On Integration of Appends and Merges in Log-Structured Merge Trees

Caixin Gong
Alibaba Group, China
caixin.gcx@alibaba-inc.com

Shuibing He
College of Computer Science and Technology, Zhejiang University, China heshuibing@zju.edu.cn

Yili Gong*
School of Computer Science, Wuhan University, China
yiligong@whu.edu.cn

Yingchun Lei
Daowoo Times Tech. Co., China
leiyingchun@daowoo.com

ABSTRACT

As widely used indices in key-value stores, the Log-Structured Merge-tree (LSM-tree) and its variants suffer from severe write amplification due to frequent merges in compactions for write-intensive applications. To address the problem, we first propose the Log-Structured Append-tree (LSA-tree), which tries to compact data with appends instead of merges, significantly reduces the write amplification and solves the issues existed in current append trees. However LSA increases read and space amplifications. Furthermore based on LSA, we design the Integrated Append/Merge-tree (IAM-tree). IAM selects appends or merges in compaction operations according to the size of memory-cached data. Theoretical analysis shows that IAM reduces the write amplification of LSM while keep the same read and space amplification.

We implement IAM as a user library named IamDB. Experiments show that its write amplification is much less than that of LSM, only 8.71 vs. 19.00 for 1TB data with 64GB memory. Compared with nicely tuned LevelDB and RocksDB, IamDB provides 1.4-2.7× and 1.6-1.9× better write throughput, saves 12% and 10% disk space respectively, as well as the comparable read and scan performance. At the meantime IamDB achieves the most stable tail latency.

KEYWORDS

LSM-tree, Write-optimized Trees, IAM-tree, LSA-tree

ACM Reference Format:
Caixin Gong, Shuibing He, Yili Gong, and Yingchun Lei. 2019. On Integration of Appends and Merges in Log-Structured Merge Trees. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337836

1 INTRODUCTION

Extremely large data volumes become common nowadays. Cheap and numerous information-sensing devices, social networking, e-commerce and on-line gaming produce data at unprecedented rates. The Log-Structured Merge-tree (LSM-tree) [13] is widely adopted for data-intensive key-value stores, such as LevelDB[8], RocksDB[7], Bigtable[3], Cassandra[10], etc. Not only KV stores lean on LSM but traditional databases, such as MyRocks[6], try to utilize the write-optimized trees. LSM meets the challenge to effectively load, update and retrieve large amounts of data.

LSM flushes the items accumulated in memory to on-disk components in batches by a chain of compactions. Substitution of sequential reads and writes for random ones greatly improves the write performance over the classical B-tree. LSM is a multi-level tree-like structure with sorted nodes, termed tables. LSM keeps records in a node sorted by merging nodes during compacting, which helps fast data retrieval. However, facing heavy data insertions, LSM suffers from high write amplification due to frequent data read and re-writes in merges. Consequently the write throughput deteriorates and storage devices with limited write endurance are prone to wearing out. Meanwhile high write amplification decreases query performance since writes might saturate disk bandwidth and block user queries.

LSM-trie[19] and FLSM[14] are works proposed to reduce write amplification by replacing appends with merges in compactions. A node in an append tree contains multiple sorted sequences instead of a single one. Thus, scans are sacrificed due to higher read amplification. Appends do not eliminate outdated records as merges do, and accordingly they may underutilize disk space and increase space amplification. The amplifications of LSM and append trees are summarized in Table 1. The read amplifications for point reads are the same since the Bloom filters[2] are used, thus the read amplifications for scans are compared here. LSM-trie shares similar amplifications with FLSM except that it does not support scans. Due to large read amplification for scans and space amplification, append trees only work effectively in limited scenarios.

In addition, LSM-trie and FLSM have other limitations shown in Table 2, and will be discussed in Section 2.2. To address these issues, we propose the Log-Structured Append-tree (LSA-tree), which avoids the worst write case, has good sequential write performance as well as supporting scans. However LSA is still categorized as append trees and shares the common issues as shown in Table 1.

To take full advantages of merges as well as appends, this paper further presents a novel data structure, called the Integrated Append/Merge-tree (IAM-tree), which introduces merges into LSA. IAM inherits the advantages of LSA shown in Table 2.

IAM organizes KV items according to their locations in a tree. For the items in the upper levels, smaller in volume, hotter for access,

*Corresponding author.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337836

Table 1: The amplifications of LSM, LSA and IAM. LSM-trie and FLSM are append trees and share the same amplifications as LSA.

Amplification	LSM	LSA	IAM
Write	high	low	relatively low
Read (scan)	low	high	low
Space	low	high	low

Table 2: The characteristics of trees with appends.

Tree	worst write case avoided	good sequential writes	scan support
LSM-trie	yes	no	no
FLSM	no	no	yes
LSA / IAM	yes	yes	yes

and thus often cached in memory, appends are adopted. For the uncached items in the lower levels, traditional merges are maintained. For the partly cached tables in the middle level, both appends and merges are integrated in terms of the size of the memory cached data. Compared to LSM, IAM has lower write amplification and the same read and space amplifications. The flexible integration of appends and merges alleviates write amplification in contrast with LSM without sacrificing read or scan performance. With proper user configuration, IAM degenerates into LSM in the special case with no appends and into LSA with minimal merges.

The comprehensive experiments show the write amplification of IAM is much smaller than that of LSM, only 8.71 vs. 19.00 for 1TB data with 64GB memory. In addition, IAM saves 10-12% disk space and obtains the most stable tail latency. It is safe to say that IAM is a better alternative to LSM in almost every practical scenario.

In summary, this paper makes the following contributions:

(1) The Log-Structured Append-tree (LSA-tree), which substitutes appends for merges in compactions to reduce the write amplification of LSM. It has advantages over the existing append trees by avoiding the worse write case, achieving good sequential writes and supporting scans. (Section 4).
(2) The Integrated Append/Merge-tree (IAM-tree), which allows appends and merges flexibly in compactions to reduce the write amplification and keeps the same read and space amplifications as LSM. (Section 5).
(3) The implementation of LSA and IAM in IamDB. The experiments show that compared with LevelDB and RocksDB for the workloads in both YCSB and *db_bench*, IAM achieves the better write throughput, the same read performance, the comparable scan performance, the most stable tail latency and saves disk space (Section 6).

2 BACKGROUND

In this section, the basics of LSM and LevelDB are introduced. Then, the limitations of LSMs and append trees are analyzed. At last, the motivation for designing LSA and IAM is presented.

2.1 LSM and LevelDB

In LSM user records are first accumulated in in-memory components. On overflow, the records are flushed to the lower-level components in batches by a series of merge operations. Since all data on all levels are sorted, merge is one-pass processing on successive tables, LSM takes full advantage of sequential I/Os.

LevelDB is a representative implementation of LSM, whose architecture is shown in Figure 1. RocksDB shares similar architecture with LevelDB. The two memtable are memory resident, and the other nodes, called SSTables, are disk resident. There is a threshold for the number of nodes on each level other than level 0, which increases by a factor of 10 with the level index (from L_1 to L_n, n is 6 by default). The records in a node are sorted by their keys, and the nodes in a level are ordered by their disjoint key ranges.

When a record is inserted, it is first appended to an on-disk log and then inserted into memtable. Once memtable reaches its capacity threshold, 4MB by default, it will be renamed to immutable memtable. The background compaction thread transforms immutable memtable into an SSTable file in L_0. The files in L_0 are unordered with each other. When the number of files in L_0 reaches 4, the compaction thread merges these files with the key range overlapped files in L_1 and generates new files (in the size of 2MB by default) in L_1. If the data size in L_i $(1 \leq i \leq n-1)$ reaches its threshold, a file is picked for compaction with the overlapped files in L_{i+1}.

A point read to find a particular record with a certain key, usually searches the following objects in sequence: memtable, immutable memtable, all SSTables in L_0, one SSTable in L_1, ... and one SSTable in L_n, until a matched record is found, or all objects are searched and no match is found. With the Bloom filters, the metadata of a SSTable possibly containing the target record should be checked first. In this paper it is assumed that the metadata of all tables are cached. If a hit, a further disk seek is required for the record data. Besides two memtables, a scan searches every related SSTable in all levels. The number of seeks equals the number of disk objects, which cannot be helped by the Bloom filters.

Write amplification is the ratio of the actual size of data written to the secondary storage to the size of data written by users. A node to be compacted in L_i merges with the overlapped nodes in L_{i+1}, whose number is 10 on average. Accordingly, the average write amplification of a compaction between two adjacent levels is 11, and the write amplification of LSM is about $11 \times (n-1)$. Besides degrading the write throughput, the large write amplification brings more problems. First, the occupation of disk bandwidth slows down concurrent queries. Second, it seriously shortens the life span of solid-state disks (SSDs) if used.

2.2 The Issues of Append Trees

LSM-trie[19] and FLSM[14] substitute appends for merges in LSM and drastically decrease write amplification. The different designs of existing append trees have their own issues, as shown in Table 2. When a node reaches its size threshold, its records are partitioned and appended to its children. A node's children should be limited to a small number, otherwise appends turn into random writes, which violates the core idea of LSM to take full advantage of sequential I/O bandwidth to improve write performance. We call it *the worst write case* if the number of a node's children might be unbounded

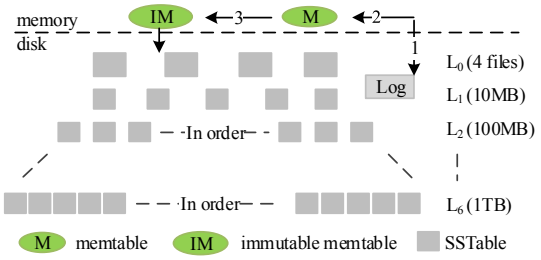


Figure 1: The architecture of LevelDB.

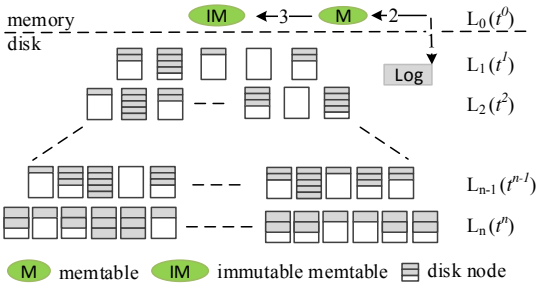


Figure 2: The architecture of LSA.

by design. FLSM leaves the avoidance of the worst write case as its future work.

With sequential writes, ordered records are continuously inserted into a tree. The key range of a table does not intersect with that of any table in the next level, and can be moved down simply by modifying the metadata of LSM without rewriting. But both LSM-trie and FLSM lose the benefit. In addition, as a hash-based method, LSM-trie does not support scans.

We propose LSA to address the issues of the existing append trees shown in Table 2. Append trees remarkably decrease write amplification of LSM by substituting appends for merges. A node in an append tree is composed of multiple sorted sequences instead of a single one. Accordingly, append trees decrease write amplification at the expense of the read amplification for scans and the space amplification. Then, based on LSA, we design IAM to select between appends and merges flexibly to obtain similar write amplification of LSA and the same read and space amplifications of LSM.

3 RELATED WORK

The Log-Structured Merge-tree is widely used in practical applications. LevelDB[8] and RocksDB[7] are state-of-the-art implementations and used in various data stores[3, 6, 10]. LSMs are also adopted in file systems[15, 18]. The widespread use of write-optimized trees shows promise for the future use of IAM.

Extensive work has been done to reduce the write amplification of LSM-trees. One way to optimize is to employ the partitioned tiering merge policy with vertical or horizontal grouping [12]. By substituting appends for merges, LSM-trie[19], FLSM-tree[14], and LWC[20] all drastically decrease write amplification at the expense of the read amplification for scans and the space amplification. LSM-trie[19], FLSM-tree[14] face the issues shown in Table 2. LWC keeps the timing and condition of LSM to evoke a compaction, which may introduce oversized or undersized tables. Dostoevsky[4] shares the hybrid strategy with IAM to mix tiering and leveling approaches to reduce merges, whose levels are divides two kinds: the largest level and the non-largest levels. While IAM divides the levels into three kinds: the append levels, the merge levels and the mixed level. and determines either appends or merges based on the size of cached data. IAM is flexible enough to reduce the write amplification but keep the same read and space amplifications.

Some techniques promote LSM and are orthogonal to IAM. PebblesDB, the implementation of FLSM[14], utilizes parallel seeks to lift scan which is effective for the scenarios where a few on-line users access SSD. WiscKey[11] reduces the write amplification of LSM by separating values from keys. FloDB[1] designs a two-level memory structure, a small hash table and a larger skip list, for the memory component of LSM to improve memory performance. SlimDB[16] presents the three-level compact index and the cuckoo filter to shrink the metadata of LSM-trie to improve read performance. NoveLSM[9] optimizes LSM to better exploit non-volatile memories. The Succinct Range Filter (SuRF)[21] almost eliminates the unnecessary seeks for the sequences without the target records in scans, which can lift very short range scans.

4 THE LOG-STRUCTURED APPEND-TREE

In this section, we introduce the design of the Log-Structured Append-tree (LSA-tree). LSA is a type of append trees which compact by appends mostly. The overall structure is presented first, and then the operations of LSA are described.

4.1 The Overall Structure

The strurture of LSA is illustrated in Figure 2. LSA consists of one in-memory level, L_0, and n on-disk levels, from L_1 to L_n. The key ranges of nodes in an on-disk level are disjoint and sorted but unnecessary to be continuous. LSA does not have an on-disk level like L_0 in LevelDB, where the ranges of nodes overlap with each other. Hence, a point read searches at most one node in each on-disk level.

The threshold of the number of nodes in an internal level L_i is t^i $(0 \le i < n)$. The actual number of nodes in L_i, denoted by N_i, usually equals t^i. Once $N_i > t^i$, the nodes in L_i will be combined to keep $N_i = t^i$. The number of nodes in L_n is smaller than t^n. Formally, a node in L_{i+1} whose key range overlaps with that of a node in L_i is called the *child node*, and the node in L_i is the *parent node*. A node has t children averagely if the children are not leaf nodes. A node in LSA has a size threshold, C_t (128MB by default). When an internal node reaches C_t, its sequences will be merged, partitioned and appended to its according children. An on-disk node is probably composed of multiple sorted sequences.

An on-disk node is a file named *MSTable* (Multiple Sequence Table) which is designed to support efficient queries by aggregating all metadata of data sequences together. The records are partitioned into 4KB blocks and filled from the beginning towards the end in the MSTable. The metadata, containing the index for the data blocks and the corresponding the Bloom filters, starts from the end and grows in the opposite direction. The middle part is a hole for future appending. The structures of data blocks and metadata are

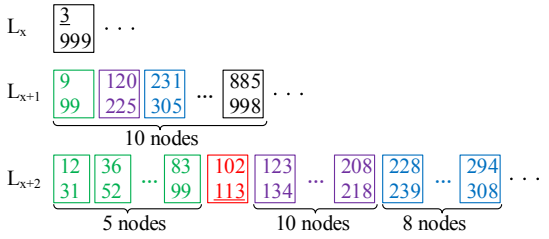


Figure 3: The structure and key ranges of nodes in a LSA-tree. A pair of numbers in a box represent the key range of a node.

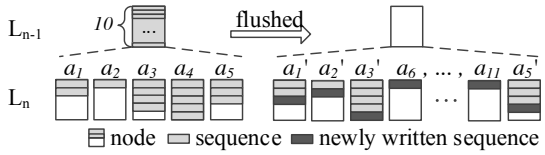


Figure 4: Flushing data to the leaf level. The full node, a_4, is merged with the partitioned records, resulting in 6 new nodes (a_6-a_{11}); the other nodes are not full and the partitioned records are simply appended to them.

the same as those of SSTables (Sorted String Table). Looking up a record in an MSTable, the clustered metadata is fetched first, then the Bloom filters help locating the data block with the target record and prevent unnecessary reads, while a scan needs to seek data blocks of every sequence in an MSTable and merges them to get the sorted result.

4.2 LSA Operations

LSA has three types of operations: *flush*, *split* and *combine*. A flush operation moves the data in a parent node to the next level to vacate space for newly coming data. However, with only flushes, the tree structure may be skewed, i.e. some nodes may have significantly more children than the others. When flushing such a parent node, the data will be partitioned and appended to its children, leading to a large number of time-consuming seeks. If the records keep being inserted into these nodes, the write performance will be continuously poor. In order to eliminate the worst write case, a split operation is introduced. Specifically, a node whose number of children reaches $2t$ splits into two and either node has half of the children. Splits and flushes may increase the number of nodes, while combines reduce nodes to keep $N_i = t^i$, i.e. the tree structure.

4.2.1 Flush. The operation that moves the data stored in a parent node in L_i $(0 \le i \le n-1)$ to its children in L_{i+1} is called a *flush*. The preconditions for flushing a full node in the L_i are as follows:

1) Its number of children is less than $2t$, otherwise the parent node will be split (will be discussed in Section 4.2.2).
2) If L_{i+1} is an internal level and none of its children is full, otherwise the full child should be flushed or split first.

Once the conditions are satisfied, a flush runs as follows.

Without children, the node is directly moved to the next level and N_{i+1} increases by 1. In a scenario of sequential writes, nodes always have no children and all records are written to disk only once. Thus the performance of sequential writes is as good as in LSM.

If the parent node has children, the records to be flushed are loaded into memory first, then are sorted by merging its sequences, and finally are partitioned and appended to the proper children. There are two cases for this situation.

In one case, the children are leaf nodes. The records are partitioned according to the key ranges of the children. If the key of a record falls out of all ranges, it is assigned to the child with the closest range. For example, L_{x+2} shown in Figure 3 is the leaf level, and the node {9, 99} with 5 children is about to be flushed, the records in the key range {9, 33} goes to the child {12, 31}. The partitioned records will be written to the corresponding children as shown in Figure 4. If the child is full, the partitioned records will merge with it, generating new children with an initial size of $\frac{C_t}{s}$, $\frac{C_t}{5}$ by default. If a child is not full, the partitioned records will be appended to the child. The key range of the child changes accordingly. For example, the node {12, 31} is extended to {10, 32} when the parent node contains the records with the keys 10 and 32.

In the other case, the children are internal nodes and not full, thus the partitioned records will be appended. When the key of a record falls in no child's range, the partition method is different from the above case. It is preferentially assigned to the child with fewer children to alleviate range skew. For example, the node {9, 99} and {120, 225} are the children of {3, 999} having 5 and 10 children respectively. When {3, 999} flushes, the records within the key range {3, 113} are appended to the child {9, 99}. Then, the range of the node {9, 99} extends, and the number of its children may get close to that of {120, 225}.

For the parent node, its key range usually remains unchanged but may be reduced after flushing. This design tries to balance the numbers of children between adjacent nodes. For example, after the node {120, 225} flushes, the left endpoint of its range interval moves right to get the same number of children as the node {9, 99}, i.e. 5. In the flush of {3, 999}, the key ranges of the two children are partitioned evenly to allow them have the same number of children, 8. To sum up, a flush alleviates range skew without extra cost.

4.2.2 Split. The effect of a flush to lessen range skew is limited, because it may only re-assign a node's children to its siblings. If several adjacent nodes all have large numbers of children, the adjustment is restricted. A *split* is evoked when a node is full and the number of its children reaches $2t$. The parent node will be rewritten into two new nodes each with half of its children. The initial key range of the new node is formed by the smallest and largest keys of the records stored in itself and its assigned children. A split prevents a node from flushing its data to more than $2t$ children and thus avoids the worst write case.

4.2.3 Combine. The constraints $N_n < t^n$ and $N_i = t^i$ $(0 \le i \le n-1)$ may be broken since flushes possibly increase N_{i+1} and splits increase N_i. Before flushes or splits, the following pre-processing should be done to guarantee both constraints are met. First, if $N_n \ge t^n$, the number of the on-disk levels, n, will increases by 1. As a result, the original leaf level becomes an internal level, and a new empty leaf level is added. Then, check whether $N_i = t^i$ $(0 \le i \le n-1)$ still holds. If $N_i > t^i$, combines will be evoked to

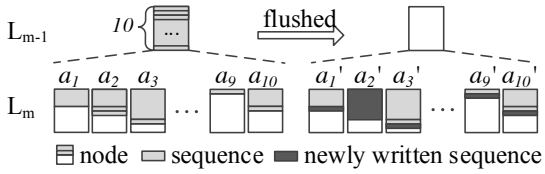


Figure 5: Flushing to the mixed level in IAM. k is set to 3 and the node a_2 is the only child with 3 sequences. Hence, a_2 are merged with its assigned data and a single sequence a_2' is generated. The other children are appended.

reduce nodes in L_i. Actually, a *combine* is a special type of flushes, by which the data of the parent are flushed to the next level and the node itself is destroyed, accordingly N_i decreasing by 1.

The node to combine needs to be selected carefully, otherwise the key ranges and children numbers of its two adjacent siblings may grow quickly, incurring frequent splits. We propose a strategy for selecting a node to combine. First, a candidate set is initialized with all the nodes with two adjacent siblings. The candidate nodes should also satisfy the equation that $T_{cn} \leq 3 \cdot t$, where T_{cn} is the number of the children covered by the combined key ranges of the candidate node and its two neighbors. For example, in Figure 3, the T_{cn} of the node {120, 225} is the number of the children covered by the range {9, 305}, i.e. 24. Since the average value of T_{cn} is $3 \cdot t$, the candidate set should not be empty. Then, the node with the smallest T_{cn} of the candidate set is picked for combining. After the target node is destroyed by combine, the ranges of the two neighbors will be adjusted evenly in the latter flush of their parent node, and their children numbers are at most $1.5t$. For example, after {120, 225} is combined, the ranges of {9, 99} and {231, 305} extend evenly in latter flush of {3, 999}, and both will have at most 12 children. As a result, neither neighbor will split immediately, which prevents splitting frequently and degrading performance.

5 THE INTEGRATED APPEND/MERGE-TREE

LSA utilizes appends to compact data instead of merges and accordingly obtains much smaller write amplification. A scan requests to check all the records within a key range. To get the required records in order, a scan seeks two memory nodes as well as all sequences in on-disk nodes and merges them. Multiple sequences in an on-disk node incur more seeks and larger read amplification. Besides, LSA may have larger space amplification due to fewer merges. In order to achieve the low write amplification as well as keep the same read and space amplifications as LSM, we allow both appends and merges for flushes and propose the Integrated Append/Merge Tree (IAM-tree) based on LSA.

5.1 The Flush Strategy

To achieve the comparable scan performance with LSM, the flush strategy in IAM aims to make a scan take at most one disk seek at each level. The determination and adjustment of the key range of a node are the same as those of LSA. The key difference lies in choosing an append or a merge according to the size of memory-cached data when a node flushes.

Table 3: The write amplification of each level after hash loading 100GB data into IAM and L_3 is the mixed level.

Level	1	2	3	4	total
k=1	1.03	1.04	3.88	0.23	6.18
k=2	1.03	1.04	2.41	0.23	4.70
k=3	1.03	1.05	1.90	0.20	4.17

5.1.1 The Simple Flush Operation. For the top levels, called *the appending levels*, the data is of smaller size, hotter for access, and cached in memory. Consequently a scan in these levels does not resort to the disk, and the flush that moves data to the appending levels remains as LSA does. The write amplification incurred by these levels is much smaller than that in LSM.

For the bottom levels, called *the merging levels*, the amount of data is much larger than the memory size. Hence, the flush strategy is similar to the compaction of LSM so that each node stores only one sequence. Specifically, the partitioned records flushed from its parent will be merged with a node and a new single-sequence node is generated. Accordingly, scans and point reads can be as good as those of LSM in these levels.

5.1.2 The Finely Tuned Flush Operation. To further reduce write amplification, *the mixed level* is introduced, denoted by $L_m (1 \leq m \leq n+1)$. The mixed level is the first level whose nodes cannot be totally cached in memory. We try to keep that a node in L_m contains at most k sequences, only one of which is not totally cached.

The procedure of flushing data to L_m is shown in Figure 5. For a child with k sequences, the partitioned records will be merged with it, and a new child containing a single sequence is generated. For a child with fewer sequences, the data from the upper level is appended. On one side, the appended sequence have more recently inserted data and are more likely to be frequently accessed. On the other side, the average size of appended sequences is $\frac{C_t}{t}$ and usually much smaller than that of a merge-generated sequence. It is reasonable to try to keep the appended sequences in memory. Once the appended and hotter sequences are cached, it only takes one disk seek if the original merge-generated sequence is requested.

Compared with the simple flush strategy, the introduction of the mixed level avoids more merges and therefore further reduces write amplification. Table 3 shows the advantage of the mixed level, L_3. When there is no mixed level ($k = 1$), the write amplification is 6.18. If $k = 2$, it is reduced to 4.70, i.e. by 23.9%. As k grows, the write amplification decreases.

5.1.3 Discussion on m and k. The parameters m and k are tuned according to the sizes of the memory and all appended sequences. If the former size is larger than the latter, it is possible to cache all appended sequences.

The memory cache size, denoted by M, is the sum of the memory cached data blocks of all MSTables. The *mincore* function is used to sample whether data blocks are memory resident. To estimate the size of all appended sequences, the size of data stored in each level, denoted by D_j $(1 \leq j \leq n)$, are collected first. Then, the average size of appended sequences in L_m with parameter k, denoted by $S_{m,k}$, is calculated. Assuming that the number of appended sequences in

a node is a random number out of 0 to $k-1$, and that the average data sizes of an appended sequence and a node are $\frac{C_t}{t}$ and $\frac{C_t}{2}$ respectively, then

$$S_{m,k} = D_m \cdot \frac{\sum_{i=0}^{k-1} i \cdot \frac{C_t}{t}}{k\frac{C_t}{2}} = \frac{D_m \cdot (k-1)}{t}. \quad (1)$$

The data size of all appended sequences is the total data size of appending levels plus that of the appended sequences in L_m, i.e. $\sum_{j=1}^{m-1} D_j + S_{m,k}$. It should be no more than the cache size M, i.e.

$$\sum_{j=1}^{m-1} D_j + S_{m,k} \leq M. \quad (2)$$

Once the inequality is satisfied, the system has enough memory to cache all appended sequences. The larger m and k leads to smaller write amplification, thus the largest m and k satisfying the inequality is preferred.

The right of the inequality can be set to a smaller value than M, say $\frac{M}{2}$, to allow the remaining memory to cache part of the merge-generated sequences. If all appended sequences are forcibly cached, a scan takes at most ons disk seek for a node in each level. Since the appended sequences are generally hotter than the merge-generated ones, it allows IAM to use the common flexible cache strategy that the hotter data is preferentially cached instead of the forcebile caching strategy.

5.2 User Operations

A user write is processed in the same way as LevelDB, that is, a record is first appended to the user log for recovery and then inserted into memtable. Once memtable reaches the capacity threshold, C_t, it will be renamed to immutable memtable ready for flushing. The processing of a delete is the same as that of an insert except that the record contains a flag indicating the deletion type instead of a value. The actual deletes and updates are deferred and fulfilled during later compactions. In merges, the outdated records are removed and the valid records remain.

A point read searches memtable, immutable memtable and the disk nodes whose key ranges cover the requested key. Since the key ranges of nodes in an on-disk level are disjoint and sorted, at most one node is searched in one level. The nodes in higher levels and the lately appended sequences in a node always have more recently inserted records, and should be searched first. Once the record is found in a sequence, the search stops. The seeks for the sequences without the target can almost be avoided by the Bloom filters.

A scan checks memtable, immutable memtable and all sequences in a node in every on-disk level and merges them to get the required records in order, which cannot be helped by the Bloom filters.

5.3 Amplification Analysis

The amplifications, including write, read and space amplifications, approximate the performance of a tree. Write amplification is the ratio of the actual size of data written to the secondary storage to the size of data written by users. Read amplification is the expected number of random disk I/Os to finish a query, assuming that the total data volume is much larger than the system memory. Space amplification is the ratio of the actually-taken storage size to the logical size of the database. The amplification characteristics of LSM, LSA and IAM are summarized in Table 1. The existing append trees share the similar amplifications with LSA because on-disk nodes in these trees also contain multiple sequences.

5.3.1 Write Amplification. The write amplifications of LSA and IAM are incurred by flushes and splits. For flushing data to an L_i, LSA only appends and the amplification is 1. For flushing data to L_n, the default initial size of a node in L_n is $\frac{C_t}{5}$, thus the amplification is close to 1. Here it is assumed to be 1 for simplicity. Accordingly, the amplification incurred by flushes equals the number of on-disk levels, n. Thus the write amplification of LSA is

$$W_{lsa} = W_{sp} + n, \quad (3)$$

where W_{sp} is the amplification brought by splits.

When IAM flushing data to an appending level, the amplification is 1. For a merging level, the partitioned data always merges with the nodes. Since the average size of the partitioned data is $\frac{C_t}{t}$ and that of a node is $\frac{C_t}{2}$, the average write amplification is $(\frac{C_t}{t} + \frac{C_t}{2}) \div (\frac{C_t}{t}) = \frac{t}{2} + 1$. For the mixed level, one merge occurs when the partitioned data is written to the node with k sequences. Assuming that the sequence number of a node in L_m is randomly picked from 1 to k, one merge occurs every k writes on average, that is, the frequency of merges in L_m is $\frac{1}{k}$ of that in a merging level. Hence, the average write amplification for a flush in L_m is $(\frac{kC_t}{t} + \frac{C_t}{2}) \div (\frac{kC_t}{t}) = \frac{t}{2k} + 1$. Summing up all levels, the write amplification for IAM is

$$W_{iam} = W_{sp} + n + \begin{cases} \frac{t}{2k} + \sum_{j=m+1}^{n} \frac{t}{2} & 1 \leq m \leq n \\ 0 & m > n. \end{cases} \quad (4)$$

With larger m and k, the write amplification decreases but it takes more memory as illustrated in formula 2.

At last, we discuss the write amplification induced by splits, W_{sp}. A split in L_{n-1} increases N_{n-1} by 1 and triggers a node in L_{n-1} to combine. After the combine, either of its neighbors has at most $1.5t$ children as described in Section 4.2.3. A node splits when its number of children increase to $2t$. Splits are invoked most frequently when either neighbor of the combined node keeps flushing and its number of children increases to $2t$. Users need to insert data of the size $\frac{0.5t \cdot C_t}{2}$ on average to make the number of children increased by $0.5t$ and a split is triggered. Since a split writes data of the size C_t, the write amplification induced by a split in L_{n-1} is $C_t \div \frac{0.5t \cdot C_t}{2} = 2 \cdot \frac{2}{t}$.

Similarly, the split of an L_{n-2} node requires the number of its children to increase by at least $0.5t$. Since it is needed for users to insert data of the size $\frac{0.5t \cdot C_t}{2}$ to add one new node in L_{n-1}, the required inserted data that triggers a split in L_{n-2} is of the size $\frac{(0.5t)^2 \cdot C_t}{2}$. Thus, the write amplification in L_{n-2} is $C_t \div \frac{(0.5t)^2 \cdot C_t}{2} = 2 \cdot (\frac{2}{t})^2$. Accordingly, the write amplification for all levels incurred by splits is

$$W_{sp} = 2 \sum_{j=1}^{n-1} (\frac{2}{t})^j. \quad (5)$$

Since $t = 10$ by default, W_{sp} is very small compared to the amplification brought by flushes. Furthermore, since flushes also have the ability to adjust the key ranges of nodes, the write amplification incurred by splits is smaller.

5.3.2 Read Amplification. Since the metadata of MSTable is clustered and of the same average size with that of SSTable in LSM, we only analyze the scenario that all metadata can be cached as in [14, 17] for clarity. A point read or scan intends to search every sequence of one node in each on-disk level. For LSM, assuming that the data stored in the levels above L_m are totally cached in memory as IAM and LSA, no disk seek for these levels is required for all three trees. For each of the mixed level and the merging levels, IAM reduces the disk seek to at most 1. Consequently, the read amplification is the same with that of LSM, i.e. $n - m + 1$. However, IAM needs additional memory to cache the appended sequences in L_m, therefore the merge-generated sequence in L_m are less likely to be cached than in LSM. For LSA, the average number of sequences in a node is $0.5t$, and the average read amplification is $0.5t \cdot (n - m + 1)$. Since $t = 10$ by default, the read amplification is 5× larger than that of IAM or LSM.

With the Bloom filters, point reads can almost skip the sequences without the target record. Specifically, allocating 14 bits per record leads to a 0.2% false positive rate. The read amplification is about 1 if the target record exists in LSA or IAM, or 0 if does not, which is the same with LSM.

5.3.3 Space Amplification. If users launch no updates or deletes, the space amplification is the same for all three trees. Otherwise, they differ. In LSA, the outdated records exist in two situations. First, a node has multiple sequences of which most records can be outdated. Second, the newer data stays in the upper levels, and many outdated records are in the lower levels. In LSM, only the second situation leads to space redundancy. For IAM, because the merging levels take up the majority of the data, the space amplification is similar to that of LSM.

6 PERFORMANCE EVALUATION

LSA and IAM are implemented in a persistent, crash-recovery and MVCC-supported key-value storage library, called IamDB. IamDB is based on LevelDB and works as either LSA or IAM with proper configuration. LevelDB does not support parallel background compaction while IamDB does as RocksDB[7]. RocksDB adopts extra optimizations for SSDs, which are complementary to IamDB.

Since multiple sequences are allowed for an on-disk node, different append trees share similar performance with the same configuration and optimizations. Since both LSA and IAM solve the issues of the existing append trees as shown in Table 2, and have the same optimization strategies in IamDB, the performance difference between the append trees and our proposed hybrid method can be precisely reflected by comparing LSA and IAM.

6.1 Experiment Setup

Our experimental platform is a server with two eight-core 2.10 GHz Intel Xeon E5-2620 v4 processors, 64GB RAM, a 200GB SSD (Intel DC S3710 series), and a 1.2TB HDD (10000-RPM SEAGATE). The operating system is 64-bit Linux 3.10 and the file system is ext4.

LevelDB and RocksDB are configured and tuned to the best of our knowledge. IamDB uses the similar refinements with LevelDB for a fair comparison. The file size in LSM is set to 64MB as suggested by RocksDB and the memtable threshold to 128MB. The threshold sizes of the levels in LSM are 640MB, 640MB, 6.4GB, 64GB... for L_0 L_1, L_2, L_3 ..., respectively. To be fair, the capacity threshold of a node in IamDB, C_t, is set to 128MB, and the average file size is 64MB. The threshold numbers of nodes for levels in IAM are 10, 100, 1000 for L_1, L_2, L_3..., respectively. Thus, the average size of an on-disk level in IamDB equals the threshold size of the corresponding level in LSM. The size of the Bloom filters for each record is 14 bits and data compression is turned off. The maximum number of open files is set to be large enough to ensure that all metadata can be cached in memory. Read-ahead of the file system is generally enabled in RocksDB in accord with LevelDB and IamDB.

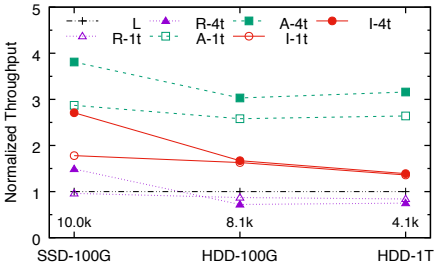


Figure 6: The throughputs of hash load.

The evaluations are based on both YCSB and *db_bench*, the default benchmark of LevelDB. A record value is set to 1024 bytes. Each workload is tested on a database for an hour as suggested by YCSB. The database is loaded with either 100GB or 1TB data. For a database with 100GB, it is newly loaded for each run. Since it takes too long to load 1TB data, all runs are on a database loaded only once. To test the out-of-RAM performance, we make only 16GB memory available for the tests with 100GB data.

In the following text, *L* represents LevelDB using a single thread for background compactions; *R-nt*, *A-nt*, and *I-nt* respectively represent that RocksDB, LSA and IAM use *n* threads for background operations. The multi-threaded performances of RocksDB and IamDB under all workloads are tested. For simplicity, only the single-threaded performance is shown unless the multi-threaded has notable difference. Figure 6, 7, 8 and 9 show the throughputs normalized to LevelDB's, and the IOPS of LevelDB are presented at the bottom. The tail latency will also be discussed.

6.2 Hash-Load Performance

Hash load is the default load method of YCSB, in which unordered records without collisions (no updates) are written into stores by insert operations. Figure 6 shows the detailed throughputs for 100GB data loaded in SSD, 100GB data in HDD and 1TB data in HDD. In general, LSA is the best for the smallest write amplifications and IAM takes the second place. The amplifications of LevelDB are 8.83, 8.71 and 14.66 for the three tests, which decrease to 3.16, 3.15 and 4.10 of LSA and 4.70, 4.72 and 8.71 of IAM. RocksDB is the largest, 9.90, 9.61 and 19.00 respectively, leading to the poorest throughputs when configured with single-threaded compaction. Here the write amplifications do not include what is incurred by writing log.

After hash loading 100GB data, both LevelDB and RocksDB have serious data overflows. The overflow makes the factor for the increase in actual data sizes of two adjacent levels less than 10. For

Table 4: The write amplification caused by each level after hash loading 1TB data for HDD.

Level	L	R-1t	R-4t	A-1t	A-4t	I-1t	I-4t
0	1.03	1.03	1.03	-	-	-	-
1	2.05	1.73	1.88	1.03	1.03	1.03	1.03
2	4.66	5.07	5.32	1.03	1.03	1.03	1.03
3	5.48	6.68	6.82	1.03	1.05	2.52	2.63
4	1.44	4.48	4.47	0.97	1.00	4.05	3.96
5	0	0.01	0.01	0.04	0.13	0.08	0.29
Sum	14.66	19.00	19.53	4.10	4.24	8.71	8.94

1TB data, LevelDB still allows serious overflows while RocksDB barely does. For example, The actual data size in Level 1 in LevelDB is 5.6× the threshold size, and that in Level 2 is 3.0×. As a result, the actual factor for level 1 and 2 of LevelDB with 1TB data is 5.4 instead of 10. Accordingly, the number of children of a node is fewer, making the write amplification smaller. RocksDB has less serious or no overflows and thus suffers larger write amplification and lower throughput. However, the overflow not only lengthens the time to reach the stable performance but deteriorates the tail latency. Specifically, the data overflow need be compacted, which is concurrent with the running workload and occupies part of the disk bandwidth. The phase involving compactions to move down all data overflows is called *the tuning phase*.

The write amplification caused by each internal level of LSA is about 1, as shown in Table 4. In IAM, level 3 is the mixed level. The write amplification of each level above level 3 is about 1, and the one of level 3 is 2.52. Level 4 is a merging level and the write amplification is larger, 4.05. All the phenomena conform to the theoretical analysis in Section 5.3.1. Most nodes in level 5 are moved directly from level 4 without rewriting, and therefore the write amplification is small. Multi-threaded IamDB and RocksDB have a slight increase in write amplification because they perform data flush or compaction more promptly.

LevelDB suffers serious bursts and stalls[17] while RocksDB has extra control for stalls[5]. For SSD, LevelDB has a very large maximum latency, 136.2s, but a good 99% latency, 1.48ms. The maximum latency of RocksDB is better, 0.80s, but the 99% latency is worse, 4.54ms. With various maximum latencies, comparing the 99% latencies is meaningless.

Without the extra control of RocksDB, LSA achieves the similar maximum latency with RocksDB and better 99% latency than LevelDB. Specifically, the maximum latency is 1.12s and the 99% one is only 0.31ms. IAM-tree falls in-between of LevelDB and RocksDB for both tail latencies, whose maximum latency is 7.4s and 99% latency is 2.67ms. The results are similar in HDD, and thus are omitted here. Accordingly, reducing write amplification not only improves throughput but also QoS. It is reasonable to infer that the QoS of IamDB will be better with the optimizations of RocksDB.

6.3 Write-intensive Workloads

YCSB defines six workloads, namely workload A, B, ... and F. Workload A and F are write-intensive ones. Workload A, the update heavy workload, has a mix of 50/50 reads and writes. Workload F, the read-modify-write workload, has a mix of 50/50 reads and reads-modifies-writes. Figure 7 shows all the results normalized to LevelDB's throughputs. For workload A and F in SSD, the conclusions for tail latencies are similar with for loads and thus the specific numbers are omitted, that is, LSA is the best and IAM takes the second place, both way outperforming LSM.

The bottleneck for HDD is with random reads and the performance of all trees is alike. However, due to serious overflows in LSM, LSA and IAM have better throughputs than LevelDB for HDD with 100GB data by 40%-50%. For 1TB data, LSA and IAM obtain 1.1 and 1.3× better throughputs. RocksDB performs similarly with IAM for no overflows. For the scenarios where the bottleneck are queries, no stalls occur, thus the maximum latencies are all small. Hence the comparison of 99% latency is meaningful. For 100GB data in HDD with workload A, the 99% latency of LevelDB is 0.31s and RocksDB is 0.37s, whereas both LSA and IAM are 0.18s. For 1TB HDD, LevelDB is 0.39s and RocksDB is 0.27s, while LSA is 0.33s and IAM is 0.26s. The results of workload F are similar with those of workload A. In a word, IAM is the best for HDD in both throughput and QoS.

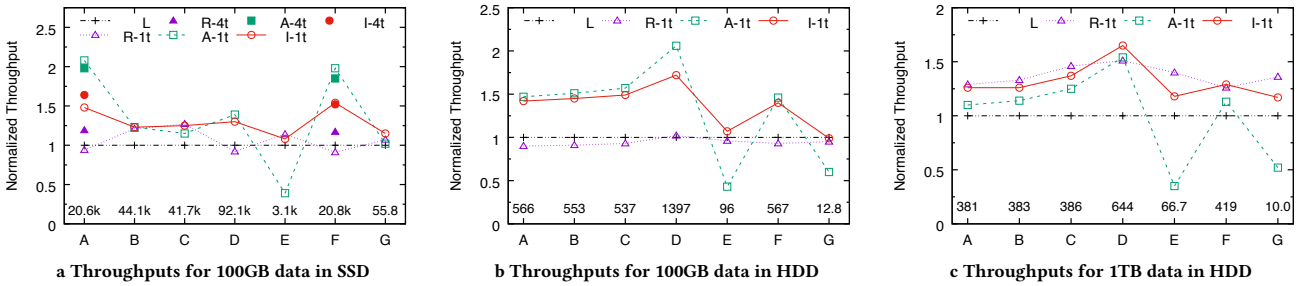
6.4 Read-intensive Workloads

Workload B, C and D are read-intensive workloads. Workload B is read heavy and workload D is read latest, both of which have a mix of 95/5 reads and writes. workload C is the read only workload.

Due to serious overflows and large write amplifications in LSM, it takes time for the system to become stable after loading. In Figure 7 the throughputs are the average performance of the default one-hour runs. The number for RocksDB in Figure 7c is also the stable performance since it has no overflows with 1TB data for HDD. SSD has higher bandwidth, thus the systems get into the stable state faster than in HDD. In Figure 8, the stable performance is used for each system, which is in favor of the LSMs.

First, we discuss the performances of IamDB and LevelDB for the read-only workload C. Since LevelDB has the longer tuning phase, LSA and IAM obtain 15% and 25% improvement for SSD, outperform it by 1.5× for HDD with 100GB data, and by 1.3× and 1.4× for 1TB data, respectively. Their stable throughputs are nearly the same, as shown in Figure 8. The throughputs for workload B are similar with those of workload C, while LSA and IAM respectively obtain 23% and 32% enhancement for SSD. The 5% writes are faster, furthermore occupy less bandwidth leading to the improvement. For HDD, the bottleneck is reads, and adding a few writes makes little difference. For the read-latest workload D, IamDB always achieves the best because it divides the latest data in top levels into smaller pieces and obtains better cache locality.

For the stable performances with the three workloads in both SSD and HDD, both IAM and RocksDB have their own advantages and the difference is small, as shown in Figure 8 and 7c. It further confirms the read performances of IAM and LSM are almost the same. However, when RocksDB needs a long tuning phase, IAM wins, as shown in Figure 7a and 7b. Generally speaking, IAM and LSA have similar throughputs, while IAM is the most stable one and always obtains the first or second best in both throughput and QoS, shown in Table 5. This is because on one side, IAM has smaller write amplification and shorter tuning phase as well as the



a Throughputs for 100GB data in SSD

b Throughputs for 100GB data in HDD

c Throughputs for 1TB data in HDD

Figure 7: The normalized throughputs under the YCSB workloads.

Table 5: The 99% latencies for queries-intensive workloads. A cell contains the latencies for 100GB SSD, 100GB HDD and 1TB HDD respectively. The top two results for each workload are shown in boldface.

	B	C	D	E	G
L	2.75ms, 0.31s, 0.37s	2.78ms, 0.32s, 0.41s	**1.78ms**, 0.20s, 0.30s	116.9ms, **1.00s**, 1.19s	0.49s, **5.49s**, 5.44s
R-1t	**2.25ms**, 0.35s, **0.24s**	**2.15ms**, 0.34s, **0.24s**	2.47ms, 0.18s, **0.19s**	138.1ms, 1.14s , **0.87s**	0.54s, 6.02s, **3.79s**
A-1t	2.87ms, **0.18s**, 0.30s	2.95ms, **0.18s**, 0.29s	1.79ms, **0.10s**, 0.21s	**86.4ms**, 1.41s, 2.17s	**0.36s**, 6.81s, 7.70s
I-1t	**2.58ms**, **0.18s**, **0.25s**	**2.58ms**, **0.18s**, **0.25s**	**1.73ms**, **0.11s**, **0.19s**	**33.6ms**, **0.82s**, **0.96s**	**0.29s**, **4.88s**, **4.16s**

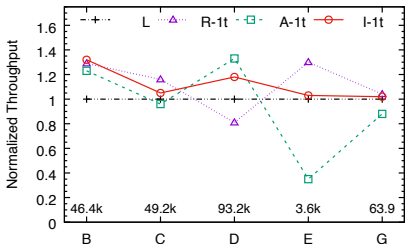


Figure 8: The stable throughputs in query-intensive workloads for 100GB data in SSD.

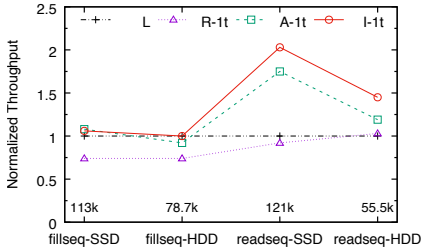


Figure 9: The throughputs of sequential load and read for 100GB data. They are nearly impervious to loaded data size.

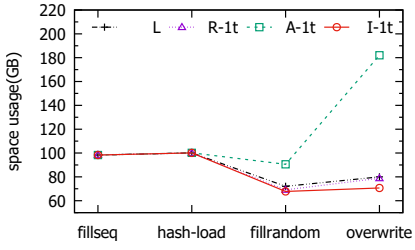


Figure 10: The space usages of write tests for 100GB data in SSD. The space usage is impervious to the media.

same read amplification; on the other side, the performance of LSA excessively depends on the Bloom filters which have false positive and lengthen the tail latency.

6.5 Short-running Scan Workloads

Workload E, the short range (0-100 records) workload, has a mix of 95/5 short range scans and writes. Workload G, the relatively long range (0-10,000 records) workload, has a mix of 95/5 relatively long range scans and writes. The queries in both workload E and G are regarded as short-running because several requests finish in only one second even for slow HDD. The performance of long range scans is investigated in Section 6.6.

Compared with LevelDB for workload E, LSA respectively suffers more than 2.3× worse throughputs because of the 5× larger read amplification, as described in Section 5.3.2. For workload G, LSA does not lose in throughputs in SSD but still has 1.7× and 1.9× smaller throughputs in HDD. On the contrary, IAM obtains small improvement for both workload E and G. The reasons lies in the same read amplification and the shorter tuning phase. For the stable performances shown in Figure 8, LSA suffers 2.9× worse throughput and loses 11% throughput, respectively, while IAM has the same throughputs with LevelDB for both workloads.

Compared with RocksDB for both workloads in SSD, IAM achieves the similar performance. For workload E in HDD with 100GB data, IAM outperforms by 11%. For workload E and G in HDD with 1TB data, RocksDB achieves 1.4× better throughputs than LevelDB and 4.0× and 2.6× better than LSA, respectively. Because of the same read amplification, IAM only loses 15% and 14% performances to RocksDB. The performance loss is due to the smaller probability that the disk seek for the mixed level is cached in memory, as mentioned in Section 5. Because RocksDB has optimized especially for SSD[7], the stable performance of workload E in SSD is better, that is, it obtains 1.3× better performance than both LevelDB and IAM.

For 100GB tests in both SSD and HDD, IAM always has the best 99% latency. For 1TB data, RocksDB achieves only a little bit better than IAM. LSA is the worst and usually much worse than the others,

while IAM outperforms LevelDB. IAM takes at least the second place in both throughput and QoS and accordingly is comparable to that of LSM.

6.6 db_bench Tests

For comprehensive evaluation, we also run two tests included in *db_bench*. One is the sequential load, inserting ordered records, and the other is the sequential read (long range scan), scanning all records in a data store. Figure 9 shows the throughputs.

For sequential load, all trees write the inserted data to disk twice, first to the log and then from immutable memtable to the corresponding on-disk level of a tree. In consequence, the throughputs of LevelDB and IamDB are nearly the same. However, RocksDB has about 25% performance drop. For sequential read, IAM is the best. The bandwidth of reading each sorted sequence from disk becomes the bottleneck, especially for SSD. Though the throughputs of all trees should be similar in theory, the long tuning phase weakens the performance of two LSM implementations.

6.7 Space Usage

Figure 10 shows the space usage, another important factor especially for expensive SSD, under sequential load, hash load, random load and overwrite tests. The first two have no updates or deletes, therefore the space usages are the same. Since random load involves many updates and overwrite only contains updates, the space usages differ. The throughputs of random load and overwrite are similar with that of hash load and hence are not shown. Compared with LevelDB, RocksDB has fewer overflows and accumulates fewer outdated records. Specifically, it saves 4.1% and 1.5% space after random load and overwrite. IAM has the similar space amplification with LSM, as discussed in Section 5. In practice, IAM saves space because it has no data overflows. It respectively saves 6.0% and 11.6% spaces compared with LevelDB and saves 2.0% and 10.3% with RocksDB. As analyzed theoretically, LSA takes much more space, 25.8% more for random load and 2.3× more for overwrite due to its larger space amplification.

6.8 Discussion on Append Trees

Section 4 describes how LSA solves the issues of existing append trees shown in Table 2. LSA/IAM avoids the worst write case since the number of children of a node cannot exceed 20. The performance of LSA/IAM for sequential write is good because no records need rewrites, which is verified in Section 6.6. The performance of FLSM for sequential write is also tested under the default configuration for 100GB data in SSD. The records are always rewritten when compacted to a level, and its write amplification is 6.42. The IOPS is 16.7k, which is 6.7× worse than LevelDB and IamDB. LSA/IAM supports scan as planned. As the experiments show, LSA solves the issues of the existing append trees, so is IAM since it is designed based on LSA.

7 CONCLUSIONS

This paper proposes Log-Structured Append-tree (LSA-tree), based on which further proposes Integrated Append/Merge-tree (IAM-tree). Compared with LSM for both HDD and SSD, IAM gains much better performance for loads and write-intensive workloads, the same (even better) performance for read-intensive workloads and matches LSM for short-running scans. In addition, IAM obtains the most stable tail latency and saves 10-12% disk space in the overwrite test. The smaller write amplification of IAM may lengthen the life span of SSD. In a word, IAM outperforms LSM in many aspects and is a better alternative to LSM in almost every practical scenario.

LSA is a special case of IAM with minimum merges. The write amplification increases merely by about 1 when the data stored increase by a factor of 10. LSA gains the best performance for writes but has very poor performance for short-running scans and takes much more space for overwrite. Thus, LSA is suitable for the scenarios with few short-running scans, updates and deletes.

8 ACKNOWLEDGEMENTS

This research was sponsored by the National High Technology Research and Development Program of China, grant 2017YFC08038, and the National Science Foundation of China, grant 61572377 and 61572373.

REFERENCES

[1] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking memory in persistent key-value stores. In *Proceedings of EuroSys'17*. 80–94.
[2] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[3] 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. *TOCS* 26, 2 (2008), 4.
[4] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of SIGMOD'18*. 505–520.
[5] Facebook. 2017. Write stalls. https://github.com/facebook/rocksdb/wiki/writestalls.
[6] Facebook. 2019. MyRocks. http://myrocks.io/.
[7] Facebook. 2019. RocksDB. https://rocksdb.org/.
[8] Google. 2019. LevelDB. http://leveldb.org/.
[9] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea ArpaciDusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of ATC'18*. 993–1005.
[10] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[11] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpacidusseau, and Remzi H Arpacidusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of FAST'16*. 133–148.
[12] Chen Luo and Michael J. Carey. 2018. LSM-based Storage Techniques: A Survey. https://arxiv.org/abs/1812.07527.
[13] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[14] Raju Pandian, Kadekodi Rohan, Chidambaram Vijay, and Abraham Ittai. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of SOSP'17*. 497–514.
[15] Kai Ren and Garth A Gibson. 2013. TABLEFS: enhancing metadata efficiency in the local file system. In *Proceedings of ATC'13*. 145–156.
[16] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. In *Proceedings of the VLDB Endowment*, Vol. 10. 2037–2048.
[17] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of SIGMOD'12*. ACM, 217–228.
[18] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment* 5 (2012), 1004–1015.
[19] Wu Xingbo, Xu Yuehai, Shao Zili, and Jiang Song. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of ATC'15*. 71–82.
[20] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *Proceedings of MSST'17*.
[21] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of SIGMOD'18*. 323–336.