Towards Read-Intensive Key-Value Stores with Tidal Structure Based on LSM-Tree

Yi Wang[1], Shangyu Wu[1], Rui Mao[1,2]
1. The National Engineering Laboratory for Big Data System Computing Technology, Shenzhen University
2. Guangdong Province Engineering Center of China-Made High Performance Data Computing System, Shenzhen University
Shenzhen, 518060, China
yiwang@szu.edu.cn, shangyuwu1006@gmail.com, mao@szu.edu.cn

***Abstract*—Key-value store has played a critical role in many large-scale data storage applications. The log-structured merge-tree (LSM-tree) based key-value store achieves excellent performance on write-intensive workloads which is mainly benefited from the mechanism of converting a batch of random writes into sequential writes. However, LSM-tree doesn't improve a lot in read-intensive workloads which takes a higher latency. The main reason lies in the hierarchical search mechanism in LSM-tree structure. The key challenge is how to propose new strategies based on the existing LSM-tree structure to improve read efficiency and reduce read amplifications.**

This paper proposes *Tidal-tree*, a novel data structure where data flows inside LSM-*tree* like *Tidal* waves. Tidal-tree targets at improving read efficiency in read-intensive workloads. Tidal-tree allows frequently accessed files at the bottom of LSM-tree to move to higher positions, thereby reducing read latency. Tidal-tree also makes LSM-tree into a variable shape to cater for different characteristic workloads. To evaluate the performance of Tidal-tree, we conduct a series of experiments using standard benchmarks from YCSB. The experimental results show that Tidal-tree can significantly improve read efficiency and reduce read amplifications compared with representative schemes.

***Index Terms*—Storage system, key-value store, LSM-tree, read amplifications, write amplifications**

I. Introduction

Persistent key-value (KV) store has been a widely used technique that maps a collection of objects or records to the corresponding data. With higher performance, better scalability and more flexibility, KV store becomes an efficient choice for many data-intensive applications to achieve excellent performance, including web indexing, e-commerce, social networking, and online gaming. KV store enables various operations to speed up the processing of applications, such as quick insertions or updates, flexible queries for point or range and efficient indexing.

The log-structured merge-tree (LSM-tree) [1] is one of most preferred data structures in key-value store which is often adopted by several large Internet companies for large-scale data applications, such as BigTable [2] and LevelDB [3] at Google, RocksDB [4] and Cassandra [5] at Facebook, HBase [6] at Apache, PUNTS [7] at Yahoo!. Due to the outstanding performance on write-intensive workloads, LSM-tree has become a standard technique. The main idea of LSM-tree is to maintain a batch of random writes in a buffer waiting to be flushed into storage devices without changing its sequentiality. In other words, those random writes are stored as key-value pairs in the buffer. When the buffer is full, these key-value pairs will be flushed into storage devices in the form of files. All files are organized by a hierarchical structure in storage devices. In order to enable efficient lookups, LSM-tree maintains those files organized in sorted order.

Although LSM-tree significantly improves write efficiency, as a trade-off, read efficiency is relatively low. In previous work, lots of techniques are proposed to reduce write amplifications by reducing the amount of data involved in compaction operations [8], [9], [10], [11], [12]. However, the more compactions occur, the more I/O operations are involved. Other works combine LSM-tree with novel external structures to improve read efficiency [13], [14], [15]. In these works, the searching process inside LSM-tree is not accelerated. LSM-tree is originally designed for hard disk drives (HDDs), and it is friendly to HDDs. With the development of new storage devices, some researchers have combined LSM-tree with emerging storage devices such as flash memory [16], [17]. These techniques focus on how to take advantage of the characteristics of storage devices to improve read or write performance. However, the characteristics of different workloads should also be considered.

This paper presents *Tidal-tree*, a novel data structure for read-intensive key-value stores. The objective is to make data inside LSM-tree move in both directions like tidal waves. In order to achieve the objective, Tidal-tree firstly identifies frequently accessed files. Tidal-tree then calculates the destination of these files in the upper layers in the LSM-tree. Tidal-tree uses a *floating* or moving up operation to re-allocate those files to the destination. This can effectively improve read efficiency and reduce read amplifications in read-intensive workloads. To avoid extra overheads caused by the floating process, Tidal-tree adopts a stretching mechanism to reshape LSM-tree. The stretching mechanism changes the shape of LSM-tree based on the characteristics of different workloads.

To evaluate the effectiveness of Tidal-tree, we conduct a series of different characteristic workloads generated by the standard database evaluation tool Yahoo! Cloud Serving Benchmark (YCSB) [18]. Tidal-tree is compared with representative schemes [3], [11] in terms of latency, read or write amplifications, the amount of data involved in compaction operations, and the number of times to access LSM-tree. Experimental results show that Tidal-tree can effectively achieve over 25% reductions in latency. Moreover, Tidal-tree can significantly reduce read amplifications by over 65%.

978-1-7281-4123-7/20/$31.00 ©2020 IEEE

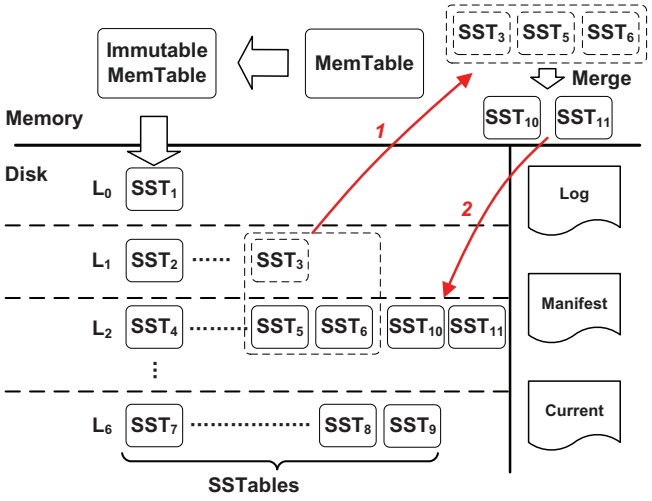


Fig. 1. The architecture of LevelDB.

The rest of this paper is organized as follows. Section II introduces system architecture used in the paper, and presents the motivation of this paper. Section III presents the proposed Tidal-tree in detail. Section IV shows the experimental results. Finally, in Section V, we conclude this paper and discuss the future work.

II. Background and Motivation

A. LSM-tree

LSM-tree adopts a log structure to convert a batch of random writes to sequential writes, resulting in high write performance. LevelDB [3] is one of the most popular key-value store databases based on LSM-tree. Figure 1 shows the overall structure of LevelDB. LevelDB divides the memory into two parts called `MemTable` and `Immutable MemTable` implemented by SkipList. In disk, all files called `SSTable` (Sorted String Table) are organized in a multi-level structure, including 7 levels denoted by $L_0, \ldots, L_6$, respectively.

Each SSTable consists of several data blocks, an index block, and several meta blocks. Data blocks contain key-value pairs. The index block indexes data blocks. Meta blocks include extended functions such as bloomfilter. The footer block indicates the location of the index block and meta blocks. There are some other auxiliary files such as `Log` files for the recovery, `Manifest` files for storing the metadata and the `Current` file for indicating the current version of the database. Each level has a limit on the number of SSTables where 4 is for Level L_0 while 10^i for Level $L_i (i > 0)$. Except for Level L_0, there is no key overlap between SSTables at each level.

When a new write operation (a delete is treated as a special update with a delete flag) comes, it is inserted into the MemTable or replaces the former one if its key exists. When the MemTable is full (4MB as a default), it is converted to an Immutable MemTable which is read-only. Then a new MemTable is created for following operations. The Immutable MemTable is gradually dumped into the disk in the form of SSTable and the SSTable is appended to the head of the file list in Level L_0. The dumping process is done in the background benefited from a version control mechanism of LevelDB, and the new version is merged with the current version after the dumping process is finished. When the number of files in each level is beyond the limit number, a *Compaction* operation is triggered.

Figure 1 shows an example of how a compaction operation operates. The compaction operation triggered in Level L_1 first selects a victim file (SST_3) in Level L_1 and several files (SST_5 and SST_6) in Level L_2 whose key range have an overlap with the victim file. These SSTables are loaded into the memory (Step 1) and merged into a series of new SSTables (SST_{10} and SST_{11}). Subsequently, SST_{10} and SST_{11} are inserted into Level L_2 (Step 2). When searching for the value of a key, LevelDB first searches the MemTable and Immutable MemTable in the memory. If the key does not exist, LevelDB searches the key from Level L_0 to Level L_6 until it is found. Due to the compaction operation and the hierarchical structure, the first found key-value pair is the latest one.

B. Motivation

Read and write amplifications are two major problems in LSM-tree. Several techniques have been proposed to reduce write amplifications while few focus on read amplifications. Read amplifications can lie in two factors: First, the hierarchical structure and the searching mechanism in LSM-tree can deteriorate read efficiency. For example, LevelDB needs to check SSTables for at most 14 times in the worst case [11]. Especially, in read-intensive workloads, searching keys at the bottom layers of LSM-tree will lead to tremendous latency due to several failed attempts before the key is found. Second, the compaction operation not only causes write amplifications but also increases read amplifications. A compaction operation in LevelDB often involves over 5 SSTables in the merging process which causes extra timing overhead [16].

To improve read efficiency while maintaining the same write efficiency, firstly we intend to reduce the number of times to search keys in the hierarchical structure. We observe that the data flow in LSM-tree is always in one single direction, from top level to bottom level driven by compaction operations. To find the data stored in the bottom level has to traverse the LSM-tree. Different from the conventional LSM-tree, we believe that the bottom-up movement of data in LSM-tree can improve read efficiency during read-intensive workloads. We also observe that, most compaction operations are triggered by moving data from the upper level to the bottom level. Since the conventional LSM-tree restricts the number of SSTable files in each layer, the floating or moving up operation may potentially cause more compaction operations. It is essential to relax the constraint by dynamically modifying the limited number of SSTable files in each level. These observations motive us to propose a novel data structure for read-intensive key-value stores based on LSM-tree.

III. Tidal-tree: A Tidal data flow structure based on LSM-tree

A. Overview

There are several challenges to improve the read efficiency based on the existing LSM-tree structure. First, the hierarchical file organization and the compaction operation signif-

icantly improve write efficiency at the cost of reducing read efficiency. If the hierarchical file organization is changed or the compaction operation is replaced, the great write efficiency may be lost. How to design a new strategy based on the existing structure and mechanism without losing a lot of write efficiency becomes a key point. Second, the characteristics of different data also have a great influence on the storage structure. For example, the LSM-tree structure prefers write-intensive workloads. It is difficult to propose a structure that performs well on all different characteristic workloads, but designing a structure for specific characteristic workloads seems to be a viable solution.

This section presents *Tidal-tree*, which is designed to optimize read efficiency among read-intensive workloads. First, Tidal-tree presents a new strategy called floating process to move some bottom-level SSTables to the upper levels. Second, Tidal-tree identifies read and write characteristics of workloads and presents a new mechanism called the stretching mechanism to adjust the shape of LSM-tree for a better read and write efficiency.

B. Floating Process

As shown in Figure 2, when an SSTable SST_8 in Level L_6 is frequently accessed and exceeds the floating threshold of Level L_6, a floating process is triggered to move SST_8 to the upper level (Level L_1) to reduce read latency (Step 1). Before moving, the floating process firstly calculates the destination level of SST_8 (i.e., Level L_1). If there exists obsolete or redundant copy of the old data in its passing level (i.e., $L_i(2 \leq i \leq 5)$), these copies will be removed during the floating process (Step 2). The rest data of SST_8 is merged with SST_3 (Step 3) and further divided into several new SSTables (SST_{10} and SST_{11}) to be inserted into Level L_1 (Step 4).

From the whole floating process, three problems should be addressed: 1. How to identify the SSTable that needs to be moved to an upper level? 2. If an SSTable is selected to perform floating process, which level does it need to move to? 3. How to remove the old data and merge the rest data into the destination level during the floating process?

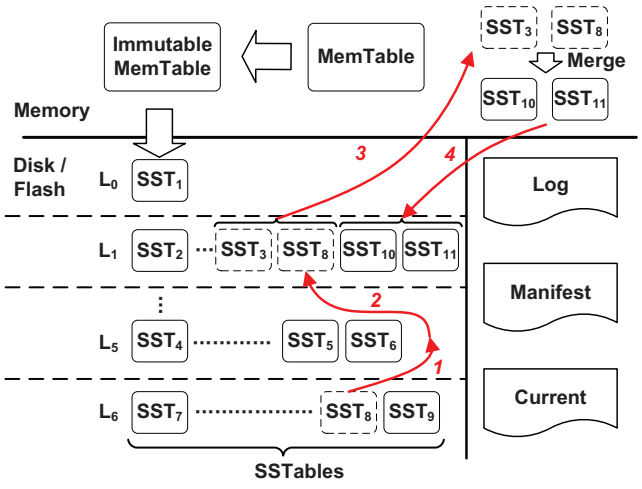


Fig. 2. The architecture of Tidal-tree and the floating process.

1) Identify the SSTable that needs to be moved: In the default LSM-tree structure, all SSTables in the hierarchical structure are organized in chronological order. The new data will be placed in the upper level in the hierarchical structure. Once the data is allocated in the bottom level, the read request will not change its level. The frequently accessed SSTable at the bottom level will cost extra read overhead, which greatly increases read amplifications.

Our aim is to move these frequently accessed SSTables to its upper level. First, we record the access frequency denoted as $f_{i,t}$ for each SSTable, where i and t denote that the SSTable is the t-th SSTable SST_t^i in Level L_i. Second, the SSTable with the access frequency $f_{i,t}$ will be moved to the upper level if it satisfies the following condition,

$$f_{i,t} \geq min(f_{i-1,max}) \times \frac{\gamma}{\eta} \tag{1}$$

, where $i > 0$, $f_{i-1,max}$ is the maximum access frequency of SSTable in Level L_{i-1}, γ is a constant parameter that controls the frequency of the floating process, and η is the ratio of read to write for the current workload.

2) Determine the destination level: SSTables normally have different access frequency. The allocation of SSTable should consider its access frequency. In Tidal-tree, SSTables are organized not only in chronological order but also by access frequency. When an SSTable is ready to move, we calculate the destination level of this SSTable based on its access frequency.

Assuming the SSTable SST_t^i is selected to perform floating process and L_j is the destination level, we calculate the total time cost of searching SSTable SST_t^i for two cases: (1) moving up SST_t^i and (2) not moving up the SST_t^i, respectively. The time cost includes the hierarchical searching process involved in future read operations. Especially, the cost of the floating process is also included in the case of moving up SST_t^i. We use T_R, T_W to denote the time cost for each page read and page write operation in flash memory.

- **Not moving up SST_t^i:**

$$T_{NM} = f_{i,t} \times 3 \times T_R \times (4 + i) \tag{2}$$

T_{NM} includes the time cost of the hierarchical searching process. In the worst case, each level has 1 SSTable that needs to be checked except for 4 SSTables at most in Level L_0. Therefore, there are $4 + i \times 1$ SSTables in total waiting to be checked. For each SSTable, at most three I/O operations are needed: reading a filter block, reading a index block and reading a data block. Each I/O operation involves in reading a physical page in flash memory. Based on the access frequency $f_{i,t}$, SST_t^i will be accessed $f_{i,t}$ times in the future.

- **Moving up SST_t^i:**

$$T_M = f_{i,t} \times 3 \times T_R \times (4+j) + T_R \times (1 + \sum_{l=j}^{i-1} c_l) + T_W \times c_j \tag{3}$$

T_M includes the time cost of the hierarchical searching process and the floating process. The calculation of the time cost of the hierarchical searching process is similar

Algorithm III.1 Moving an SSTable from Level L_i to Level L_j.

Input: The SSTable SST^i in Level L_i, SSTables $\{SST^l_1, \ldots, SST^l_{c_l}\}$ with a key overlap in Level L_l $(i-1 \leq l \leq j)$.
Output: None.

```
1: Create an iterator It^i of SST^i, then delete SST^i.
2: for l from i − 1 to j do
3:     Create all iterators {It^l_1, ..., It^l_{c_l}} of {SST^l_1, ..., SST^l_{c_l}}
       with a key overlap in Level L_l.
4:     Remove the old data in It^i compared to {It^l_1, ..., It^l_{c_l}}.
5: end for
6: Merge It^i and {It^j_1, ..., It^j_{c_j}}, then create new SSTables and
   write them back to Level L_j.
7: if Level L_j needs to do a compaction operation then
8:     Do a compaction operation in Level L_j.
9: end if
```

to not moving up SST^i_t. We assume that there are c_l $(j \leq l \leq i-1)$ SSTables with a key overlap in the SST^i_t's passing levels. Including reading the SST^i_t, the total number of reading SSTables is $1 + \sum_{l=j}^{i-1} c_l$. For each SSTable, one I/O operation needs to be performed: reading a data block. In Level L_j, SSTables need to be merged in memory, then they are written back to flash memory. Therefore, the time cost of the floating process is $T_R \times (1 + \sum_{l=j}^{i-1} c_l) + T_W \times c_j$.

Then, we define $T_d = T_{NM} - T_M$ to indicate the reduced time cost after moving SST^i_t up. If T_d is negative, it means that the floating process causes the extra time cost. $T_W \approx \alpha \times T_R$, T_d can be simplified,

$$T_d = T_R \times \left(3 \times f_{i,t} \times (i-j) - \sum_{l=j}^{i-1} c_l - \alpha \times c_j - 1 \right) \quad (4)$$

, where $0 \leq j \leq i-1$. Since the maximum value of j is 7, the largest T_d can be found by enumerating j.

3) Remove the old data and merge the rest data during the floating process: There are two properties in LSM-tree which ensure the correctness of searching data:

- There is no key overlap between SSTables at each level except Level L_0.
- The newer a key-value pair is, the upper level it is in.

In order to maintain these two properties, we need to remove the old data in SST^i_t during the floating process, then merge the rest data in SST^i_t into the target level. The removing process reads SSTables with a key overlap into memory and removes the old data in SST^i_t. Especially, at the target level, the old data in SST^i_t needs to be removed and the rest data needs to be merged to create new SSTables. Although the overhead of removing and merging process cannot be neglected, it has been taken into consideration when determining the destination level.

Algorithm III.1 presents the floating process of moving SST^i from Level L_i to Level L_j. SSTables with a key overlap in SST^i's passing levels will be read into memory for removing the old data. Each SSTable is associated with an iterator for enumerating key-value pairs. Algorithm III.1 creates an iterator It^i of SST^i in memory, then creates iterators of SSTables with a key overlap while looping from Level L_{i-1} to Level L_j. The removing process can be implemented by a multi-way algorithm. After all the old data of It^i is removed, Algorithm III.1 merges the rest data of It^i into Level L_j. To avoid breaking the constraint in Level L_j, Algorithm III.1 checks whether a compaction operation in Level L_j should be triggered.

C. Stretching Mechanism

Frequent floating processes will lead to frequent compaction operations which cause the degradation of efficiency. Although we have set a parameter called γ to control the frequency of the floating process, there is still a potential ping-pong conflict of moving SSTables up and down. The main reason is that each level has its constraint on the number of SSTables. For example, the maximum number of SSTables in Level L_0 is 4. It means that the compaction operation will be triggered when there are more than 4 SSTables in Level L_0. To avoid such a potential problem, we propose a stretching mechanism to break the constraint.

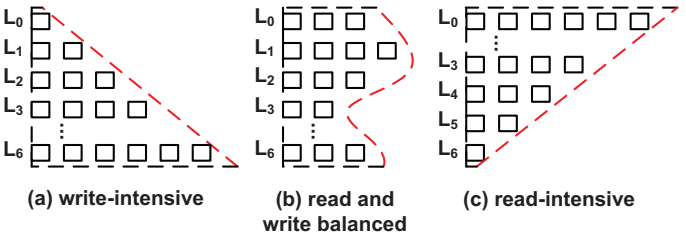


Fig. 3. Adjust LSM-tree shape to adapt to different read and write characteristic workloads.

Figure 3 shows various shapes adjusted for different characteristic workloads, and the red dashed line is the constraint. Due to the excellent write efficiency of original shape of LSM-tree showed in Figure 3 (a), we adopt the shape for write-intensive workloads. The shape showed in Figure 3 (c) is better for read-intensive workloads, since more data can be found in upper levels. For other workloads, the shape depends on the ratio of the number of read operations to the number of write operations. Figure 3 (b) is applied for read and write balanced workloads.

The parameter δ_i represents the amount of changes of the constraint in Level L_i, the parameter θ_i represents the maximum number of SSTables in Level L_i, and parameter ρ_i represents the original number of SSTables in Level L_i. We design a formula to adjust LSM-tree's shape,

$$\theta_i = \rho_i + \eta \times (7-i) \times \delta_i \quad (5)$$

, where η is the read and write ratio. The stretching mechanism adjusts the shape of LSM-tree according to θ_i whenever η changes.

IV. Evaluation

A. Experimental Setup

We conduct experiments using various workloads from *Yahoo! Cloud Serving Benchmark*[18]. YCSB is a popular database evaluation framework designed to provide a series of common benchmarks to facilitate the comparison of the performance of new generation cloud data service systems with

TABLE I
THE LATENCY FOR LEVELDB [3], WISCKEY [11], AND OUR TIDAL-TREE

Workloads	R:U:I	LevelDB (μs)	Wisckey (μs)	Tidal-tree (μs)
workload1	1:0:0	512,650,125	577,589,925	381,452,050
workload2	9:1:0	445,705,625	372,636,825	310,519,975
workload3	9:0:1	636,266,525	546,240,575	417,856,825
workload4	4:1:0	526,766,325	364,671,700	282,732,375
workload5	4:0:1	844,639,500	514,764,550	360,232,425
workload6	1:1:0	776,803,150	284,216,400	159,527,250
workload7	1:0:1	1,954,055,575	476,449,925	286,489,775
workload8	2:1:1	1,393,392,625	341,406,925	204,750,475
workload9	1:4:0	1,019,924,925	205,826,400	195,903,075
workload10	1:0:4	3,218,374,125	503,556,000	406,143,000
workload11	0:1:0	1,195,470,825	153,969,300	153,969,300
workload12	0:0:1	4,872,872,300	563,718,400	563,718,400

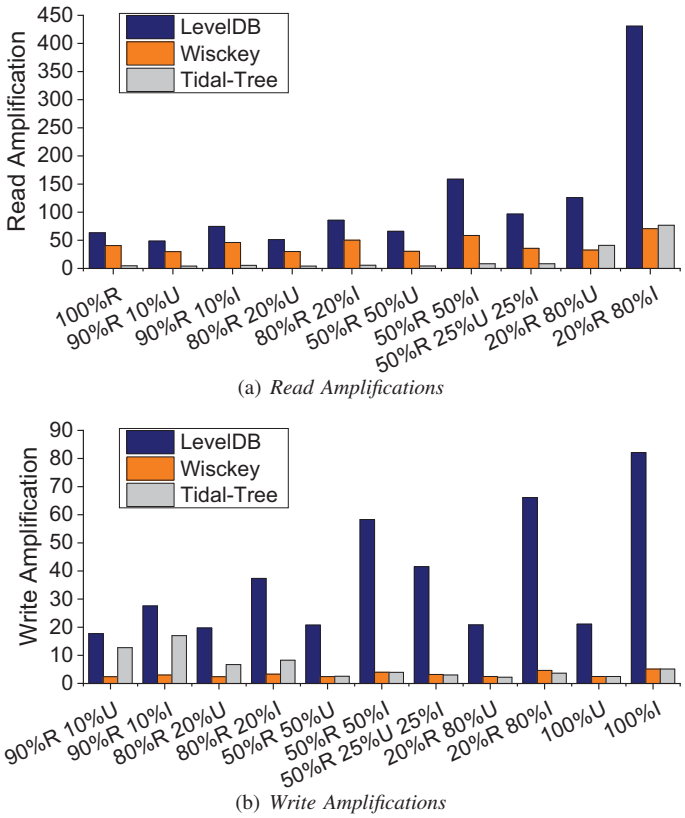
different read and write characteristics. There are two phases to generate YCSB workloads: a loading phase which involves massive insert operations, and a running phase composed of several read, insert and update operations.

We have collected 12 different workloads that satisfy *Zipfian* distribution. Each workload includes 100,000 insert operations during the loading phase and 500,000 different operations during the running phase. Among them, workloads 1-5 are read-intensive workloads including at least 80% read operations and workloads 9-12 are write-intensive workloads including at least 80% write operations. Workloads 6-8 are read and write balanced workloads. The size of the key is 16 Bytes and the size of the value is 1 KByte. We simulated a Intel 3D NAND flash memory chip. The time costs for a page read operation, a page write operation, and a block erase operation take 75 μs, 1250μs, and 5 ms, respectively. We compared Tidal-tree with two representative schemes, Wisckey[11] and LevelDB[3]. Wisckey is a representative LSM-tree management scheme with key-value separation. LevelDB is the standard LSM-tree management scheme that is widely used in commercial database systems. Therefore, they are selected for comparison.

B. Results and Discussion

1) Latency: The latency reflects the time cost of conducting a series of operations. Table I presents the latency of LevelDB[3], Wisckey[11] and the proposed Tidal-tree. For the column "R:U:I", "R" , "U" , and "I" represent the normalized number of read operations, update operations, and insert operations, respectively. From the experimental results, Tidal-tree reduces the latency by 22.88% and 65.65% on average compared to Wisckey and LevelDB, respectively. For read-intensive workloads, the reductions are 25.32% and 38.79%, respectively. For write-intensive workloads, Tidal-tree has almost the same latency as Wisckey, and Tidal-tree can still achieve a significant reduction compared to LevelDB. The main reason is that Tidal-tree will move frequently accessed key-value pairs to upper levels in order to reduce read latency. Meanwhile, Tidal-tree sets a parameter to limit the frequency of moving SSTables, avoiding the increase of extra latency for write-intensive workloads.

2) Read and Write Amplifications: Read and write amplifications indicate the ratio between the amount data read from or written to the underlying storage device and the amount of data requested by the user. Read and write amplifications are major



(a) *Read Amplifications*

(b) *Write Amplifications*

Fig. 4. Read and write Amplifications of LevelDB [3], Wisckey [11] and Tidal-tree.

problems in LSM-tree. Read amplifications are caused by the mechanism of searching keys in LSM-tree. A compaction operation will lead to a set of read and write operations for SSTables. These operations inevitably introduce read and write amplifications.

Figure 4 illustrates read and write amplifications of LevelDB[3], Wisckey[11] and our Tidal-tree. Experimental results show that Tidal-tree can reduce read amplifications by 65% and 89% compared to Wisckey and LevelDB, respectively. Tidal-tree effectively reduces the number of SSTables that needs to be checked before the latest key-value is returned. In terms of write amplifications, Tidal-tree maintains write amplifications at a low level, which are about 2.09 times of Wisckey but 0.23 times of LevelDB. Since Tidal-tree adopts the same strategy of key-value separation as Wisckey, the write amplification compared to LevelDB is significantly reduced. Moving SSTables to upper levels will lead to extra compaction operations, which causes higher write amplifications compared to Wisckey.

3) Compaction: When a compaction operation is triggered, lots of computing and I/O resources are required for merging and persisting key-values. Table II presents the amount of read or write data involved in compaction operations. In Table II, the amount of data involved in compaction operations of Tidal-tree is 9.48 times of Wisckey but 0.36 times of LevelDB on average among read-intensive workloads. Among write-intensive workloads and read and write balanced workloads, Tidal-tree involves almost the same amount of data as Wisckey which is 0.88 times of Wisckey but 0.03 times of LevelDB.

TABLE II
THE AMOUNT OF DATA INVOLVED IN COMPACTION OPERATIONS FOR LEVELDB [3], WISCKEY [11], AND OUR TIDAL-TREE

Workloads	R:U:I	LevelDB (KB)	Wisckey (KB)	Tidal-tree (KB)
workload2	9:1:0	988,710	37,982	538,497
workload3	9:0:1	1,519,601	59,026	746,419
workload4	4:1:0	2,233,017	77,645	512,786
workload5	4:0:1	4,210,675	150,421	678,717
workload6	1:1:0	5,897,768	205,704	235,614
workload7	1:0:1	16,790,831	584,753	574,791
workload8	2:1:1	11,986,286	389,318	329,568
workload9	1:4:0	9,475,576	335,668	200,839
workload10	1:0:4	30,650,561	1,251,457	774,292
workload11	0:1:0	12,002,833	423,665	423,665
workload12	0:0:1	47,889,316	1,851,025	1,851,025

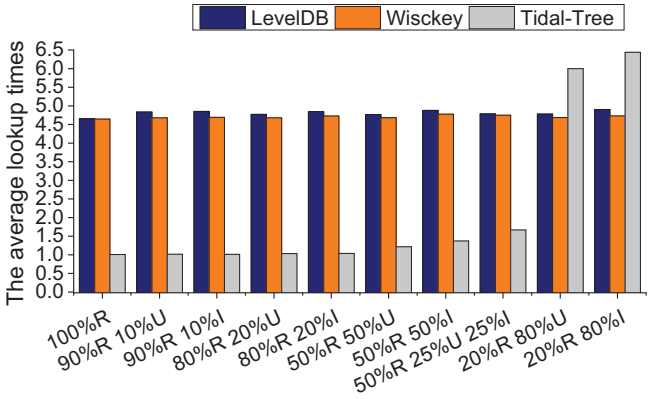


Fig. 5. The average number of times to check SSTables for LevelDB [3], Wisckey [11] and Tidal-tree.

Each level of LSM-tree has a limit on the number of files or the total amount of data. More SSTables are moved to upper levels in Tidal-tree among read-intensive workloads, which causes more compaction operations. To prevent frequently doing compaction operations, Tidal-tree dynamically adjusts the limit for the number of SSTables at each level to fit the characteristics of different workloads.

4) The number of times to check SSTables: To check an SSTable involves several I/O operations such as reading index blocks, reading data blocks and reading meta blocks. Reducing the number of times to check SSTables can effectively improve read efficiency. Figure 5 illustrates the average number of times to check SSTables. Tidal-tree checks about 2.2 times on average before the latest key-value is returned. Wisckey and LevelDB check about 4.7 and 4.8 times on average, respectively. The reduction is mainly due to the fact that, Wisckey and LevelDB move SSTables to bottom levels by compaction operations rather than moving SSTables to upper levels. Different from Wisckey and LevelDB, Tidal-tree moves SSTables to upper levels to reduce the extra checks, especially for those frequently accessed SSTables in bottom levels.

V. CONCLUSION

This paper presents a data structure called *Tidal-tree*. Tidal-tree modifies the existing LSM-tree structure considering the access frequency of each file. Tidal-tree targets at read-intensive workloads and aims to reduce the read amplification of key-value stores. Experimental results show that the proposed Tidal-tree can significantly reduce the total latency and read amplifications and effectively improve the read efficiency compared to the previous studies. In the future, we plan to investigate the use of our technique on other applications and propose a general structure that can be applied to workloads with different characteristics.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (61972259), in part by the National Technology Innovation Special Zone under (19-163-11-ZD-001-005-06), in part by the Guangdong Natural Science Foundation (2019B151502055 and 2017B030314073), in part by the Shenzhen Science and Technology Foundation (JCYJ20170817100300603).

REFERENCES

[1] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-Tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 4:1–4:26, 2008.
[3] *LevelDB, http://code.google.com/p/leveldb*.
[4] *RocksDB, http://rocksdb.org/*.
[5] *Cassandra, http://cassandra.apache.org/*.
[6] *HBase, http://hbase.apache.org/*.
[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 2, pp. 1277–1288, 2008.
[8] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, "Building efficient key-value stores via a lightweight compaction tree," *ACM Transactions on Storage (TOS)*, vol. 13, no. 4, pp. 29:1–29:28, 2017.
[9] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 497–514.
[10] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 477–489.
[11] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, 2016, pp. 133–148.
[12] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 1007–1019.
[13] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *2015 USENIX Annual Technical Conference (USENIX ATC)*, 2015, pp. 71–82.
[14] Y. Zhang, Y. Li, F. Guo, C. Li, and Y. Xu, "ElasticBF: Fine-grained and elastic bloom filter towards efficient read for LSM-tree-based KV stores," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.
[15] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang, "LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 68–79.
[16] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014, pp. 16:1–16:14.
[17] J. Zhang, Y. Lu, J. Shu, and X. Qin, "FlashKV: Accelerating KV performance with open-channel SSDs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5, pp. 139:1–139:19, 2017.
[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 143–154.