This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

Building an Efficient Put-Intensive Key-Value Store with Skip-tree

Yinliang Yue, Bingsheng He, Yuzhe Li, Weiping Wang

Abstract—Multi-component based Log-Structured Merge-tree (LSM-tree) has been becoming one of the mainstream indexes. LSM-tree adopts component-by-component KV item flowing down mechanism to push each KV item from one smaller component to the adjacent larger component during compaction procedures until the KV items reach the largest component. This process incurs significant write amplification and limits the write throughput. In this paper, we propose one multi-component Skip-tree to aggressively push the KV items to the non-adjacent larger components via skipping some components and then make the KV items' top-down move more efficient. We develop adaptive and reliable KV item movements among components. By reducing the number of steps during the flowing process from memory-resident component to the disk-resident largest component, Skip-tree can effectively reduce the write amplification and thus improve the system throughput. We design and implement one high performance key-value store, named SkipStore, based on Skip-tree. The experiments demonstrate that SkipStore outperforms the state-of-the-art open-sourced system RocksDB in Facebook by 66.5% under HDD and 61% under SSD.

Index Terms—Storage System, KV Store, LSM-tree, Compaction, Performance Optimization;

1 Introduction

In the recent years, latency-sensitive user interactive internet applications increase gradually. These interactive internet applications, such as WeChat, Twitter and Facebook, encourage users to produce and share short messages and small-sized pictures with simultaneously low read and write latency. Lots of efforts have been done to improve either read performance or write performance [1][4][6][23][27][37]. For example, B-tree supports fast read access but poor update performance[13]. On the contrary, log-structured file system provides high sequential write performance but unacceptable read access performance [33]. However, it is challenging and difficult to build one massive storage system to simultaneously support both high read and write performance. Multi-layer hierarchical storage architecture has been widely used to boost the read performance with speedy storage devices and alleviate the disk read and write I/O compete.

Although reads are also more than writes in most internet-based applications, many reads are absorbed by multi-level caches in the Internet architectures, and writes become more dominant for the accesses to the back-end storage systems[14][16][17][18][22]. Facebook [25] reveals that the popularity of photos is highly dependent on content age, and the multi-layer cache is extremely effective to intercept the reads. That is to say, 90.1% read requests are served by multi-layer cache, while only 9.9% read requests reach the back-end storage. On the other hand, more write-intensive applications that ingest event logs are becoming emerging, such as user clicks and mobile device sensor readings [1].

Compared with relational databases, KV stores pose simplified, easy-to-use interface and high scalability. KV stores are suitable for latency-sensitive internet services, and have been widely used in large-scale data intensive internet applications [1]. In order to simultaneously support increasing writes, low write latency and range-based scan, LSM-tree [3] and its variants including COLA [5], SAMT [6] and Hbase [8] have been widely used in the current emerging internet applications. LSM-tree is originally proposed to manage the data in disk-resident storage and the multi-layer disk-resident components are designed to balance the performance of read and write I/Os. Specifically, the newly written data are accessed with a great probability in its initial life-cycle, and so the LSM-tree employs *component-by-component* flowing mechanism. That is to say, each KV item is pushed from one smaller component to the adjacent larger component during compaction procedures until it reaches the largest component.

However, the LSM-tree is designed for embedded storage, in which memory footprints are commonly in small-size. Later, lots of researches have been done to boost the read or write performance for large-scale data management. For example, COLA [5] and FD-tree [7] use forward pointers to improve the read performance, while GTSSL [6] reinserts reads KV items into upper components to expedite lookup. In modern data centers, large-capacity memory is

- *Yinliang Yue is with Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. E-mail: yueyinliang@iie.ac.cn.*
- *Bingsheng He is with National University of Singapore, Singapore.*
- *Yuzhe Li and Weiping Wang are with Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

usually deployed and many more KV items can be held in memory for faster access. As most gets can be absorbed by large-capacity multi-layer caches, and only a few scattered gets fall in the LSM-tree, many data center applications are actually put-dominated. Thus it is unnecessary to discreetly push KV items top-down with *component-by-component* mechanism, which incurs excessive write amplification and then sacrifices the system throughput.

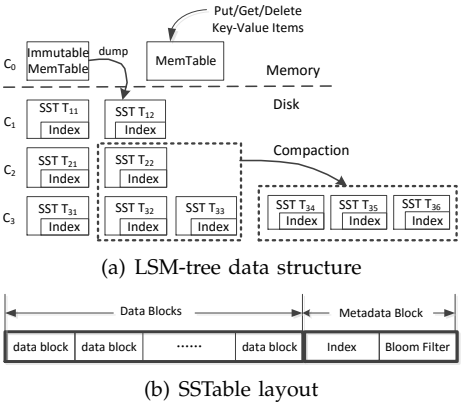
In this paper, we propose Skip-tree, to aggressively push the KV items to the non-adjacent larger components via skipping some components. By reducing the number of steps during the flowing process from memory-resident component to the disk-resident largest component, Skip-tree reduces the read and write I/Os, decreases the write amplification. Besides, the bloom filter [31] is used in the Skip-tree to further reduce the read I/Os caused by the version constraint. We develop adaptive and reliable KV item movements among components. As a consequence, Skip-tree improves the throughput of key value stores. We design and implement SkipStore based on Skip-tree. The experiments demonstrates that SkipStore outperforms RocksDB by 66.5%. Since SkipStore uses buffer to cache some KV items in newly added buffer component, we also design and implement reliability mechanism, which is based on write-ahead log, for SkipStore to prevent data loss. Benefitting from better put and scan performance, SkipStore can be used as the back-end storage engine of both cloud storage systems and other data analysis processing systems, such as PNUTS [12], Walnut [38] and Hadoop [39].

The rest of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 presents the overview of our solution. Section 4 describes the Skip-tree data structure, and Section 5 presents the design and implement issues of SkipStore based on Skip-tree. Section 6 presents and discusses the evaluation results. Section 7 presents the related work. Finally, we conclude this paper in Section 8 by summarizing the main contributions of this paper.

2 Background and Motivations

2.1 Multi-layer cached Data Center

Nowdays, most data centers are using multi-layer cache to reduce the average read latency and the read request counts to the backend system. We take Facebook's photo-serving stack [25] as an example to illustrate the architecture of multi-layer cached data center. There are three layers of caches in Facebook's photo-serving stack, which are browser cache, edge cache and origin cache. The first cache layer is in the client's browser. It caches the most read request, which is 65.5%. The Facebook Edge is comprised of a set of Edge Caches that each run inside points of presence (POPs) close to end users. As the second cache layer, edge cache caches 20% of read requests.



(a) LSM-tree data structure

(b) SSTable layout

Fig. 1. The basic LSM-tree data structure, SSTable layout and compaction procedure

The last cache layer, origin cache, is located with backend storage system. It caches 4.6% read requests and leaves 9.9% to the backend storage system, which is Haystack in Facebook.

Although there exist some temporary KV items in specific scenarios, it is unnecessary to persist them to the disk-based storage system. However, massive KV items in most scenarios are needed to be persistent to the disk-based storage systems. In this paper, we focus on solving the performance bottleneck of the LSM-tree based storage systems. Although KV items can be written to the memstore with extremely low latency, these should *eventually* be dumped into the disk-based persistent storage and involved in the compaction procedure to flow down to the larger components in most cases. Reducing write amplification and then improving write throughput of the back-end storage engine are challenging problems.

2.2 Basic KV organization of LSM-tree

LSM-tree organizes KV items in multiple tree-like components, generally including one memory resident component and multiple disk resident components, as shown in Figure 1(a). Each component size is limited to a predefined threshold, which grows exponentially. We use a representative design, RocksDB at Facebook, as an example to present the design and implementation of LSM-tree. RocksDB first uses an in-memory buffer, called MemTable, to receive the incoming KV items and keep them sorted. When an MemTable is filled up, it will be dumped to the hard disk to be an immutable SSTable, such as T_{12} in Figure 1(a). KV items are key-sorted and placed in fix-sized blocks. The key of the first KV item in each block is recorded as index to facilitate the KV item locating. Each disk component consists of multiple SSTables, whose key ranges do not overlap with each other except those in C_1. Figure 1(b) presents the layout of the SSTable. Each SSTable contains multiple data blocks and one metadata block. The data blocks contain the sorted KV items, while the metadata block

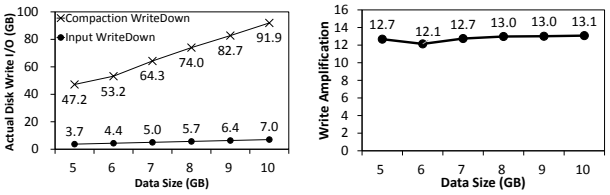
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

contains both indexes and bloom filters of each data block.

2.3 Compaction Procedure and Write Amplification in LSM-tree

LSM-tree defers and batches the updates on KV items, and cascades the changes from memory buffer component to multiple disk resident components. As shown in Figure 1(a), for one KV item, it is first inserted into memory buffer component C_0, and then dumped to C_1 and pushed to C_2, C_3, ..., C_k in sequence during the compaction procedure. A compaction entails reading sorted KV items in one or more SSTables from C_i, such as T_{22}, and that with the same key ranges from C_{i+1}, such as T_{32} and T_{33}, merging and sorting them, and then writing the sorted KV items back to C_{i+1} in the unit of fix-sized SSTables, such as T_{34}, T_{35} and T_{36}. For one specific KV item to reach C_k, it should be involved in the compaction between C_0 and C_1 to reach C_1 first, then involved in the compaction between C_1 and C_2 to reach C_2, and so on. Finally, it should be involved in the compaction between C_{k-1} and C_k to arrive at the destination component C_k. So it would be involved in at least k times of compaction procedures during its travel from C_0 to C_k. We call this KV item flow pattern as *component-by-component* flow.

Observation 1: LSM-tree has serious write amplification. The Amplification Factor (AF) is an important design parameter for LSM-tree. AF is defined to be the ratio of two adjacent components' size ($Size(C_{i+1})/Size(C_i)$, where i = 0,1,...). The ratio is 10 in RocksDB by default. We define the *Write Amplification Ratio (WAR)* as the proportion between actual write amount to the disk and the amount of data written by users. Because the key range covered by each component is roughly the same, the key range of one SSTable in component C_i may cover that of ten SSTables in component C_{i+1}. In order to push one SSTable at a component C_i to its next larger component C_{i+1}, in the worst case, eleven SSTables may be read from disk-resident component C_i and C_{i+1}, merged and sorted in memory, and then written back to C_{i+1}. We can say that the write amplification is 11, i.e., $WAR = AF + 1$. For a KV item flows from C_0 to C_k via component-by-component flow mechanism, the WAR can reach up to $K \times (AF + 1)$. We conduct experiments on RocksDB to measure the write amplification of LSM-tree based KV store using a heavy updated data set generated by Yahoo! YCSB with 1KB value size, random key, Zipf distribution and 100% update proportion. After loading 500MB data, we execute the run phase with data size of 5GB, 6GB, 7GB, 8GB, 9GB and 10GB respectively. From Figure 2(a) we can see that when the input data size is 5GB (in fact 3.7GB after compression), the actual disk write I/O is 47.2GB. In this case, the write amplification is 12.7. From Figure 2(b) we can see that WAR is increasing up to 13.1 with an average of 12.8. In addition, the write amplification becomes more severe with the increase of data size.



(a) data size and actual disk write I/O comparison (b) write amplification

Fig. 2. (a) Data size and actual disk write I/O comparison and (b) write amplification of RocksDB.

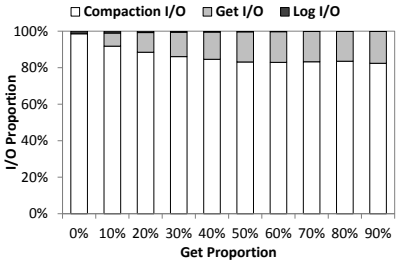


Fig. 3. Comparison of compaction, get and log I/Os.

Observation 2: compaction dominates the disk I/Os of LSM-tree. Without considering scan operation, the disk I/O of LSM-tree based KV store is mainly comprised of three parts, namely *log I/O*, *get I/O* and *compaction I/O*. *Log I/O* is the write-ahead log I/O for ensuring the reliability of KV items. *Get I/O* is issued to read data block from SSTable and get the specific KV items. *Compaction I/O* includes reading KV items from SSTables and writing the sorted and merged KV items to SSTables. We conduct the experiment on RocksDB to compare the proportion of the three types of I/Os using the data set generated by YCSB with 1KB value size, random key and Zipf distribution. In this experiment, both update and get operations are incorporated and we vary the get proportion from 0% to 90%. After loading 2GB data, we execute the run phase with 200,000,000 operations, i.e., 200GB data when the get operation proportion is 0. From Figure 3 we can see that the proportion of write-ahead log I/O is always less than 2% and the proportion of get I/O is increasing but never exceeds 20% even when the get proportion increases to 90%. The proportion of compaction I/O is consistently larger than 80%. We can conclude that serious write amplification greatly consume most of the disk I/O bandwidth, and leave little for servicing the frontend application requests.

Neither reducing the AF nor reducing the number of components simply can indeed decrease the WAR. For fixed size user data, reducing AF will enlarge the number of components, and vice versa. This motivates us to re-examine the KV items' component-by-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

component flow mechanism and explore whether it is possible to reach the destination component via as few compactions as possible.

2.4 Version Constraint in LSM-tree

LSM-tree is comprised of multiple ordered components. The deferral and batch features of LSM-tree means that out-of-place update is adopted and the actual update actions are postponed to compaction procedure. There may be multiple value versions of one specific key existing in the KV store. In practice, different value versions may spread among multiple components. Note that the KV items distribution in KV store must obey the value Version Constraint rule to ensure that the get operations can get the correct value versions.

For one get operation, memory-resident component C_0 is examined first followed by the disk-resident components C_1, C_2, ..., C_k in sequence. In order to get the newest value version, get operation should first encounter the newest value version. Thus, newer value version should resident in the smaller component, while older value version should resident in the larger component.

In the following, we have the version constraint rule. *For any two value versions K_mV_p and K_mV_q of one key K_m, if value version V_p is newer than V_q, the hosting component C_i of K_mV_p and the hosting component C_j of K_mV_q should meet the constraint $i \leq j$, which is called value Version Constraint rule of LSM-tree.* Obeying this rule can ensure that the applications can get the correct version for both get and scan operations.

During the current compaction procedure of LSM-tree, the value version constraint of all the KV items are guaranteed naturally. Specifically, when a number of SSTables in components C_i and C_{i+1} are involved in one compaction, if there exists multiple value versions of one key K_m in C_i and C_{i+1}, they would be sorted and merged during the compaction procedure, and all the compaction output KV items would be pushed to C_{i+1}. Even if there exist multiple value versions of key K_m in the adjacent smaller component C_{i-1} or larger component C_{i+2}, multiple value versions of key K_m in these multiple components will not break the version constraint, because the input and output KV items of one compaction procedure are limited to C_i and C_{i+1}, and none of the output KV items tries to jump into adjacent components.

3 Solution Overview

To avoid the component-by-component compaction, we can aggressively put the compaction output KV items to the non-adjacent larger component by skipping some components. By this way, few compactions are needed before KV items' reaching the destination component C_k and write amplification can be greatly reduced. However, there exists some challenges to aggressively push compaction output KV items to the non-adjacent much larger components.

First, *which and how many components should be skipped by one specific KV item*. LSM-tree is one multiple-component data structure. On the premise of satisfying value version constraint, the compaction output KV item K_mV_p may skip across one or more components. The number of components skipped across by one KV item have significant impact on the write amplification and put latency. The KV item's flowing down may produce the following benefit. That is to say, the more components are skipped across, the smaller the write amplification can achieve. However, we must make sure that the component skipped by the KV item does not contain any KV item with the same key. One intuitive method is to read the SSTables of the component, and check out whether they contain the KV item or not. However, this simple approach will bring extra I/O cost. The more components are skipped, the more SSTables will be read, and the more I/O cost will be incur. We need a better approach to determine how many components should be skipped can achieve the best overall throughput.

Second, *how to efficiently locate the KV item in the destination components*. In LSM-tree, all the KV items in one component is located in disk-resident SSTables in key order. The output KV items of compaction between C_i and C_{i+1} are located in C_{i+1} by default. If aggressive KV item's flowing down mechanism is adopted, the skipping down KV item should be merged and sorted with the corresponding disk-resident SSTable in the larger component. One intuitive method is to read the host disk-resident SSTable into memory, inserting new KV item in order, and then writing them back to disk as an new SSTable. However, this method would incur huge I/O cost. Thus how to merge one KV item with the larger components is one challenge.

By addressing the above two challenges, we propose *Skip-tree*, an enhanced version of LSM-tree. Skip-tree allows KV items to efficiently skip the components as many as possible during compaction procedure while preserving the version constraint. Besides, new buffer components are added in Skip-tree to cache the KV items that can solve the second challenge. Then we implement an efficient KV store, which is called SkipStore, based on Skip-tree. In the following sections, we will present the details of our design and implementation.

4 Skip-tree Data Structure

Different from LSM-tree [3] and its variants [1][2][4][8][23][27] that are designed to balance the read and write performance and mainly used in embedded storage, Skip-tree is designed to be the local back-end storage in data centers with

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

intensive put operations. Skip-tree aims to high write performance with acceptable read and scan performance. Similar with the LSM-tree, Skip-tree is composed of one memory-resident component and multiple disk-resident components. The KV items are injected into memory component first, dumped to disk, and then compacted for merge and sort. There are two design features of Skip-tree for put-intensive workloads. First, Skip-tree uses KV item skipping mechanism to make KV items skip across one or more components during compaction procedure. Second, each component of Skip-tree has a buffer to locate KV items that skips from upper component.

4.1 KV Item Skipping Mechanism

For each KV item, it is important to judge whether one component should be skipped or not. Recall that one KV item's aggressive skipping must obey the value version constraint rule. Since SSTables are sorted by key in one component, one specific KV item can only exist in one SSTable. Note that the index in metadata block in one SSTable does not tell whether an item is actually in the data block or not. It would incur huge I/O cost by reading the data block, picking out of each key and comparing them with the specific key.

To address the above problem, we check whether there exists one value version with the specific key in one data block with the help of bloom filter. We maintain a bloom filter for each block, and the key of every KV item should be added into the block's bloom filter before it is stored in the block. Thus we can simply use the bloom filter to indicate whether one KV item exists in the block or not. In practice, the number of hash functions is set as 3 and we set the bit space of bloom filter based on the number of KV items in one block, which is 8 bits per KV item.

Since the bit space of bloom filter is limited, there exists false positive rate in bloom filter. Although we can firmly believe that one key does not exist in one data block, we can not guarantee that one key exists in one data block even if all the bits corresponding to all the hash locations are 1. That is, the data block *may*, despite that it might not in fact, contain one value version of the given key. The false positive feature of bloom filter prevent us from pushing KV item K_mV_p to the larger components, such as C_{i+2},C_{i+3} and so on. In this paper, we conservatively regard that data block B_j of one SSTable in C_{i+2} contains at least one value version with the same key K_m, although data block B_j may not contain any value version of the key K_m. Our policy can avoid the huge I/O cost when to check the existence of value version of key K_m by reading the disk-resident data block B_j. Note that although the conservative judgement might be an obstacle to the possible further flow towards larger components, version constraint rule can be consistently maintained.

KV items' skipping across one or more components must satisfy the value version constraint rule. As discussed in Section 3, the number of components skipped across by one KV item have significant impact on the write amplification, get latency and put latency. Although skipping across only one component can minimize the increase of get latency and put latency, the write amplification reduction will be limited. On the contrary, skipping across as many components as possible can dramatically reduce the write amplification, however, it incurs significant put and get latency. For each compaction output KV item, we limit the max number of components being skipped. Note that the actual number of components being skipped across is less than or equal to the predefined max number of components. Although we limit the max number of component being skipped, the actual number of components being skipped across is decided by the distribution of value versions of one key, and constrained by the value version constraint. Based on the extensive evaluation result we can see that setting max number of components being skipped across as two can achieve the best total system throughput.

TABLE 1
Symbols definition

Symbols	*Definition*
K_s, K_e	start key and end key of a compaction
BKV, BK	KV items in buffer components; the key of BKV
S-SST,	the source SSTable of one BKV
S-BUF	source buffer component of one BKV
D-SST	the destination SSTable of one BKV
D-BUF	the destination buffer component of one BKV
PSST	Parent SSTable, i.e. the S-SST of in-buffer KV items
PSST–>CC	Count of in-buffer Children KV items of PSST
$\langle BK, PSST\rangle$	the mapping relationship between BK and PSST
BK-PSST Map	the map of $\langle BK, PSST\rangle$
BPM_i	the BK-PSST Map for in-buffer KV items in B_i

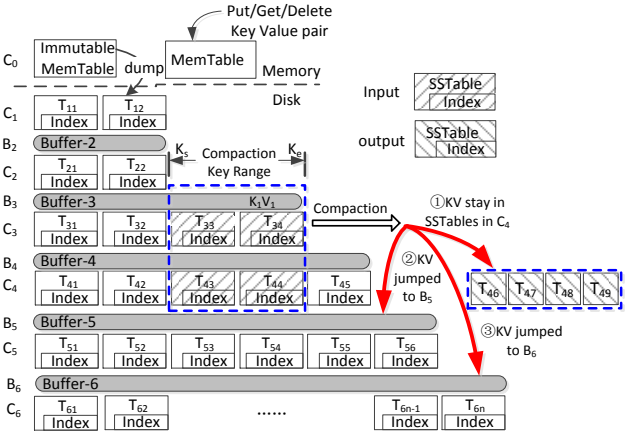


Fig. 4. The architecture of Skip-tree.

4.2 Buffer-based Delayed Merge and Sort

When one compaction output KV item K_mV_p needs to skip to the destination SSTable in component C_{i+2} or larger components C_{i+3}, C_{i+4}, ..., C_k, the most intuitive approach is to merge the KV item with the destination SSTable immediately. However, in this

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

way, all the KV items in destination SSTable should be read from disk and written back to disk after being merged and sorted with the KV item, which obviously incurring lots of extra disk I/Os, resulting in serious write amplification and finally sacrificing the throughput of LSM-tree based KV stores.

To avoid the massive I/O on reading SSTables, we propose buffer-based delayed merge and sort mechanism. As shown in Figure 4, we add a memory-resident buffer component B_i for each disk-resident component C_i to buffer the skipping down KV items whose target component is C_i. In this way, the merge and sort between the skipping down KV item and destination SSTable in target component are delayed and postponed to the consequent compactions. The detailed compaction of Skip-tree is described in Section 5.1.

The organization of the KV items in memory-resident buffer components B_i is different from that in disk-resident components C_i. As presented in Section 2.2, KV items in disk-resident components C_i are grouped into a number of fixed size SSTables. However, all KV items in memory-resident buffer component B_i are organized as a simple link list sorted by key.

5 SkipStore Design and Implementation

We build SkipStore based on Skip-tree. SkipStore supports put, get, delete and scan interfaces. Since SkipStore caches some KV items in buffer component, these KV items may be lost in some scenarios, such as sudden power or system failure. Since reliability is an important design issue of storage systems [9][10] [22], we propose an efficient reliability mechanism based on write-ahead log to prevent data loss.

5.1 Compaction Procedure

In the compaction of LSM-tree, a number of SSTables in two adjacent components C_i and C_{i+1} are selected as inputs. After being merged and sorted, all the KV items are grouped as fixed-size SSTables and written back to disks as outputs.

In Skip-tree, the KV items in the corresponding buffer components B_i and B_{i+1} should also be involved in the compaction procedure. Specifically, a number of SSTables are picked out as the inputs of one compaction first, and then determine the start key K_s and end key K_e of these SSTables. All the KV items falling in the key range $[K_s, K_e]$ of buffer component B_i and B_{i+1} are involved in this compaction.

Figure 4 shows a compaction procedure of Skip-tree whose input includes SSTables T_{33} and T_{34} in component C_3 and SSTables T_{43} and T_{44} in component C_4 as well as the BKVs (Buffered KV items, see Table 1 for symbols definition) in buffer component B_3 and B_4 within the key range $[K_s, K_e]$. Each compaction output KV item is pushed to as large components as possible according to the above mentioned aggressive skipping mechanism, meanwhile ensure the version constraint via bloom filter based key existence judgement.

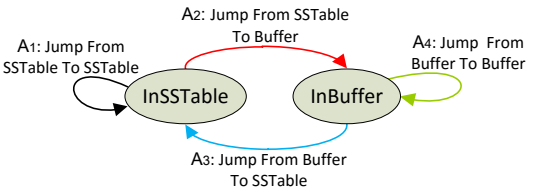


Fig. 5. State transition of KV items between InSSTable and InBuffer during the compaction procedure.

In Skip-tree, both MemTable and buffer components are memory-resident. KV items in MemTable are inserted by users. Write-ahead log is adopted in LSM-tree implementations, such as LevelDB at Google and RocksDB at Facebook, to guarantee the reliability of both keys and values in MemTable. When MemTable is filled up, all the KV items in MemTable are dumped in component C_1 together. There are some differences between MemTable and buffer components. First, all the KV items in buffer components come from disk-resident SSTables and can be read from their Parent SSTable (PSST for short and defined in Table 1) with the corresponding key. Second, different from one-way dump from memory to disk for KV items in MemTable, each KV item takes one of the four actions A_1, A_2, A_3 and A_4 during compaction procedure to move from disk components to buffer components, from one buffer component to another buffer component, or from buffer components to disk components, as shown in Figure 5. Correspondingly, we say that the KV items is in either *InSSTable* state or *InBuffer* state, and convert between them during compaction procedure. Note that *InSSTable* is the initial state for each KV item, because it locates in the disk-resident SSTable before moving to buffer components.

5.2 Reliability Mechanisms

5.2.1 Write-ahead Log

All the in-buffer KV items move from disk components into buffer components during the compaction procedures. When the storage servers suffer from sudden power off, we can recover the in-buffer KV items by getting them from the corresponding SSTables. In general, storage space is sufficient and we delay the deletion of stale SSTables. Thus the only thing need to do for reliability guarantee is to record the KV items' state transitions as write-ahead log during the compaction procedure.

When action A_1 is executed, it is not necessary to record the KV item's state transition. The reason is that KV items move from one disk-resident SSTable to another disk-resident SSTable. We should record the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

KV item's state transition as write-ahead log when action A_2, A_3 and A_4 are executed. For actions A_2 and A_4, i.e., moving from disk components to buffer components, and moving from one buffer component to another buffer component, we record the action type of the KV items, such as *S2B* or *B2B*, the source of the KV items, such as S-SST, S-BUF, and the destination of the KV items, such as D-SST, D-BUF. For action A_3, i.e., moving from buffer components to disk components, it means that one compaction is scheduled. Some in-buffer KV items falling in $[K_s, K_e]$ are involved as inputs, and move to disk components as outputs. We record the input components and both the start key and end key of the compaction. Since write-ahead log records all the KV item's state transition in time order during compaction procedures, we can recover all in-buffer KV items and put them in the right buffer components by replaying the write-ahead log.

5.2.2 Memory Data Structures

For each in-buffer KV item, we record the following two points, i.e., jumping from which SSTable, and then locating in which buffer component. For the first point, we utilize an map table to record the PSST of each in-buffer KV item, as shown in Figure 7. When one KV item jumps from disk components to buffer components, we create one new mapping entry, including the in-buffer key *BK* and its parent SSTable *PSST*, and insert the mapping entry <*BK, PSST*> into BK-PSST Map. For the second point, we set one mapping table BPM_i for each buffer component B_i to facilitate the management of in-buffer KV items.

Note that multiple in-buffer KV items may come from the same PSST. These KV items may jump into buffer components, or jump back to disk components during the different compaction procedures. Note that it is necessary to record the count of in-buffer children KV items of each PSST. For one SSTable, we call each of its KV item as children item, and define the Count of in-buffer Children KV items as PSST−>CC, as shown in Table 1. Along with the KV items' jumping between buffer components and disk components, BK-PSST Map would be changed, followed by the increase or decrease of PSST−>CC, i.e., PSST−>CC + 1 or PSST−>CC - 1. Note that one children KV item's jumping from disk components to buffer components result in PSST−>CC + 1, meanwhile one in-buffer children KV item's jumping from buffer components to disk components result in PSST−>CC - 1. For one SSTable whose PSST−>CC = 0, we say that this SSTable is no more the PSST(short for Parent SSTable) of any in-buffer KV item, and it can be deleted from disks to release the disk space in time to make room for newly written data. The SSTables whose PSST−>CC $\neq$ 0 must stay in disk in order to recover its in-buffer children KV items.

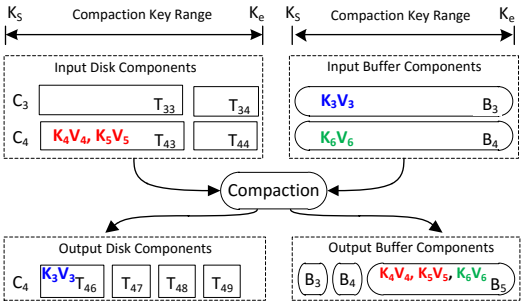


Fig. 6. The KV items jump between SSTable and buffer during the compaction procedure. K_4V_4 and K_5V_5 jump from SSTable to buffer, K_6V_6 jumps from buffer to buffer and K_3V_3 jumps from buffer to SSTable.

5.2.3 A Case Study for Reliability

In this section, we conduct one case study to show the BK-PSST Map's change during the compaction procedure.

Figure 6 shows a compaction procedure of Skip-tree whose input includes SSTables T_{33} and T_{34} in component C_3 and SSTables T_{43} and T_{44} in component C_4, as well as the in-buffer KV items in B_3 and B_4 within the key range $[K_s, K_e]$. The outputs are SSTables T_{46}, T_{47}, T_{48} and T_{49} in C_4 as well as the KV items in B_5. We mark some specific KV items such as K_4V_4 and K_5V_5 jumping from T_{43} to B_5 by taking action A_2, K_6V_6 jumping from B_4 to B_5 by taking action A_4, K_3V_3 jumping from B_3 back to T_{46} by taking action A_3. It is worth noting that we do not focus on action A_1 that jumping from SSTable to SSTable since it has nothing to do with buffer. We assume $K_1<K_2<K_s<K_3<K_4<K_5<K_6<K_e<K_7$. Note that KV items with key K_1, K_2 and K_7 do not participate in the compaction procedure, thus they are not shown in Figure 6. KV items with key K_3, K_4, K_5 and K_6 participate in the compaction, during which KV items jump into or out from buffer components and result in the change of both BK-PSST Map and PSST−>CC, as shown in Figure 7.

Figure 7(a) shows the initial status of BK-PSST Map and count of in-buffer children KV items of each PSST, i.e., PSST−>CC, before compaction.

Before compaction, both K_4V_4 and K_5V_5 stay in SSTable T_{43}. During compaction, they jump into buffer component B_5 by taking A_2 in accordance with the order of keys. Thus the following two records <*S2B*, K_4, T_{43}, B_5>, <*S2B*, K_5, T_{43}, B_5> would be appended to the write-ahead log, as the Line 1 and Line 2 shown in Figure 8. These two records represent that both K_4V_4 and K_5V_5 jump from SSTable T_{43} into buffer component B_5. Besides, <K_4, T_{43}> and <K_5, T_{43}> are inserted into BPM_5 in sequence, as shown in Figure 7(b). We set the initial value of PSST−>CC as zero, i.e., T_{43}−>CC = 0. Both the jumping of K_4V_4 and K_5V_5 trigger the operation T_{43}−>CC + 1, so T_{43}−>CC is increased twice after the compaction, i.e., T_{43}−>CC

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

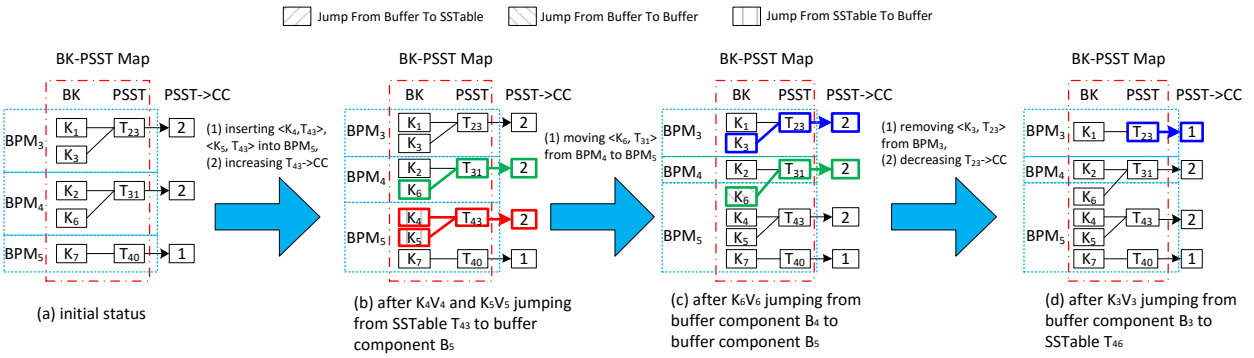


Fig. 7. Change of both BK-PSST Map and count of in-buffer KV items of each PSST during compaction procedure.

	Log Type	In-buffer key	S-SST/S-BUF	D-SST/D-BUF
Line 1	S2B	K_4	T_{43}	B_5
Line 2	S2B	K_5	T_{43}	B_5
Line 3	B2B	K_6	B_4	B_5

	Log Type	Input Buffer Components	Start Key	End Key
Line 4	B2S	B_3, B_4	K_s	K_e

Fig. 8. Write-ahead log generated during the compaction procedure. S2B means KV items jump from SSTable to buffer, B2B means KV items jump from buffer to buffer, B2S means KV items jump from buffer to SSTable.

= 2, as shown in Figure 7(b).

During the compaction procedure, K_6V_6 jumps from B_4 to B_5 by taking A_4, the record <*B2B*, K_6, B_4, B_5> would be appended to the write-ahead log, as the Line 3 shown in Figure 8. Besides, <K_6, T_{31}> moves from BPM_4 to BPM_5, as shown in Figure 7(c). Since K_6V_6 is still staying in buffer components and none of the KV items of T_{31} jumps into or out from buffer components, the count of T_{31}'s in-buffer children KV items T_{31}->CC is no need to change and keep as 2.

K_3V_3 jumps from B_3 back to disk-resident T_{46} by taking A_3 during the compaction procedure. Note that we take K_3 for example to represent all in-buffer KV items in $[K_s, K_e]$ jumping back to disks from buffer components. The corresponding record appended to write-ahead log is <*B2S*, B_3, B_4, K_s, K_e>, representing the compaction among B_3, B_4 and their corresponding disk-resident components C_3 and C_4 with the key range of $[K_s, K_e]$, as the Line 4 shown in Figure 8. Besides, <K_3, T_{23}> will be removed from BPM_3. Since T_{23} is the parent SSTable of K_3V_3, the K_3V_3's jumping out from buffer components triggers the operation T_{23}->CC - 1, so T_{23}->CC is decreased by one, and changed from 2 to 1, as shown in Figure 7(d).

5.3 In-Buffer KV Items Checkpoint and Recovery Mechanism

When the system suffers sudden power off, all the in-buffer KV items in buffer components are lost. In order to recover the in-buffer KV items, we first need to recover the BK-PSST map via sequentially replay the write-ahead log, and then retrieving the KV items from corresponding disk-resident SSTables. Note that the size of write-ahead log keeps ever-increasing along with the system run, during which KV items jumping between disk-resident components and buffer components. The ever-increasing write-ahead log incurs the following two issues. First, the larger the write-ahead log increases, the more time the recover process needs. Second, disk space is wasted for keeping lots of PSSTs. In this paper, we periodically make checkpoints to flush all the in-buffer KV items onto disks for persistence, in order to shorten the recovery time, and release the disk space in time.

During the checkpoint process, all the in-buffer KV items in buffer components are dumped onto disks first, and then we delete the write-ahead log. Besides, it is not necessary to retain the in-buffer KV items' parent SSTables in the hard disks. The reason is that all the in-buffer KV items can be recovered from the checkpoint file. So all the SSTables who are the parent SSTables of in-buffer KV items will be deleted.

During the recovery process, we first reload all the KV items in checkpoint file into memory, and then relay the write-ahead log to add some KV items into buffer components, or delete some KV items from buffer components. Specifically, for Line 1 and Line 2 in Figure 8, the log type *S2B* means that one KV item takes action A_2 and jumps from one disk-resident SSTable into buffer components during the compaction procedures. When to replay the Line 1 in Figure 8, we get one KV item with key K_4 from the S-SST T_{43} and insert it into the D-BUF B_5. Similarly, we get one KV item with key K_5 from the S-SST T_{43} and insert it into the D-BUF B_5 when to replay the Line 2 in Figure 8. For Line 3 in Figure 8, the log type *B2B* means that one KV item takes action A_4 and jumps from one buffer component to another buffer

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

components during the compaction procedures. When to replay the Line 3 in Figure 8, we move one KV item with key K_6 from S-BUF B_4 to D-BUF B_5. For Line 4 in Figure 8, the log type *B2S* means that a group of KV items in one specified key range take action A_3 and jump from buffer components to disk-resident SSTables. When to replay the Line 4 in Figure 8, we remove all the KV items locating in the key range between K_s and K_e from two input buffer components B_3 and B_4.

5.4 Discussion

5.4.1 Get Performance

While Skip-tree is designed for put-intensive workloads, the gets performance of Skip-tree is not significantly worse than that of RocksDB.

First, in practical storage systems, LSM-tree based KV stores are usually used as the back-end persistent engine coupled with one or multiple levels of caches. For example, Taobao tair[15], a distributed key-value storage system developed and widely used in China's largest e-commerce website Taobao.com to support a bulk of e-commerce business related workloads, adopts one in-memory KV store as KV items cache and LSM-tree based KV store as backend persistent engine. Besides, the analysis results of photo caching mechanism in Facebook[25] show that, 90.1% of get requests are served by multiple-layer cache and only 9.9% of get requests served by back-end storage. In Skip-tree, KV items cache is checked first, followed by each component with order C_0, B_1, C_1, B_2, C_2, ..., B_n, C_n. For KV items reads and writes with zipfian key distribution, most gets are serviced by the KV item cache. For that with uniform key distribution, since writes are closely followed by the corresponding reads with the same key, lots of gets can also be serviced by the KV item cache.

Second, gets in RocksDB are seriously disturbed by compaction I/Os, which dominate the disk I/Os, as shown in Figure 3. Most of the time, gets are hanged up and wait for the time-consuming compaction procedure. In contrast, Skip-tree dramatically decrease write amplification and reduced compaction reads and writes alleviates the disk I/Os contention among compaction I/Os and get I/Os.

5.4.2 Write-ahead Log Cost

The KV items state transitions are recorded as write-ahead log during the compaction procedure. The results of the experimental evaluation show that the write-ahead log I/O for KV items' state transition is less than 10KB/s, and its influence on the compaction I/O activities is negligible.

5.4.3 Checkpoint Cost

Checkpoint may disturb the get and compaction I/O operations, thereby sacrificing the performance of the system. Frequent checkpoint can effectively reduce the recovery time and improve the system availability, however, it will introduce more extra checkpoint I/Os. Decreasing the frequency of checkpoint incurs less extra checkpoint I/Os and improves the system performance, with the cost of extended recovery time. The quantitative impact incurred by write-ahead log cost and checkpoint cost will be evaluated in section 6.

5.4.4 Buffer Management

In SkipStore, we add one buffer component for each disk-resident component. Along with the increase of total data volume, the number of disk-resident components and buffer components increases accordingly. Although the total size of buffer components is capped, the actual buffer footprint may slightly surpass the theoretical boundaries. During the compaction procedure, some key/value items jumped into the buffer components, meanwhile some key/value items jump out of the buffer components. In order to avoid the excessive increase of the memory footprint, we reclaim the free memory periodically and allocate them to the key/value items newly jumped into the memory.

5.4.5 Applicability of SkipStore

With better put and scan performance, SkipStore can be used in the two mainstream scenarios. First, SkipStore is suitable for put-intensive workloads, and thus can be used as the back-end storage engine of cloud storage systems such as PNUTS [12] and Walnut [38]. Second, SkipStore has better scan performance, and thus can also be used as the back-end storage engine of data analytical processing systems such as Hadoop [39].

6 Evaluations

In this section, we conduct extensive experiments to compare the throughput and failure recovery time of SkipStore with that of RocksDB[10]. RocksDB is a representative LSM-tree implementation distributed by Facebook and the version of RocksDB that we use is 4.6.1. We examine the impacts of request distribution, buffer size and storage medium.

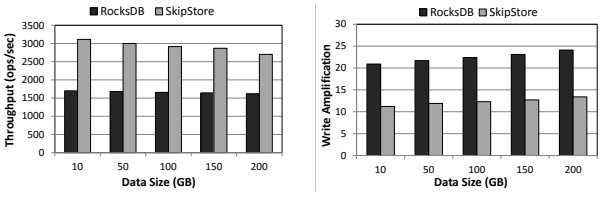
6.1 Evaluation Methodology

We use synthetic data generated by YCSB as the data sets. YCSB is widely used to evaluate the performance of cloud serving systems with multiple workloads. We use the default configurations of random key and value size of 1KB, and set the request distribution to Zipf. The parameters of RocksDB and SkipStore are set as default values. In addition, we set the buffer size of SkipStore (see Section 4.2) as 256MB. We evaluate both RocksDB and SkipStore on Linux servers equipped as TABLE 2.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

TABLE 2
System configuration parameters

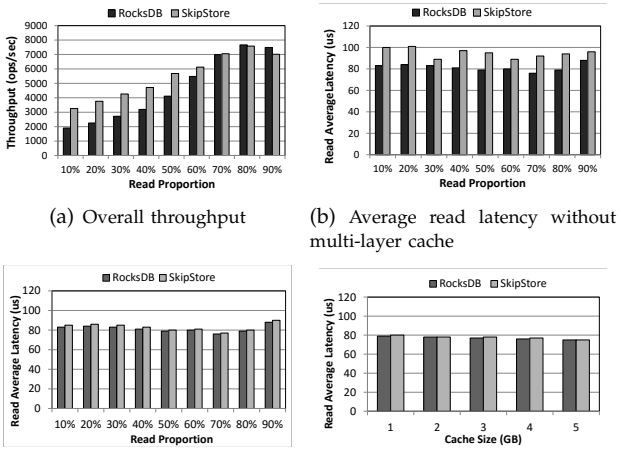
Hardware Configuration	*Value*
CPU	Intel Xeon E5-2630 v3@2.40GHz
Available Memory Size	2GB
Software Configuration	***Value***
OS Distribution	Red Hat Enterprise Linux Serv. 6.5
Kernel	2.6.32
HDD Parameter	***Value***
HDD Model	TOSHIBA AL13SEB600
Capacity/Rotational Speed	600GB/15000 RPM
Avg. Seek/Rotational Time	2.7/4.1ms
Sustained Transfer Rate	286 MB/s
SSD Parameter	***Value***
SSD Model	Intel SSD DC S3700 Series
Capacity	400GB
Read/Write Transfer Rate	500 MB/s 460 MB/s



(a) Overall Throughput (b) Write Amplification

(c) Update Latency

Fig. 9. Comparison of throughput, write amplification and update latency between SkipStore and RocksDB with 100% update workloads under different data size



(a) Overall throughput (b) Average read latency without multi-layer cache

(c) Average read latency with total 1GB multi-layer cache (d) Average read latency with total 1GB $\sim$ 5GB multi-layer cache

Fig. 10. Comparison of both overall throughput and read latency between SkipStore and RocksDB with hybrid put/get workloads.

6.2 Main Evaluation

6.2.1 Overall Throughput

In order to evaluate the overall put throughput of SkipStore compared with RocksDB, we set the YCSB workload to 100% update operations. Then we set the run phase data size of YCSB workload to 10GB and repeat 20 times to reach the total data size of 200GB. We show the throughput of RocksDB and SkipStore in Figure 9(a) with the data size grows from 10GB to 50GB, 100GB, 150GB and 200GB. We can see that the throughput of SkipStore outperforms that of RocksDB at least 66.5% with the data size increasing. The reason is that the write amplification is greatly reduced in SkipStore. As shown in Figure 9(b), the write amplification of SkipStore is only 54% $\sim$ 56% of that of RocksDB. The decrease of write amplification can greatly reduce the update latency and further improve the write throughput of SkipStore. As shown in Figure 9(c), the update latency of SkipStore is only 56% $\sim$ 62% of that of RocksDB.

Then we conduct experiments on YCSB data set to evaluate the overall throughput of SkipStore compared with RocksDB under mixed put/get workloads. In YCSB data set, we set the data size as 50GB. We use both update and read operations and vary the read proportion from 10% to 90%. As shown in Figure 10(a), with the increase of read operation's proportion, the performance improvement of SkipStore compared with RocksDB decreases gradually, from 66.5% to -6%. Figure 10(b) shows that the read latency of SkipStore without multi-layer cache is about 7%$\sim$ 21% higher than that of RocksDB, the reason is that the cache hit ratio of RocksDB and SkipStore is 89.7% and 83.6% respectively. We deploy multi-layer cache size as 1GB for both RocksDB and SkipStore, and one can see that the average read latency gap between RocksDB and SkipStore is less than 2%, as shown in Figure 10(c). We then keep read proportion as 50% and vary the multi-layer cache size from 1GB to 5GB. As shown in Figure 10(d), one can see that the average read latency gap between RocksDB and SkipStore is less than 1%.

In summary, SkipStore outperforms RocksDB by at least 66.5% under YCSB dataset, which shows that SkipStore performs very well for put-intensive workloads. Meanwhile, as we expected, SkipStore's read performance is not as good as RocksDB. However, this defect can be easily solved by using multi-layer cache, which has been widely used in practical applications, such as Facebook's photo-serving stack [25].

6.2.2 Details of Skipping levels

In order to illustrate the details of skipping levels in SkipStore, we show the number of KV items that skips n(n = 0, 1, 2, 3, 4, 5, 6) levels during compaction procedure in SkipStore and RocksDB in Table 3. We can see that KV items in RocksDB can only skip 0 or 1 level during compaction procedure, which causes the large write amplification (see Section 2.3). However, in SkipStore, the KV items can skip at most 6 level during compaction procedure. From Table 3 we can see that the average skipping levels of RocksDB and SkipStore are 0.09 and 0.23 respectively. The kv item's

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

TABLE 3
Comparison of KV items' skipping between SkipStore and RocksDB (Skipping speed factor means KV item's skipping speed compared with that of RocksDB)

	Number of KV items that skipping 0 ∼ 6 levels							Avg. skipping levels	Skipping speed factor
	0	1	2	3	4	5	6		
RocksDB	514947985	52349414	0	0	0	0	0	0.09	1
SkipStore	298505734	55748476	8062923	2808652	685316	3217	341	0.23	2.56

skipping speed of SkipStore is 2.56 times faster than that of RocksDB. As a result, the write amplification of SkipStore is far less than that of RocksDB, as shown in Figure 9(b).

6.2.3 Scan Performance

We conduct experiments on YCSB data set to evaluate the Scan performance of SkipStore compared with RocksDB. We use the workload including both update and scan operations and vary the scan proportion from 80% to 100%. Compared with RocksDB, the scan performance improvement ratio of SkipStore is up to 85.9%∼ 180.1%. The reason is that a substantial part of key/value items are stored in buffer components, which can greatly reduce the scan time.

6.2.4 Recovery time

During the recovery, RocksDB needs to restore the key/value items stored in MemTable. SkipStore needs to restore the key/value items stored in buffer components besides the MemTable. We conduct experiments to measure the throughput and recovery time of SkipStore and RocksDB. Experimental results reveal that the recovery time of RocksDB and SkipStore is 3.25 second and 4.66 second respectively. The recovery time of SkipStore is longer than that of RocksDB since SkipStore needs extra step to recover the key value items in Buffer Components. Note that the recovery time of SkipStore is only 1.41s longer than that of RocksDB, the reason is that the key/value items are periodically checkpointed, and the key/value items need to be recovered are on average the half of the total buffer size.

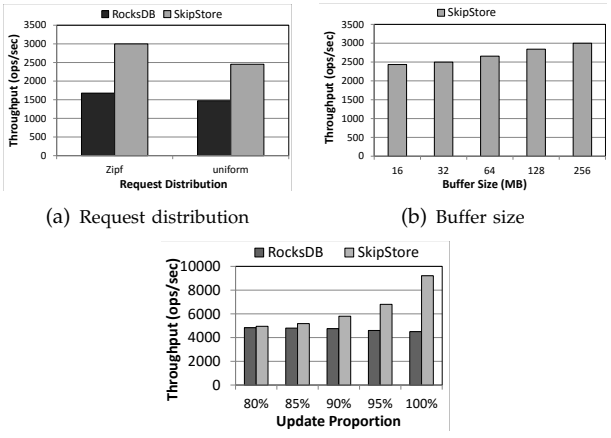
6.3 Sensitivity Study

In all experiments of sensitivity studies, we set data size as 200GB and update operation ratio as 100%. We set buffer size of SkipStore as 256MB in Section 6.3.1 and the request distribution as zipf in Section 6.3.2.

6.3.1 Request Distribution (Zipf vs Uniform)

To examine the impacts of request distribution, we conduct experiments with request distribution of Zipf and uniform respectively. As shown in Figure 11(a), the throughput of SkipStore and RocksDB under uniform distribution is lower than that under Zipf distribution.

The reason is that there is many duplicate KV items under Zipf distribution, and only the KV item of the newest version can be retained if the KV items with same key meet in the compaction procedure (except the KV items in different snapshot, which has been discussed in Section 2.4), so the output data size of one compaction is usually smaller than the input data size of this compaction under Zipf distribution, which means the write amplification would be smaller. However, there is no duplicate KV item under uniform distribution, so the output data size of one compaction is as same as the input data size of this compaction under uniform distribution, and the write amplification would be larger. As a result, the throughput under Zipf distribution is higher than that under uniform distribution.



(a) Request distribution (b) Buffer size

(c) under SSD

Fig. 11. Throughput of RocksDB and SkipStore (a) with different request distribution (b) with different buffer size (c) under SSD

6.3.2 Buffer Size

To examine the impacts of buffer size, we conduct experiments with buffer size of 16MB, 32MB, 64MB, 128MB and 256MB respectively. As shown in Figure 11(b), the throughput of SkipStore gradually increases with the increase of buffer size. The reason is that with the increase of buffer size, more key/value items can be stored in buffer and further decreases the write amplification.

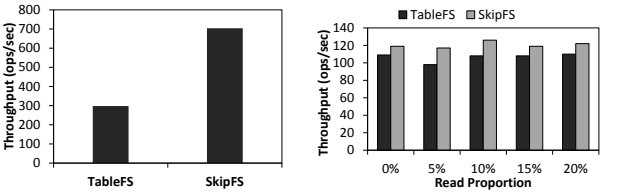
6.3.3 Storage Medium (HDD, SSD)

To examine the impacts of different storage medium, we conduct experiments with storage medium of HDD and SSD. We set the block size and buffer size as 8KB and 256MB respectively, and use random key as well as Zipf distribution. Besides, we vary the update proportion from 80% and 100%. From Figure 11(c) one can see that the performance improvement

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

TABLE 4
Throughput and recovery time with different checkpoint interval

Checkpoint Interval (min)	Throughput (ops/s)	Recovery Time (s)
No WAL / No Checkpoint	5699	218
20	5682	4.66
15	5659	4.27
10	5642	4.12
5	5631	3.98
2	5225	3.85
1	4908	3.67



(a) Create transaction (b) Read-Append transaction

Fig. 12. Throughput of TableFS and SkipFS for (a) create transaction and (b) read-append transaction.

ratio of SkipStore compared with RocksDB increases from 5% to 61%. Note that the absolute performance improvement ratio under SSD is slightly smaller than that under HDD. The main reason is that SkipStore is designed to improve throughput by reducing write amplification. Since bandwidth of hard-disk is far less than that of Solid State Disks, the impact of write amplification under SSD is not as serious as that under HDD. Thus the proposed Skip-tree mechanism brings smaller performance boost under SSD compared with HDD.

6.3.4 Checkpoint Interval

To examine the impacts of checkpoint interval, we set the checkpoint interval as 1, 2, 5, 10, 15 and 20 minutes. Besides, we also evaluate the performance without write-ahead log and checkpoint I/Os. We set the read proportion as 50%, and run the experiments with 200GB data.

As shown in Table 4, the throughput without write-ahead log and checkpoint is only slightly improved, which demonstrates that the write-ahead log I/Os and checkpoint I/Os have negligible impacts on the system performance when the checkpoint interval is larger than 5 minutes. The reason is that both write-ahead log I/Os and checkpoint I/Os are relatively few compared with the compaction I/Os when the checkpoint interval is larger than 5 minutes. However, excessively frequent checkpoints may harm the system performance. The experimental results show that significant loss of system throughput occurred when the checkpoint interval is shortened to 2 minutes or 1 minute. The reason is that HDD performance is limited by mechanical constraints such as disk-head movement. Frequent write-ahead log I/Os and checkpoint I/Os incur significant disk-head seek latency and rotational latency.

However, the recovery time without write-ahead log and checkpoint is almost 50 times larger than that with write-ahead log and checkpoint, which confirms the necessary of write-ahead log and checkpoint mechanism. Different from the significant system throughput variation when the checkpoint interval is less than 5 minutes, the recovery time is kept basically stable. The reason is that the recovery I/Os are sequential and the recovery disk I/O bandwidth is not sensitive to a small amount of write-ahead log and checkpoint data.

6.4 Suitability Analysis of SkipStore and Evaluation on SkipFS

SkipStore can be used in a file system. We further develop SkipFS by taking SkipStore as the metadata and small files storage engine of file system, which is similar to TableFS[32] that takes RocksDB as the metadata and small files storage engine. We conduct experiments to show the performance of SkipFS whose storage engine is SkipStore compared with TableFS whose storage engine is RocksDB. There are four transactions in postmark[34], namely, Create, Read, Append and Delete whose proportion can be dynamically adjusted. We first generate small files by Create transaction with total size of 5GB (load phase) and then execute Read and Append transactions with total operations of 1,000,000 (run phase). The size of each file ranges from 512B to 4KB. It is worth noting that for Create transaction, there are two get operations that get metadata according to file name before and after the creation of inode respectively as well as two put operations that put inode and file content respectively. For Read transaction, there are two get operations that get metadata and content respectively. For Append transaction, there are two get operations that get metadata and content respectively as well as one put operation that put the appended file.

From Figure 12 we can see that SkipFS outperforms TableFS. When executing Create transaction, the performance improvement ratio of SkipFS compared with TableFS is about 136%. Although there are two get operations in Create transaction, the data size of get operation is much less than that of put operation. From above analysis and experimental results, we know that the performance of put operation of SkipStore is much better than that of RocksDB, so SkipFS outperforms TableFS when executing Create transaction. When executing Read and Append transactions, whose proportion of Read transaction is set from 0% to 20%, the performance improvement ratio of SkipFS compared with TableFS is about 13.3% and 11.8% respectively, which is much smaller compared with Create transaction. Note that there are two get operations that get metadata and content respectively as well as one put operation that put the appended file for append transaction, and two get operations that get metadata and content respectively for read

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2609912, IEEE Transactions on Parallel and Distributed Systems

transaction. So when we mix the read and append transaction together, the data size of get operation should be larger than that of put operation. Since SkipStore mainly focus on the optimization of put performance, the lower data size of put operation in run phase results in the smaller performance improvement of Read and Append transactions.

7 Related Work

7.1 Improve compaction and write performance

To improve the compaction and write performance, the following several methods have been adopted. First, exploiting and fully utilizing the available computing and I/O resources. PCP [23] uses a pipelined compaction procedure that makes use of the parallelism of underlying hardware including CPUs and I/O devices to speed up the compaction procedure. Second, reducing the unnecessary data blocks moving. VT-tree [4] uses the stitching technique to avoid unnecessary disk I/Os for sorted and non-overlap key range. But the stitching technique may incur fragmentation which degrades the performance of scans and compactions. Third, decreasing the frequency of compactions. bLSM [1] uses the replacement-selection sort algorithm in the compaction procedure of MemTable to increase the length of sorted key-value pairs. However, it ignores a large proportion of compaction I/O activities caused by other key-value pairs. Fourth, accelerating the data flow. bLSM [1] and PE [20] partition the key range into multiple sub-key range and confine compactions in hot data key ranges, which accelerate the data flow. Finally, other optimizing approaches. LSM-trie [27] improve the write performance by optimizing the data organization based on LSM-tree by sacrificing the supporting of scan operations. Fast Compaction [28] formally define compaction as an optimization problem, and prove that compaction approach using a balanced tree is most preferable. *Atlas* [29] is proposed for ARM processors based cloud storage servers and separate the metadata and data to enable efficient data coding and storage in distributed environments. LOCS [30] is designed for customized SSD to leverage the multiple channels of an SSD to exploit its abundant parallelism. Hyeontaek Lim *et.al.* [35] propose an accurate and fast evaluation model of MSLS to find the optimized system parameters. WiscKey [36] is proposed to minimize I/O amplification of LSM-tree-based key-value store via separating keys from values in SSD environments.

Most of the existing approaches to improve the performance of LSM-tree rely on the storage devices parallelism or special workload characteristics. In this paper, we explore the component-by-component flowing down mechanism, which is the root source of the write amplification of LSM-tree. Then we propose Skip-tree to aggressively push some KV items top-down via skipping some components. Note that Skip-tree does not rely on specific storage device characteristics. Since bLSM, PCP and SkipStore solve the compaction performance problem from the different aspects, SkipStore is orthogonal to most of the existing schemes.

7.2 Improve read performance

To improve the read performance, the following several methods have been adopted. First, some schemes, such as RocksDB [10], Cassandra [19] and bLSM [1], adopt bloom filter to avoid the unnecessary disk I/Os incurred by checking the existence of one specific KV item. Second, exploiting fractional cascading [21] and partitioned exponential file [20] to confine the search data extent. COLA[5] and FD-tree[7] use forward pointers to improve the lookup performance. bLSM [1] and the partitioned exponential file [20] partition the key range into multiple sub-key ranges, and then point queries just search one smaller sub-key range. Third, promoting partial KV items from larger components to smaller components. GTSSL [6] uses the reclamation and re-insert techniques to put key-value pairs in smaller components to decrease the count of components that point queries go through.

SkipStore boosts the get performance via the following two approaches. On the one hand, using bloom filter is able to avoid the unnecessary disk I/Os for the component which does not contain the specific key. On the other hand, fully utilizing the buffer components is able to improve the cache hitting ratio of KV items. Different from GTSSL whose read performance improvement is at the cost of significantly increasing compaction I/O activities and decreasing the write performance, SkipStore improves the read performance via further exploiting the skipping potential of the buffer components in the design of LSM-tree.

8 Conclusion

The component-by-component flowing down mechanism in LSM-tree incurs significant write amplification and limits the write throughput. In this paper, we propose Skip-tree to make the KV items top-down moving much faster via skipping some components. Skip-tree can effectively reduce the write amplification and thus improve the system throughput. We design and implement one high performance key-value store, named SkipStore, based on Skip-tree. Extensive experiments demonstrate that SkipStore speeds up RocksDB by 66.5%. SkipStore is suitable for both put-intensive cloud oriented workloads and scan-intensive data analysis oriented workloads, and thus can be used as the back-end storage engine for both cloud storage systems and data analytical processing systems.

9 Acknowledgments

This work was partially supported by National Science Foundation of China under Grant No. 61303056, and Youth Innovation Promotion Association, CAS, No.2016146. Bingsheng's work is supported by a MoE AcRF Tier 1 grant (T1 251RES1610) and a NUS startup grant in Singapore.

References

[1] R. Sears and R. Ramakrishnan. *bLSM: A General Purpose Log Sturctured Merge Tree*. in SIGMOD'2012.
[2] F. Chang, J. Dean, et.al. *Bigtable: A Distributed Storage System for Structured Data*. in ACM Trans. Comput. Syst. 26(2): 1-26.
[3] P. O'Neil, E. Cheng, et.al. *The Log-Structured Merge-Tree (LSM-Tree)*. in Acta Informatica, 1996.
[4] P. Shetty, R. Spillane, et.al. *Building Workload-Independent Storage with VT-Trees*. in FAST'2013.
[5] M. A. Bender, M. Farach-Colton, et.al. *Cache-Oblivious Streaming B-Trees*. in SPAA'2007.
[6] R. P. Spillane, P. J. Shetty, et.al. *An Efficient Multi-Tier Tablet Server Storage Architecture*. in SoCC'2011.
[7] Y. Li, B. He, et.al. *Tree indexing on Solid State Drives*. in VLDB'2010.
[8] HBase main page. http://hbase.apache.org/.
[9] LevelDB main page. https://code.google.com/p/leveldb/.
[10] RocksDB main page. http://rocksdb.org/.
[11] G. DeCandia, D. Hastorun, et.al. *Dynamo: Amazon's Highly Available Key-Value Store*. in SOSP'2007.
[12] B. F. Cooper, R. Ramakrishnan, et.al. *PNUTS: Yahoo!'s Hosted Data Serving Platform*. in VLDB'2008.
[13] M. A. Olson, K. Bostic, et.al. *Berkeley DB*. in USENIX ATC'1999.
[14] Redis. *http://redis.io/*.
[15] Tair, *http://tair.taobao.org/*.
[16] Memcached. *http://memcached.org/*.
[17] B. Atikoglu, Y. Xu, et.al. *Workload Analysis of a Large-Scale Key-Value Store*. In SIGMETRICS'2012.
[18] R. Nishtala, H. Fugal, et.al. *Scaling Memcache at Facebook*. in NSDI'2013.
[19] A. Lakshman, P. Malik. *Cassandra: A Decentralized Structured Storage System*. In SIGOPS Oper. Syst. Rev., 2010.
[20] C. Jermaine, E. Omiecinski, W. G. Yee. *The Partitioned Exponential File for Database Storage Management*. In VLDB'2007.
[21] B. Chazellel and L. J. Guibas. *Fractional Cascding: I. A Data Structuring Technique*. Algorithmica, 1(2), 1986.
[22] Y. Yue, B. He, L. Tian, H. Jiang, et. al. *Rotated Logging Storage Architectures for Data Centers: Models and Optimizations*. in IEEE Transactions on Computers, 65(1), 2016.
[23] Z. Zhang, Y. Yue, B. He, et.al. *Pipelined Compaction for the LSM-tree*. in IPDPS'2014.
[24] B. F. Cooper, A. Silberstein, et.al. *Benchmarking Cloud Serving Systems with YCSB*. in SoCC'2010.
[25] Q. Huang, K. Birman, R. van Renesse, et al. *An analysis of Facebook photo caching*. in SOSP'2013: 167-181.
[26] Park, Dongchul, and D. H. C. Du. *Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters*. MSST'2011: 1-11.
[27] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. *LSM-trie: An LSM-treebased Ultra-Large Key-Value Store for Small Data*. USENIX ATC'2015.
[28] M Ghosh, I Gupta, S Gupta, N Kumar. *Fast Compaction Algorithms for NoSQL Databases*. in ICDCS'2015.
[29] C Lai, S Jiang, L Yang, S Lin, G Sun, et.al. *Atlas: Baidu's key-value storage system for cloud data*. in MSST'2015.
[30] Peng Wang, Guangyu Sun, Song Jiang, et.al. *An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD*. in EuroSys'2014.
[31] Bloom Filter. *https://antognini.ch/papers/BloomFilters20080620.pdf/*.
[32] Kai Ren and Garth Gibson. *TABLEFS: Enhancing Metadata Efficiency in the Local File System*. in USENIX ATC'2013.
[33] M. Rosenblum and J. K. Ousterhout. *The Design and Implementation of a Log-Structured File System*. *ACM TOCS*, 10(1):26-52, 1992.
[34] Jeffrey Katcher. *PostMark: A New File System Benchmark*. In *NetApp Technical Report TR3022* (1997).
[35] Hyeontaek Lim, David G. Andersen and Michael Kaminsky. *Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs*. In *FAST'2016*.
[36] Lanyue Lu, T. S. Pillai, Andrea C. Arpaci-Dusseau, et.al, *WiscKey: Separating Keys from Values in SSD-conscious Storage*. In *FAST'2016*.
[37] B. He and J. X. Yu and A. C. Zhou, *Improving Update-Intensive Workloads on Flash Disks through Exploiting Multi-Chip Parallelism*. In *IEEE TPDS, 26(1),2015*
[38] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, et.al. *Walnut: A unified cloud object store*. In *SIGMOD'2012*.
[39] Apache Hadoop. *https://hadoop.apache.org/*

Yinliang Yue received the PhD degree in computer architecture from Huazhong University of Science and Technology (HUST), China, in 2011. He is an associate professor in the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include computer architecture and storage systems. He has over 10 publications in major journals and international conferences including TC, ICDCS, IPDPS, etc.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Yuzhe Li received his master degree in Computer Engineering from University of Science and Technology of China in 2015. He is a research assistant in the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include computer architecture and storage systems.

Weiping Wang received the Ph.D degree in computer science from Harbin Institute of Technology, China, in 2008. He is a professor in the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include database and storage systems. He has over 20 publications in major journals and international conferences.