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

Fei Mei
Qiang Cao*
KLISS, WNLO, Huazhong University of Sci. and Tech.
Wuhan, China
{meifei,caoqiang}@hust.edu.cn

Hong Jiang
University of Texas at Arlington
Arlington, USA
hong.jiang@uta.edu

Lei Tian
Tintri
Mountain View, USA
leitian.hust@gmail.com

ABSTRACT

Key-value stores are increasingly adopting LSM-trees as their enabling data structure in the backend storage, and persisting their clustered data through a file system. 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 based Direct Storage system that manages the storage space and provides simplified consistency control by exploiting the copy-on-write nature of the LSM-tree structure, so as to fully reap the benefits of LSM-trees.

Running LSM-trees on LDS as a baseline for comparison, we evaluate LSM-trees on three representative file systems (EXT4, F2FS, BTRFS) with HDDs and SSDs respectively, to study the performance potentials of LSM-trees. Evaluation results show that the write throughputs of LSM-trees 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.

CCS CONCEPTS

• **Information systems** → *Slotted pages*; *Hierarchical storage management*;

*Corresponding author: caoqiang@hust.edu.cn. Key Laboratory of Information Storage System (KLISS), Ministry of Education. Wuhan National Laboratory for Optoelectronics (WNLO).

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127486

KEYWORDS

LSM-tree, Key-Value Store, File System Performance, Application Managed Storage

ACM Reference Format:
Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian. 2017. LSM-tree Managed Storage for Large-Scale Key-Value Store. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 15 pages.
https://doi.org/10.1145/3127479.3127486

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 [19], RocksDB [16], HBase [66], BigTable [7], Cassandra [32], PNUTS [11], InfluxDB [23], etc., because LSM-trees are capable of buffering random writes in memory and then performing sequential writes to persistent storage, which is the best expected access pattern for both hard-disk drives and solid-state devices [22, 28, 43, 60]. To benefit from these potential advantages of LSM-trees, other popular database storage backend systems that traditionally organize data in B-trees have also begun to use LSM-trees in their new releases, such as MongoDB [13] and SQLite4 [61].

Ideally, an LSM-tree expects to store its data to storage space contiguously, but a file system that manages the storage can prevent this from happening by virtue of file-system indexing, as demonstrated by an example depicted in Figure 1. Usually a file system stores the file index in the forms of file metadata (i.e., inodes) and resource allocation map (i.e., block bitmaps), to respectively locate the file/directory data blocks and find free blocks for storing the data to write. Maintaining these index blocks, on the one hand, incurs more non-sequential I/Os, which harms performance for both HDDs and SSDs [22, 28, 35, 43, 46, 60]. Moreover, all these index blocks must be updated synchronously with the data blocks for strict data consistency, which requires significant extra work to carry out [8, 9, 18, 21, 31, 51].

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 without additional indexes 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

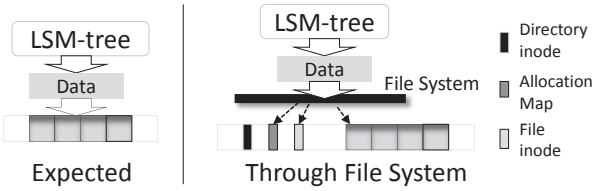

Figure 1: An example of how a sequential pattern expected from LSM-tree is broken up by a file system on the block storage. *Note that the inodes and allocation map are placed and updated by different file systems.*

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 characteristics of data layout and I/O processing, namely, ext4 (update-in-place) [40], f2fs (log-based without wander-tree updating) [33], and btrfs (B-tree structured and copy-on-write) [55]. 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 necessary background and an in-depth analysis of LSM-trees 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 5. Finally, we conclude our work in Section 6.

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 [46], 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* [16, 19], 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 [19], 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 out-of-place chunk writes, LSM-tree generates write-ahead logs to an on-disk backup in order to make the component C_0 recoverable upon a crash.

Let us take LevelDB, a widely used LSM-tree key-value store based on the partial merge introduced above, as a concrete example. Figure 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 that makes a change on a level's structure must update the MANIFEST, which is also implemented in a logging manner, called a version edit or Δversion.

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., 4KB)

[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.

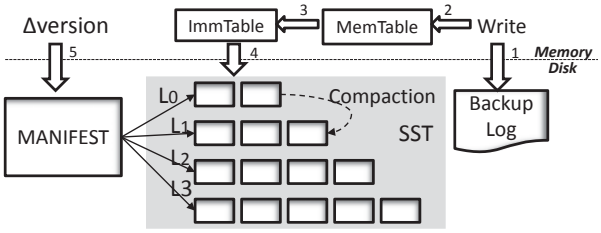

Figure 2: LevelDB implementation of LSM-tree. ***The in-memory component, C_0, is composed of two sorted skiplists (the MemTable and ImmTable), and each on-disk component (C_1, C_2, etc.) is a level (L_0, L_1, etc.). There are three types of disk writes throughout the runtime, as indicated in the figure: 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).***

as constituents of the file data. Consequently, updates to LSM chunks or LSM index come into force by non-sequential, small writes to update 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 in-place-update 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). 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 [53], resulting in space leakages or file corruptions. To maintain the data consistency, in-place-update file systems usually employ a journal to obtain atomic updates to the inodes and bitmap, which introduces additional overheads. 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 user data blocks and index blocks are stored in separate places, the amplified I/Os of in-place update are usually random accesses to the storage device.

Second, for a log-based file system [56], an instance of out-of-place update, although updates to the FS index can be contiguous to the user data, it must update the entire metadata path from the root inode to the user data as well as the allocation map, a phenomenon known as the *wandering-tree update* [33, 55, 56]. In addition, measures must be taken to reclaim the dead data blocks for the out-of-place-update file systems, a process known as garbage collection (GC). How to implement an efficient GC 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. We have experimented on running the LSM-tree on NILFS2 [30], 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 [33] is a practical log-structured file system that resolves the wandering-tree update problem by introducing a Node Address Table (NAT) for the metadata blocks and storing the resource allocation map in an in-place-update manner, at the cost of losing the benefits of logging feature and resulting in more random I/Os. One problem of the log file system is that the clustered user data (such as LSM-tree log) can be fragmented across the storage space [70].

The third type of file systems are the copy-on-write (CoW) file systems, another instance of out-of-place update, such as btrfs [55], which also update the index in a wandering fashion, except that they do not guarantee that the updates are physically contiguous.

In both types of the out-of-place-update 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. 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 such 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 4MB 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. Figure 3 illustrates the write patterns of LSM-tree through the three representative file systems and LDS respectively. 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, Figure 4a 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 Figure 4b the I/O latencies of persisting different sized requests on raw HDD and SSD devices. In our experiments for Figure 4a, 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 Figure 4b, 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

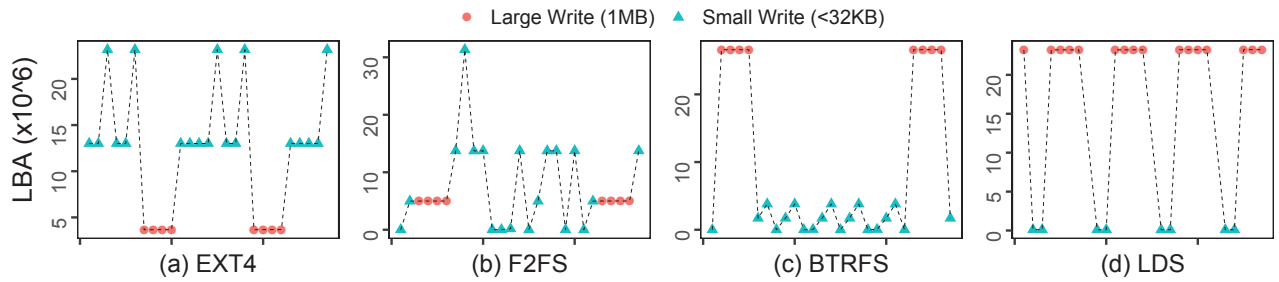

Figure 3: ***Figures a, b and c show the write patterns of LSM-trees through three representative file systems, in which the large writes are updates to the LSM chunks while the small writes are updates to FS index and LSM index. Figure d shows that the FS index writes are completely eliminated by LDS, leaving on the LSM index writes. The experiments for this figure are run with the backup log disabled under sequential workloads.***

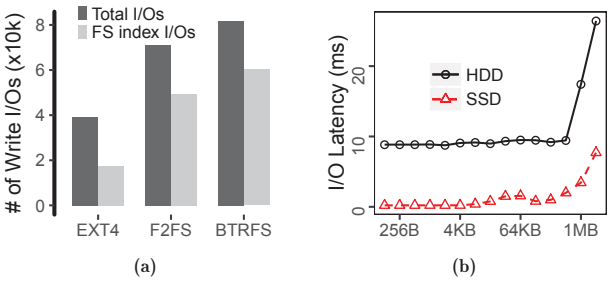

Figure 4: (a) ***I/O distributions when storing LSM-trees through three representative types of file systems.*** **(b)** ***I/O latencies of persisting different sized requests on HDD and SSD.***

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.*

While large SSTs incur lower FS index overheads, they have limitations. First, a larger SST always results in larger index data considering the fact that key-value pairs are meant to be small [2, 45, 46, 69], which leads to a longer search path. On the contrary, a smaller SST can be well organized to fit the index data into one storage block to enable efficient search. Second, compaction on large SSTs requires reserving sufficient free storage space to admit the new SSTs [20], and takes a long time to finish the merging process, which can block more urgent operations, lead to performance thrashing and degradation in the long run [17]. In this paper, we focus on the SST settings adopted by LevelDB (e.g., 2∼4MB), and provide a high performance storage system that exploits the LSM-tree structure to completely eliminate the FS index.

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 [62]. However, with the popularity of key-value stores and the wide deployments of LSM-trees, the uniform data objects from the applications benefit little from the file system abstraction. It is thus necessary to understand the overheads induced by the storage stack in order to develop a high-performance and reliable LSM-tree storage system. In this paper, we study the behaviors of different file systems in storing the LSM-tree data, and design a system that employs the LSM index to directly manage the storage space and retain the desirable properties of LSM-trees. Given the fact that key-value stores have been acting as a storage engine for relational databases such as MySQL [65], distributed stores such as MongoDB, or file systems that aim to accelerate small writes [15, 25, 54], our proposed LSM-tree Directly managed Storage, LDS, provides new opportunities for the design of storage systems to achieve significant improvement in both performance and consistency maintenance, as will be detailed in the remainder of the paper.

3 DESIGN AND IMPLEMENTATION

The purpose of LDS is to eliminate extra FS indexes and perform direct mapping between LSM-tree data and its physical location. We achieve this by designing an LSM-tree friendly on-disk layout and explicitly separating log store from chunk store. In this section we first introduce the LSM-tree customized disk layout, and how the LSM-tree data is classified. Then we describe the principles governing storage management in LDS. The section is concluded with discussions on important implementation issues.

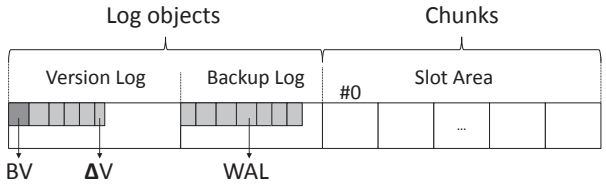

Figure 5: On-disk layout of LDS for the LSM-tree data.

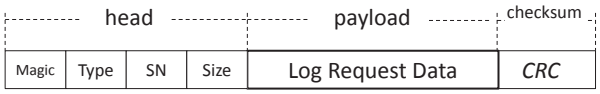

Figure 6: Log object format.

3.1 Disk Layout

The on-disk layout of LDS is depicted in Figure 5. 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 Figure 6, 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 log 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 outdated object that should be ignored.

The version log contains 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 consists of 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 Figure 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 Figure 7, therefore right

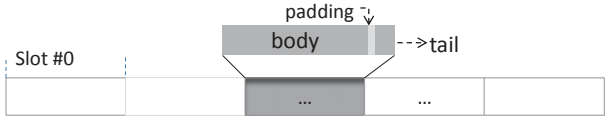

Figure 7: Chunk package. ***Padding is used to fill the gap between the body section in the left side and the tail section in the right side of a slot.***

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 [33, 55]. However, the padding area within a slot can lead to *internal fragmentation*, which is discussed in Section 3.4.

3.2 LSM-tree Managed Storage

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). In this subsection we introduce how to maintain write consistency to these three areas.

A version represents a snapshot of the backup state and the chunk organization. The organization can be changed in the future if any chunk is removed from or inserted into a level, which only happens in a compaction process. After 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. 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, the start point of live objects in the backup log is reset. The new start point is recorded in the version that represents this memory compaction. LSM-tree does not update the version in-place, instead, it commit the change (Δversion) by appending it in the version log and merges a group of Δversions with the base version to generate a new base version. Only a successfully committed Δversion guarantees the compaction results are on the LSM-tree. Otherwise, any results of the compaction are discarded as if nothing had happened. Since the new chunks are always written in free slots and are not seen by other compactions before the Δversion is committed, 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.

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

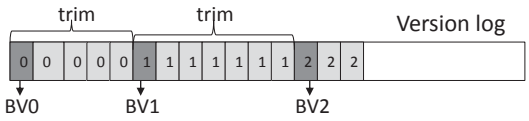

Figure 8: Trim to generate a new base version. *The Δversions share the same SN with their base version.*

Slots usage status (allocated or free) can be obtained by inspecting the base version and Δversions. On a restart the recovery process will construct a bitmap online for tracking the slots usage when executing the *trim*. 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 Δversion that has been committed, but 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 the slot to Δversion x but the slot has stored the wrong data. 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 the committed versions.

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 [33, 56], 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.

Although the online-map is a pure in-memory data structure of very small size (e.g., for a 100GB storage with 4MB slots, only 3.2KB memory is required), the allocation process can become inefficient when the space approaches full because it requires scanning more bits in the online-map to find a free one. To accelerate the allocation process, LDS also maintains a small list of the free slots to be allocated in the near future, called *partial-list*. A *find_free* thread is triggered in the background to append free slot IDs to the *partial-list* when its size drops below a threshold.

3.3 Log Write

Logs are 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 section 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. 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) [53] and is updated afterwards to identify live and outdated 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* (Figure 6), to identify live objects.

For the version log, the latest base version is used as an separation for live and outdated objects, as shown in Figure 8. A base version is persisted each time when the trim process is performed. We now assume that the version area begins with a legal object, and we will talk about how to identify if the log wraps around later on. In a recovery, LDS scans all the objects in the version log area from the beginning until garbage is encountered (i.e., illegal format or smaller SN value than the latest scanned one), 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 the Δversion that corresponds to the latest memory compaction. Recovering the memory table is achieved by scanning the backup log from the start point.

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 Figure 9, one is appended adjacent 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 outdated (by the *trim* process). As the wrapping objects have the same SN as the right-end object

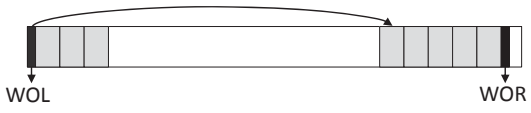

Figure 9: The *wrapping object at the left* (WOL) and the *wrapping object at the right* (WOR).

when the log wrapped around, LDS knows whether they have been outdated by finding the base version following the WOL. LDS turns to the right portion if it can 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.

Since the log areas are generally not large in size, scanning the log area will not be as costly as one may think. For instance, for a 100GB storage configured with 4MB LSM-tree chunks, a 2MB version log is sufficient with periodic trimming, and a 16MB backup log is sufficient for the two memory tables (Figure 2) with consideration of the object header overhead. Even without trimming in the runtime, which is the case in LevelDB, no more than 50MB version data is accumulated with sustained random writes filling up the storage. Besides, loading physically contiguous data blocks to memory is the best way to leverage superior sequential bandwidth of the underlying storage devices. For example, with a 150MB/s sequential read bandwidth of the commodity HDDs, it only takes 0.33 second to load the 50MB version log to memory, and performing the scanning in memory is fast. We evaluate the recovery cost in Section 4.4.

3.3.2 Commit Policy. In LDS, each compaction result (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.

Not-committed WALs of recent insertions to the backup log 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 [29], 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 [26, 47], at the cost of throughput degradation, while others may disable the backup log or flush accumulative logs to OS cache for high throughput by sacrificing durability [37, 57]. LDS inherits the LevelDB policy that flushes each WAL to the OS cache in default. Users should explicitly set the synchronizing option of an insertion request if they want the insertion to be durable. LDS guarantees that the insertions cached by the OS are committed to the backup log if a synchronizing request is received.

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

3.4 Internal Fragmentation

The padding between the body and tail sections of a chunk (Figure 7) 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 in parallel 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., 4KB). 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 [2, 45, 46, 69], 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 level L to level $L+1$. Instead of placing it in level $L+1$, the odd chunk is retained in level L and has two possibilities in the future operations. One is that it is picked by the next compaction of level $L-1$ as the overlapped chunk. The other is that it is attached to its next adjacent chunk that will participate in the next compaction of level L. 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 levels L and $L+1$, placing it in level L 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/mmap* system calls. For a synchronous request, *sync_file_range* [42] is called with the necessary flags set[3] to ensure that data is persisted. 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 implement this function in the prototype 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, version log and backup log, 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

3.6.1 Using Pre-allocated File Space. 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* and the potential file-system interference on the allocation of the physical space.

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 [42], but does not ensure that the inode data (mapping information) is synced. If the file system updates in-place [40], 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 [33, 55], 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 mapping data is not synced. Hence, fsync/fdatasync must be used to inform the file system of syncing the mapping 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 synced out 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 [12], which we will not elaborate any further in this paper.

3.6.2 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. A new version is generated to include 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.

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 32GB RAM. The operating system is 64-bit Linux 4.4. The HDD used, Seagate ST2000DM001, has a 1.8TB capacity with a 152MB/s sequential write speed, and the SSD used, Intel SSD DC S3520 Series 2.5in, has a 480GB capacity with a 360MB/s sequential write speed. Note that HDDs have a slightly faster speed on the outer cylinders. Our experiments select the partitions starting from 800GB 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) [40], f2fs (log-based) [33], and btrfs (copy-on-write) [55]. 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 4MB in LevelDB. The version log and backup log in LDS are configured to

[3] SYNC_FILE_RANGE_WAIT_BEFORE | SYNC_FILE_RANGE_WRITE | SYNC_FILE_RANGE_WAIT_AFTER

be 64MB and 16MB respectively, and the slot size is 4MB. 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).

4.2.1 Sequential Workload. Figure 10 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 Figure 11 by examining the contributions to the run time by different types of operations/events. LSM-tree

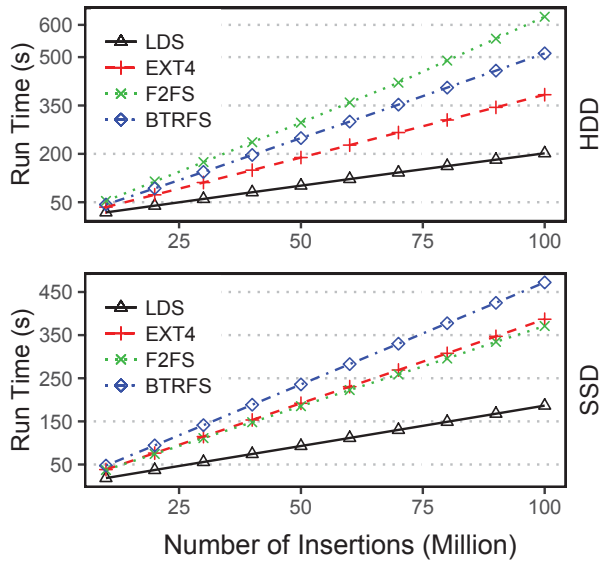

Figure 10: Sequential insertion performance (lower is better).

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

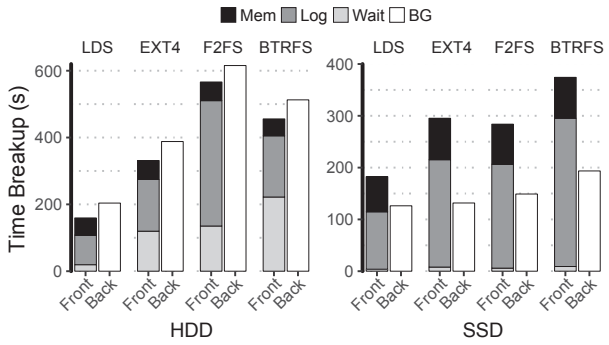

Figure 11: A breakdown of run time for sequential insertions.

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 [68].

4.2.2 Random Workload. The performance results under random workloads are shown in Figure 12, 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 Figure 13. 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 Figure 14.

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 Figure 4b). This is

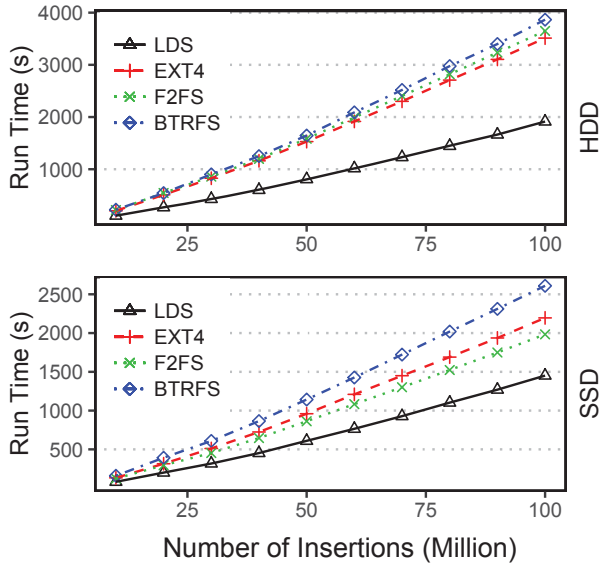

Figure 12: Random insertion performance (lower is better).

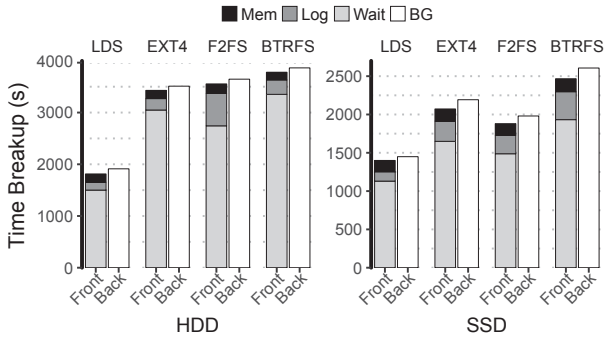

Figure 13: A breakdown of run time for random insertions.

because LDS only incurs one I/O in the backup log area. 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 [33], 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.

4.3 Read Performance

We load 1 billion random key-value pairs with a fixed size (16B key and 100B value) to set up a 100GB dataset to evaluate the read performance. The available OS cache is limited to 1GB to emulate a 100× storage/memory configuration. A larger storage system can have even higher storage/memory configuration ratio [63]. The number of concurrent threads for read are set to 4 on HDDs, and 16 on SSDs. We measure

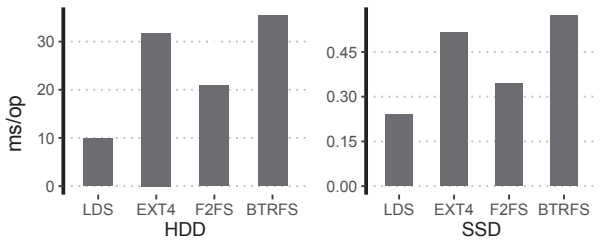

Figure 14: Insertion latency in the synchronous mode. Each WAL is guaranteed to be persisted.

the throughput and read amplification in cold cache and warm cache respectively, as shown in Figure 15.

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 [33]. 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 Figure 15c.

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 512KB, 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. [44].

4.4 Recovery

In this subsection we evaluate the cost of recovering the in-memory version from the on-disk version log. Recovering the 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 (64MB) even when we have determined all the live objects, and only after the scan finishes we begin to perform the *trim*.

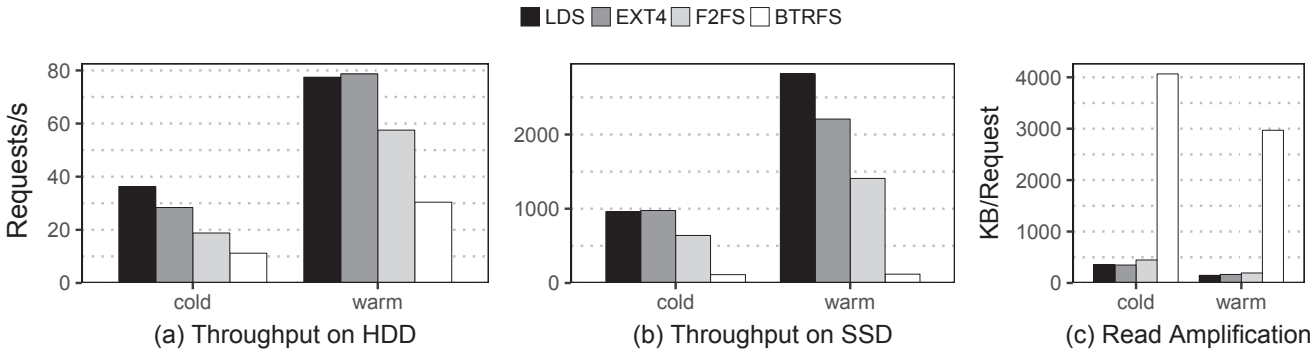

Figure 15: Read throughput and amplification on cold cache and warm cache. The number of threads for read is 4 on HDD, and 16 on SSD. The first 10×Number-of-Threads requests are accounted for cold cache results, and results on warm cache are achieved after the free cache has been filled up and the the throughput reaches the steady state. Read amplification is measured by averaged IO traffic (KB) of the requests.

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 [38] 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 3MB to 47MB).

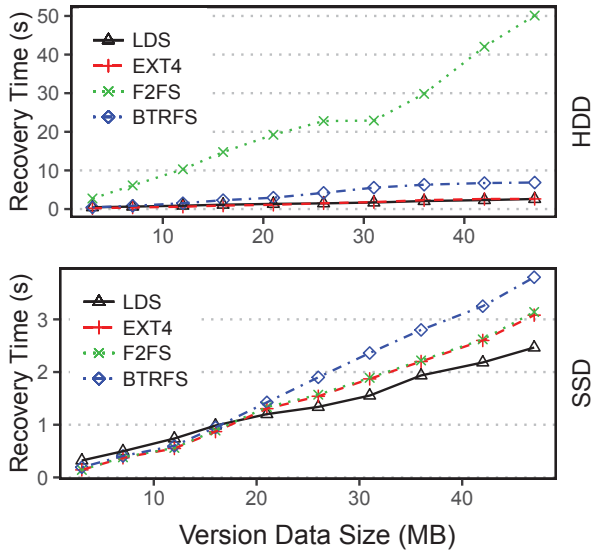

Figure 16: Recovery time as a function of the size of the accumulated version data.

Figure 16 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 3MB version data and 2 seconds for a 47MB 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 [70]. 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 [50] (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.

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 100GB 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 Figure 17. From the figure we can see that, without taking any measures to reduce fragmentation, plain-LDS has the lowest utilization,

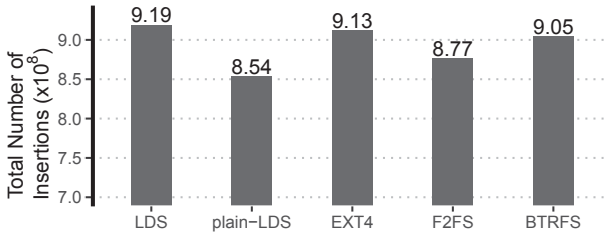

Figure 17: Total key-value pairs inserted when the system reports 'space full' on a 100GB storage device.

accommodating an amount of key-value pairs that is about 97% of f2fs, 93% of ext4 , and 94% 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 Overheads on Pre-allocated File Space

Although LDS is designed as a raw device space manager for LSM-trees, it readily supports the feature of using a pre-allocated file space with some limitations as stated in Section 3.6.1. 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[4], 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 Figure 11. Employing *mmap* to implement the flushing can significantly minimize the WAL overheads, since flushing 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.

[4]We assume that ext4 updates in-place strictly.

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 [33] 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 RELATED WORK

5.1 Write-optimized Data Structures

Traditional database systems such as SQL Server [64] employ B+trees as the backend structures, which are excellent for reads, but have poor performance for writes. Fractal-trees [4–6] 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 [46]. Write-optimized data structures have been widely used as storage engines in modern data stores [7, 19, 54, 61, 65, 66]. This paper focuses on optimizing the storage stack of LSM-tree based key-value stores.

5.2 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 [59] optimizes the write amplification in highly sequential workloads by only merging the overlapped portions of chunks. Wisckey [37] reduces the value induced amplification by moving the values out of the LSM-tree to a separate log, a similar way implemented in Bitcask [58] that uses an in-memory hash table to index the value log. LSM-trie [69] uses the hash prefix of a key to index the levels, and by allowing overlapped chunks within a level it significantly reduces the write amplification as the size-tiered compaction strategy in Cassandra [1] or the stepped-merge

mechanism [24] does. TRIAD [3] optimizes the write amplification by exploiting the skewed workloads and delaying the compaction process. PebblesDB [52] introduces the Fragmented LSM-tree (FLSM) mechanism that allows overlapped chunks within a level to avoid data rewriting, and provides tunable parameters to users for trade-off between write I/O and read latency.

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.

5.3 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 [36]. 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 [41, 62]. Stonebraker [62] examined the overheads of database systems caused by different OS components including file systems. Engler and Kaashoek [14] 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 [7, 19, 69], 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.

For example, recent work on key-value stores have persisted their data bypassing the file system because of observed performance degradation [48]. Papagiannis etc. propose the system Iris [49] to reduce the software overheads pronounced in the I/O path with low-latency storage devices. NVMKV [39] 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.

In this paper, with an in-depth study of the overheads caused by storing the LSM-trees through the representative file systems, we design the LDS that directly manages the storage with the LSM-tree structure to provide a high-performance key-value store.

5.4 New Storage Technology

New technology such as multi-stream NVM has been presented to be aware of the application-layer data streams [27], which can be an opportunity for LDS to store the LSM-tree data in an NVM-friendly way. FlashBlade [63] 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 [67] 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. [34] 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. [10] 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 [63].

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 the storage space 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.

6 CONCLUSION

In this paper we present LDS, a Log-structured-merge-tree based Direct 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. 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

We thank our shepherds Vijay Chidambaram and Russell Sears for their substantial helps to improve this paper, and thank the anonymous reviewers for their valuable comments. We also thank Mark Callaghan for his helpful feedback. This work is supported in part by the Wuhan National Laboratory for Optoelectronics Fund under Grant No.0106187015 and No.0106187027, and the US NSF under Grant No.CCF-1629625.

REFERENCES

[1] Apache. 2016. Types of compaction. http://cassandra.apache.org/doc/latest/operating/compaction.html. (2016).

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*.

[4] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. 2007. Cache-Oblivious Streaming B-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 81–92.

[5] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 546–554.

[6] Adam L Buchsbaum, Michael H Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. 2000. On External Memory Graph Traversal. In *SODA*. 859–860.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008).

[8] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 228–243.

[9] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*. San Jose, California.

[10] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 1683–1694.

[11] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288.

[12] DataStax. 2017. DSE 5.1 Administrator Guide: Changing Logging Locations. https://docs.datastax.com/en/dse/5.1/dse-admin/datastax_enterprise/config/chgLogLocations.html. (2017).

[13] Shakuntala Gupta Edward and Navin Sabharwal. 2015. MongoDB Explained. In *Practical MongoDB*. Springer, 159–190.

[14] Dawson R Engler and M Frans Kaashoek. 1995. Exterminate All Operating System Abstractions. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. IEEE, 78–83.

[15] John Esmet, Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. 2012. The TokuFS Streaming File System. In *4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '12)*.

[16] Facebook. 2013. RocksDB. http://rocksdb.org/. (2013).

[17] Facebook. 2017. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. (2017).

[18] Gregory R Ganger and Yale N Patt. 1994. Metadata Update Performance in File Systems. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association.

[19] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. http://leveldb.org. (2011).

[20] Jon Haddad. 2017. Understanding the Nuance of Compaction in Apache Cassandra. http://thelastpickle.com/blog/2017/03/16/compaction-nuance.html. (2017).

[21] Robert Hagmann. 1987. *Reimplementing the Cedar File System Using Logging and Group Commit*. Vol. 21. ACM.

[22] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-Based Solid State Drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM.

[23] InfluxData, Inc. 2017. The Modern Engine for Metrics and Events. https://www.influxdata.com/. (2017).

[24] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. In *VLDB*, Vol. 97. 16–25.

[25] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*. 301–315.

[26] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 309–320.

[27] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*.

[28] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting Storage for Smartphones. *ACM Transactions on Storage (TOS)* 8, 4 (2012).

[29] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 385–398.

[30] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux Implementation of a Log-structured File System. *ACM SIGOPS Operating Systems Review* 40, 3 (2006), 102–107.

[31] TJ Kowalski. 1990. Fsck - The Unix File System Check Program. In *UNIX Vol. II*. WB Saunders Company, 581–592.

[32] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[33] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*.

[34] Sungjin Lee, Ming Liu, Sang Woo Jun, Shuotao Xu, Jihong Kim, and Arvind Arvind. 2016. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*. 339–353.

[35] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. 2014. Assert (! Defined (Sequential I/O)). In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*.

[36] Raymond A Lorie. 1977. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems (TODS)* 2, 1 (1977), 91–104.

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*. 133–148.

[38] Ao Ma, Chris Dragga, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Marshall Kirk Mckusick. 2014. Ffsck: The Fast File-System Checker. *ACM Transactions on Storage (TOS)* 10, 1 (2014).

[39] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*. 207–219.

[40] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The New Ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux symposium*, Vol. 2. Citeseer, 21–33.

[41] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. 1984. A Fast File System for UNIX. *ACM Transactions on Computer Systems (TOCS)* 2, 3 (1984), 181–197.

[42] Michael Kerrisk. 2017. Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/sync_file_range.2.html. (2017).

[43] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[44] J. Mohan, R. Kadekodi, and V Chidambaram. 2017. Analyzing IO Amplification in Linux File Systems. *ArXiv e-prints* (July 2017). arXiv:cs.OS/1707.08514

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. 385–398.
[46] Patrick Oneil, Edward Cheng, Dieter Gawlick, and Elizabeth Oneil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Informatica* (1996).
[47] Mike Owens and Grant Allen. 2010. *SQLite*. Springer.
[48] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC '16)*.
[49] Anastasios Papagiannis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2017. Iris: An Optimized I/O Stack for Low-latency Storage Devices. *ACM SIGOPS Operating Systems Review* 50, 1 (2017), 3–11.
[50] Zachary Nathaniel Joseph Peterson. 2002. *Data Placement for Copy-on-Write Using Virtual Contiguity*. Ph.D. Dissertation. University of California Santa Cruz.
[51] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI '14)*. Broomfield, CO, 433–448.
[52] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Simultaneously Increasing Write Throughput and Decreasing Write Amplification in Key-Value Stores. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.
[53] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2015. Crash Consistency: FSCK and Journaling. http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf. (2015).
[54] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 145–156.
[55] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013).
[56] Mendel Rosenblum and John K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.
[57] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, 217–228.
[58] DJ Sheehy and D Smith. 2010. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *White paper, April* (2010).
[59] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *11th USENIX Conference on File and Storage Technologies (FAST '13)*. 17–30.
[60] Kent Smith. 2011. Garbage Collection. *SandForce, Flash Memory Summit, Santa Clara, CA* (2011), 1–9.
[61] SQLite. 2016. SQLite4: LSM Design Overview. https://www.sqlite.org/src4/doc/trunk/www/lsm.wiki. (2016).
[62] Michael Stonebraker. 1981. Operating System Support for Database Management. *Commun. ACM* 24, 7 (1981), 412–418.
[63] Pure Storage. 2017. From Big Data to Big Intelligence. https://www.purestorage.com/products/flashblade.html. (2017).
[64] ZhaoHui Tang and Jamine Maclennan. 2005. *Data Mining with SQL Server 2005*. John Wiley & Sons.
[65] Tokutek, Inc. 2013. TokuDB: MySQL Performance, MariaDB Performance. (2013).
[66] Mehul Nalin Vora. 2011. Hadoop-HBase for Large-Scale Data. In *2011 international conference on Computer science and network technology (ICCSNT)*, Vol. 1. IEEE, 601–605.
[67] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM.
[68] Ext4 Wiki. 2011. Life of an Ext4 Write Request. https://ext4.wiki.kernel.org/index.php/Life_of_an_ext4_write_request. (2011).
[69] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*.
[70] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW '14)*.