, as shown in Section 4. LevelDB naively support OS-level durability. That is, if a write request to LevelDB returns successfully, the written data can be retrieved even if LevelDB exits abnormally, as long as the OS is not corrupted. Data under OS-level durability is periodically committed to the device according to the OS configuration [51], [52], [53].

Application-level durability buffers the written data in the application managed cache, and the data will be lost if the application exits abnormally, which provides the lowest measure of durability, but obtains higher performance. For example, Wisckey [35] caches the write requests in the application buffer and flushes them to the OS cache at the 4 KB granularity, therefore making its system call overheads much less than that in LevelDB. Specifically, the system call overheads are more pronounced under sequential workloads on low-latency devices.

5.2 Concurrent Operations

Concurrent operations in LSM-trees include CPU-memory concurrency and I/O concurrency. RocksDB [2] and Cassandra [43] provide support for concurrent compaction operations, which translate to concurrent I/Os on the storage device (i.e., I/O concurrency). This kind of concurrency works better on SSD devices that are friendly to parallel accesses. Besides, as sequential workloads do not trigger compaction operations, only random workloads would benefit from this kind of concurrency. HyperLevelDB [54], [55] opertimizes LevelDB by allowing multiple threads to concurrently insert key-value pairs into the MemTable and append logs on the backup log buffer (i.e., CPU-memory concurrency). From the experimental results in this paper (Figs. 11 and 12) we can see that this kind of concurrency would only take effect under sequential workloads, especially with low latency devices, in which the *MemTable* inserting and *Log* buffer writing operations dominate the system throughput.

While LDS functions by managing the storage space in an LSM-tree friendly manner without changing the LSM-tree's operation flow, the concurrency optimizations on LSM-trees can be straightforwardly applicable to LDS. For example, two compaction operations on different sets of SSTs that are able to be accessed in parallel in LSM-trees translate to read/write operations on different sets of slots in LDS. If the slots are on the SSD device, they are accessed in parallel potentially.

6 Related Work

6.1 Write-Optimized Data Structures

Traditional database systems such as SQL Server [56] employ B+trees as the backend structures, which are excellent for reads, but have poor performance for writes. Fractal-trees [57], [58], [59] are write-optimized data structures as LSM-trees, which maintain one global B+tree with a buffer in each node, and updates descend the B+tree to the leaf nodes ***in batch*** through the buffers of the intermediate nodes, a similar idea as the LSM-tree has proposed [14]. Write-optimized data structures have been widely used as storage engines in modern data stores [1], [3], [4], [13], [60], [61]. This paper focuses on optimizing the storage stack of LSM-tree based key-value stores.

6.2 Optimizations on LSM-trees

With the popularity of LSM-trees in large data stores, a lot of techniques have been researched to optimize the write efficiency of LSM-trees. Most of the work contribute to reducing the write amplification. VT-tree [62] optimizes the write amplification in sequential write intensive workloads by only merging the overlapped portions of chunks. Wisckey [35] reduces the value induced amplification by moving the values out of the LSM-tree to a separate log, a similar way implemented in Bitcask [63] that uses an in-memory hash table to index the value log. LSM-trie [38] and PebblesDB [54] separate the keys of each level to different partitions, and reduce the write amplification by allowing

overlapped chunks within a partition. TRIAD [64] optimizes the write amplification by exploiting the skewed workloads and delaying the compaction process.

Our work is different from and orthogonal to the above existing work in that we optimize the LSM-tree by providing an LSM-tree friendly on-disk data layout.

6.3 Bypassing Storage Stack Layers

In the initial days of the database field, data was directly stored on the block storage that has small capacity, and the application was responsible for the block/segment allocation and data consistency [65]. The file system was designed to provide directory hierarchy abstraction and data store of arbitrary sized objects, by organizing the storage space with uniform file-level blocks and introducing an indirect map between the data objects and the underlying storage space [29], [66]. Stonebraker [29] examined the overheads of database systems caused by different OS components including file systems. Engler and Kaashoek [67] proposed to completely eliminate the OS abstractions and allow applications to select efficient implementations from the hardware. Nevertheless, with the rapid growth of storage capacity in the following tens of years, it was profitable to share a storage with multiple applications and offload the complicated storage management work to a file system. However, with the advent of big data, e.g., large-scale key-value stores [1], [4], [38], the data size is easy to grow out of the storage capacity, and the application that is in charge of the large and uniform data objects benefits little from the file system layer. In contrast, file systems can have negative impacts on a high-performance data store because of the extra indirections and consistency enforcement.

Recent work on key-value stores have persisted their data bypassing the file system because of observed performance degradation [68]. Papagiannis etc. propose the system Iris [69] to reduce the software overheads pronounced in the I/O path with low-latency storage devices. NVMKV [70] is a key-value store that makes a radical step by directly hashing each individual key-value pair into the sparse FTL space of SSDs. Nevertheless, they did not explicitly quantifies the overheads caused by the data deployments of different file systems, and how the LSM-tree applications are affected remains unclear.

Current implementation of LDS that bypasses the file systems trades off the convenience of file-system APIs. Nonetheless, integrating the design of LDS to a file system is feasible. For example, with a strict update-in-place file system, the discarded chunk files can be retained to hold new chunk data without updating the file index, as discussed in Section 4.7.

6.4 New Storage Technology

New technology such as multi-stream NVM has been presented to be aware of the application-layer data streams [71], which can be an opportunity for LDS to store the LSM-tree data in an NVM-friendly way. FlashBlade [44] builds up a flash-based storage array that moves the flash translation functions at the array-level software, and requires the software to carefully regulate the user data to sequential stream. LDS provides an easy way to manage the flash translation functions at the application layer because it eliminates the extra I/Os in the storage stack and retains the sequential I/O pattern of LSM-trees.

Some other work exploit the properties of new storage medias from application layer. LOCS [72] optimizes the performance of LSM-tree applications via exposing the channels of SSD to the upper application to unearth the bandwidth utilization of SSD. Lee etc. [40] proposed an application-managed flash system that resolves the discrepancy between application-layer logging and flash-layer logging to improve both the application performance and flash management overhead. Colgrove etc. [73] introduced a storage system bypassing the the kernel block device with a custom kernel module and translating application-level random writes into compressed sequential writes, to benefit the underlying flash array [44].

While LDS provides high performance for LSM-tree based key-value stores on both HDD and SSD devices, there are potential benefits that can be gained from LDS if the internal characteristics of SSDs are taken into consideration. For instance, the expensive garbage collection operations in flash storages can be eliminated as LDS always discards data in the unit of slot that can be erased without data migration. Besides, we plan to augment LDS to be flash-aware so that it can perform the wear-leveling work, which is simpler and more convenient for LDS.

7 Conclusion

In this article we present LDS, a storage system that employs the LSM-tree structure (a widely used structure for large-scale key-value stores) to manage the underlying storage space, so as to retain the full properties of the LSM-trees. We propose LDS based on a detail research on the overheads induced by the intermediate storage layer, such as file systems. With LDS, the I/O patterns from LSM-trees are fully retained on the underlying storage device. An LDS prototype based on LevelDB shows that LDS delivers significant performance improvement and I/O reduction compared to LSM-trees running on state-of-the-art file systems.

Acknowledgments

The authors thank Vijay Chidambaram, Russell Sears, and Mark Callaghan for their help which improved the SoCC version of this article. They thank all of the anonymous reviewers for their valuable and constructive comments. This work is supported in part by the Fundamental Research Funds for the Central Universities No. 2018KFYXKJC037, and the US NSF under Grant No.CCF-1704504 and No.CCF-1629625.

References

[1] S. Ghemawat and J. Dean, "Leveldb," 2011. [Online]. Available: http://leveldb.org
[2] Facebook, "Rocksdb." [Online]. Available: http://rocksdb.org/
[3] M. N. Vora, "Hadoop-hbase for large-scale data," in *Proc. Int. Conf. Comput. Sci. Netw. Technol.*, 2011, pp. 601–605.
[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008, Art. no. 4.
[5] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.
[7] InfluxData, Inc., "The modern engine for metrics and events." 2017. [Online]. Available: https://www.influxdata.com/

[8] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: Random write considered harmful in solid state drives," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–16.
[9] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Trans. Storage*, vol. 8, no. 4, 2012, Art. no. 14.
[10] K. Smith, "Garbage collection," presented at the *SandForce, Flash Memory Summit*, Santa Clara, CA, USA, 2011.
[11] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. SYSTOR: Israeli Experimental Syst. Conf.*, 2009, Art. no. 10.
[12] S. G. Edward and N. Sabharwal, "Mongodb explained," in *Practical MongoDB*. New York, NY, USA: Springer, 2015, pp. 159–190.
[13] SQLite, "SQLite4: LSM Design Overview," 2016. [Online]. Available: https://www.sqlite.org/src4/doc/trunk/www/lsm.wiki
[14] P. Oneil, E. Cheng, D. Gawlick, and E. Oneil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.
[15] C. Li, P. Shilane, F. Douglis, D. Sawyer, and H. Shim, "Assert (! defined (sequential i/o))," in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst.*, 2014.
[16] G. R. Ganger and Y. N. Patt, "Metadata update performance in file systems," in *Proc. 1st USENIX Conf. Operating Syst. Des. Implementation*, 1994, pp. 49–60.
[17] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," *ACM SIGOPS Operating Syst. Rev.*, vol. 21, no. 5, pp. 155–162, 1987.
[18] T. Kowalski, "Fsck - the unix file system check program," in *UNIX Vol. II*. Philadelphia, PA, USA: Saunders, 1990, pp. 581–592.
[19] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," presented at the *10th USENIX Symp. File Storage Technol.*, San Jose, CA, USA, Feb. 2012.
[20] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 228–243.
[21] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All file systems are not created equal: On the complexity of crafting crash-consistent applications," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, Oct. 2014, pp. 433–448.
[22] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2007, pp. 21–33.
[23] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.
[24] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, 2013, Art. no. 9.
[25] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, "Crash consistency: Fsck and journaling," 2015. [Online]. Available: http://pages.cs.wisc.edu/ remzi/OSTEP/file-journaling.pdf.
[26] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.
[27] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The linux implementation of a log-structured file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 3, pp. 102–107, 2006.
[28] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *Proc. 2nd Workshop Interactions NVM/Flash Operating Syst. Workloads*, 2014.
[29] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412–418, 1981.
[30] SQLite, "Atomic commit in sqlite," https://www.sqlite.org/atomiccommit.html, 2007.
[31] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2016, pp. 385–398.
[32] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/o stack optimization for smartphones," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 309–320.
[33] M. Owens and G. Allen, *SQLite*. New York, NY, USA: Springer, 2010.
[34] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 217–228.
[35] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 133–148.
[36] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, 2012, pp. 53–64.
[37] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al., "Scaling memcache at facebook," in *Proc. 10th USENIX Symp. Networked Syst. Des. Implementation*, 2013, pp. 385–398.
[38] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 71–82.
[39] Michael Kerrisk, "Linux programmer's manual," 2017. [Online]. Available: http://man7.org/linux/man-pages/man2/sync_file_range.2.html
[40] S. Lee, M. Liu, S. W. Jun, S. Xu, J. Kim, and A. Arvind, "Application-managed flash," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 339–353.
[41] DataStax, "Dse 5.1 administrator guide: Changing logging locations," 2017. [Online]. Available: https://docs.datastax.com/en/dse/5.1/dse-admin/datastax_enterprise/config/chgLogLocations.html
[42] E. Wiki, "Life of an ext4 write request," 2011. [Online]. Available: https://ext4.wiki.kernel.org/index.php/Life_of_an_ext4_write_request
[43] Apache, "Types of compaction," 2016, http://cassandra.apache.org/doc/latest/operating/compaction.html
[44] P. Storage, "From big data to big intelligence," 2017. [Online]. Available: https://www.purestorage.com/products/flashblade.html
[45] J. Mohan, R. Kadekodi, and V. Chidambaram, "Analyzing IO amplification in Linux file systems," ArXiv e-prints, Jul. 2017.
[46] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. K. Mckusick, "Ffsck: The fast file-system checker," *ACM Trans. Storage*, vol. 10, no. 1, 2014, Art. no. 2.
[47] Z. N. J. Peterson, "Data placement for copy-on-write using virtual contiguity," Ph.D. dissertation, Computer Engineering, Univ. California Santa Cruz, Santa Cruz, CA, USA, 2002.
[48] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, et al., "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram," in *Proc. IEEE Int. Electron Dev. Meeting*, 2005, pp. 459–462.
[49] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.
[50] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.
[51] R. Love, *Linux Kernel Development*, 2nd Edition. Novell Press, 800 East 96th Street, Indianapolis,. Indiana, 46240 USA, 2005.
[52] "The pdflush daemon." [Online]. Available: http://www.makelinux.net/books/lkd2/ch15lev1sec4
[53] "Linux 2 6 32 - linux kernel newbies," 2009. [Online]. Available: https://kernelnewbies.org/Linux_2_6_32
[54] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Simultaneously increasing write throughput and decreasing write amplification in key-value stores," in *Proc. 26th ACM Symp. Operating Syst. Principles*, Oct. 2017, pp. 497–514.
[55] "Hyperleveldb performance benchmarks," 2017. [Online]. Available: http://hyperdex.org/performance/leveldb/
[56] Z. Tang and J. Maclennan, *Data Mining with SQL Server 2005*. Hoboken, NJ, USA: Wiley, 2005.
[57] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook, "On external memory graph traversal," in *Proc. 11th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2000, pp. 859–860.
[58] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming b-trees," in *Proc. 17th Annu. ACM Symp. Parallel Algorithms Architectures*, 2007, pp. 81–92.
[59] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 546–554.
[60] Tokutek, Inc., "Tokudb: MySQL performance, mariadb performance," 2013, http://www.tokutek.com/products/tokudb-for-mysql/

[61] K. Ren and G. Gibson, "Tablefs: Enhancing metadata efficiency in the local file system," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 145–156.
[62] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 17–30.
[63] D. Sheehy and D. Smith, "Bitcask: A log-structured hash table for fast key/value data," White paper, Apr., 2010.
[64] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, no. EPFL-CONF-228863, 2017, pp. 363–375.
[65] R. A. Lorie, "Physical integrity in a large segmented database," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 91–104, 1977.
[66] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 181–197, 1984.
[67] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proc. 5th Workshop Hot Topics Operating Syst.*, 1995, pp. 78–83.
[68] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, "Tucana: Design and implementation of a fast and efficient scale-up key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 537–550.
[69] A. Papagiannis, G. Saloustros, M. Marazakis, and A. Bilas, "Iris: An optimized i/o stack for low-latency storage devices," *ACM SIGOPS Operating Syst. Rev.*, vol. 50, no. 1, pp. 3–11, 2017.
[70] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "Nvmkv: A scalable, lightweight, ftl-aware key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 207–219.
[71] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst.*, 2014.
[72] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 16.
[73] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang, "Purity: Building fast, highly-available enterprise flash storage from commodity components," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1683–1694.

Fei Mei received the BS degree in software engineering from Wuhan University, in 2010. He is now working toward the PhD degree in the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. He works on optimizing the Key-Value stores on flash based storages, and has interests in the file system and NVM storage. He is a student member of the China Computer Federation (CCF).

Qiang Cao received the BS degree in applied physics from Nanjing University, in 1997, the MS degree in computer technology, and the PhD degree in computer architecture from the Huazhong University of Science and Technology, in 2000 and 2003, respectively. He is currently a full professor wtih the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His research interests include computer architecture, large scale storage systems, and performance evaluation. He is a senior member of the IEEE and China Computer Federation (CCF) and a member of the ACM.

Hong Jiang received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, in 1991. He is currently chair and Wendell H. Nedderman Endowed professor with the Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director with the National Science Foundation (Jan. 2013 - Aug. 2015) and he was with the University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. He has graduated 16 PhD students who upon their graduations either landed academic tenure-track positions in PhD-granting US institutions or were employed by major US IT corporations. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, and performance evaluation. He recently served as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 300 publications in major journals and international conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *Proceedings of IEEE*, the *ACM Transactions on Architecture and Code Optimization*, the *ACM Transactions on Storage*, the *Journal of Parallel and Distributed Computing*, ISCA, MICRO, USENIX ATC, FAST, EUROSYS, LISA, SIGMETRICS, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc., and his research has been supported by NSF, DOD, the State of Texas and the State of Nebraska, and industry. He is a fellow of the IEEE, and member of the ACM.

Lei Tian received the PhD degree in computer engineering from the Huazhong University of Science and Technology, in 2010. He is a staff engineer with Tintri. Prior to joining Tintri, he was a research assistant professor with the Department of Computer Science and Engineering, University of Nebraska-Lincoln. His research interests mainly include storage systems, distributed systems, cloud computing, and big data. He has more than 50 publications in major journals and conferences including FAST, SOCC, HOTSTORAGE, ICS, SC, HPDC, ICDCS, MSST, MASCOTS, ICPP, IPDPS, CLUSTER, the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *ACM Transactions on Storage*, etc.