2019 IEEE 35th International Conference on Data Engineering (ICDE)

LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores

Yunpeng Chai[1,2], Yanfeng Chai[1,2], Xin Wang[3*], Haocheng Wei[1,2], Ning Bao[1,2] and Yushi Liang[1,2]

[1]Key Laboratory of Data Engineering and Knowledge Engineering, MOE, China
[2]School of Information, Renmin University of China, China
[3]College of Intelligence and Computing, Tianjin University, China
[*]Corresponding Author: wangx@tju.edu.cn

***Abstract*—Log-structured merge (LSM) tree key-value (KV) stores have been widely deployed in many NoSQL and SQL systems, serving online big data applications such as social networking, bioinfomatics, graph processing, machine learning, etc. The batch processing of sorted data merging (i.e., compaction) in LSM-tree KV stores greatly improves the efficiency of writing, leading to good write performance and high space efficiency. Recently, some lazy compaction methods were proposed to further promote the system throughput through delaying the compaction to accumulate more data within a compaction batch. However, the batched writing manner also leads to significant tail latency, which is unacceptable for online processing, and the newly proposed lazy approaches worsen the tail latency problem. Furthermore, the unbalanced read/write performance of the widely deployed SSDs make the performance optimization harder. Aiming to optimize both the tail latency and the system throughput, in this paper, we propose a novel Lower-level Driven Compaction (LDC) method for LSM-tree KV stores. LDC breaks the limitations of the traditional upper-level driven compaction manner and triggers practical compaction actions by lower-level data. It has the benefits of both decreasing the compaction granularity effectively for smaller tail latency and reducing the write amplification of LSM-tree compaction for higher throughput. We have implemented LDC in LevelDB; the experimental results indicate that LDC can reduce the 99.9th percentile latency for 2.62 times compared with the traditional upper-level driven compaction mechanism, and achieve 56.7% $\sim$ 72.3% higher system throughput at the same time.**

***Index Terms*—LSM-tree, compaction, SSD, KV store, tail latency**

I. Introduction

Key-value stores have been increasingly adopted as the low-level storage engines for many mainstream NoSQL and SQL systems like BigTable [1], Cassandra [2], HBase [3], HAWQ [4], and TiDB [5], especially in many online big data applications such as social networking [6], bioinfomatics [7], graph processing [8], and machine learning [9]. Moreover, Facebook has used key-value storage engines to replace traditional engines like innoDB [10] for relational database systems (e.g., MySQL [11]) in online data management to achieve higher performance and space efficiency in order to manage large-scale social graph data more efficiently [12].

The development of key-value (KV) storage engines exhibits the following two trends during the last few years:

1) KV trends in terms of *Software*: LSM-tree. The percentage of write requests in many big data applications keeps increasing. Due to the significantly improved write performance and space efficiency [12], the log-structured merge tree (LSM-tree) based KV stores have gradually replaced traditional ones based on B+-tree. The essential factor lies in that the LSM-tree promotes write performance by buffering writes in memory, batching writes to I/O devices, and performing the sequential I/O access mode for throughput boost. However the batched writes make the stored data less ordered, leading to some read amplification. In other words, the LSM-tree trades read performance for write boost, which fits in the recent application feature of larger write proportions and thus achieves higher system throughput.

2) KV trends in terms of *Hardware*: SSDs. Many big data systems have been upgraded to utilize the fast flash-based Solid State Drives (SSDs) for higher performance, when the SSD products are getting larger and cheaper gradually. Nevertheless, SSDs have their own drawbacks: (1) Unlike the balanced read and write performance of HDDs, the write operations of SSDs are usually one or two orders of magnitude slower than its read operations due to the necessary slow erase operations before re-writing and the inherent write amplification problem of flash chips [13]. (2) SSDs usually have the limited write endurance problem (e.g., each cell on the common flash chips can only be allowed for 5,000 $\sim$ 10,000 times of re-writing before wearing out [14]).

In this case, SSD-oriented LSM-tree KV stores should trade more read performance for less writes because of SSDs' fast reading, slow writing, and limited write endurance. Thus a common idea is to enlarge the writing batch through some lazy scheme to accumulate more data for reducing the re-writing frequency, such as the universal compaction mechanism in RocksDB [15], dCompaction [16], and PebblesDB [17].

For example, *RocksDB* [15] implements a universal compaction manner that puts all newly-written files in the same level, i.e., not sorting the new key-value pairs among different files. Only when the accumulated new files reach a threshold, all of these new files are merged into the old ones. The worst case is that all the stored data are involved into one round of compaction, leading to huge tail latency.

dCompaction [16] introduces the concept of virtual file to delay the actual I/Os of merging new data into the existing ones in order. The result is that some I/O operations may be saved, but each round of merging will involve more data and be executed in longer time, leading to serious performance

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00070

fluctuations (i.e., very unstable system throughput and request latency).

Motivation. For the lazy approaches like RocksDB or dCompaction, the enlarged writing batches increase the performance fluctuations of LSM-tree-based KV stores. For online big data systems, the unstable performance and large tail latencies are unacceptable for users. In order to observe the latency fluctuation, we have conducted a micro benchmark experiment by performing a YCSB [18] workload mixed with 10 millions of read and 10 millions of write requests on LevelDB, a widely used open-source LSM-tree KV store developed by Google [19]. The experimental results show that LSM-trees usually lead to large tail latency and a drastic performance fluctuation. The average latency per second of all the requests are plotted in Fig. 1. The fluctuation extent of the write latency reaches up to 49.13 times compared with the smallest latency. When a round of batched writing is not completed in LSM-trees, the user write requests have to wait, leading to a much larger tail latency for users.

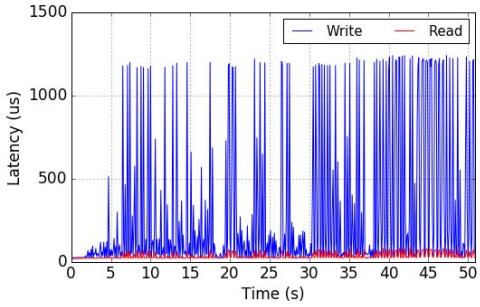


Fig. 1. Serious latency fluctuations caused by batched writing.

Therefore, for online big data applications, the **optimization objectives** of LSM-tree KV stores is *to reduce the tail latency and at the same time improve the system throughput of LSM-tree KV stores*. To the best of our knowledge, existing solutions cannot achieve both of the goals. For example, existing lazy schemes move the balance point more close to achieving higher throughput by enlarging compaction granularity, but away from reducing the tail latency.

Basic Idea. As Fig. 2 (a) plots, the key operation of LSM-tree is merging the new sorted data set into the existing one. Usually, the size of new data set (i.e., m) is much smaller than the existing data set (i.e., n). This data merging process is called *compaction*. The traditional compaction of LSM-tree works in an *upper-level-driven* manner: the new data set located in the upper-level of LSM-tree determines the data set involved in the compaction (i.e., $m + n$). This manner leads to two drawbacks: (1) Many extra I/Os are triggered to decline the throughput of LSM-tree, because the data in the existing data set (i.e., n) have to be read into memory, and then be written into the persistent storage again after sorting. (2) The compaction granularity is large (i.e., $m + n$) and it takes a long time to accomplish, along with the side-effect, i.e., large tail latency. Moreover, the above mentioned lazy schemes of enlarging write batches in the LSM-tree may increase the compaction granularity and worsen the tail latency.

In this paper, in order to achieve both low tail latency and high throughput, we propose a novel ***Lower-level Driven* Compaction (LDC)** mechanism. As Fig. 2 (b) shows, a small sorted data subset (e.g., E_i) located in the lower level triggers the compaction by pulling down some small data sets in the upper level with the same key range as E_i. LDC optimizes the tail latency and throughput of LSM-tree at the same time: (1) LDC breaks down the large compaction job into more efficient small ones, leading to less user request waiting and smaller tail latency. (2) Our proposed LDC has a mechanism to accumulate approximately the same amount of data in the upper level (e.g., S_1, S_2, S_3, ..., S_k) as E_i (see Section III for more details), so the extra I/Os (i.e., the I/Os of the involved lower-level data) in a compaction process are much reduced to improve the actual throughput of LSM-tree KV stores. We have implemented our proposed LDC mechanism in LevelDB. The experimental results indicate that LDC can reduce the 99.9th percentile latency from 469.66 us to 179.53 us, declined by 2.62 times, and at the same time lead to a 56.7% $\sim$ 72.3% higher throughput on average for typical workloads compared with the traditional upper-level driven compaction manner.

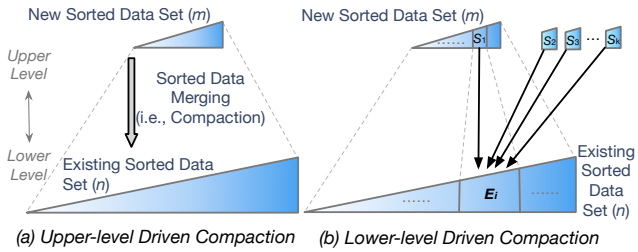


Fig. 2. The upper-level and the lower-level driven compactions.

Contributions. Our contributions in this paper can be summarized as follows:

(1) *Breaking the limitation of the traditional upper-level driven compaction manner.* We propose a novel Lower-level Driven Compaction (LDC) algorithm, which can break large batched writing job of merging new data into small pieces to not only reduce the tail latency, also reduce the extra compaction I/Os significantly for higher throughput at the same time.

(2) *Improving both the tail latency and the throughput for LSM-tree KV stores.* We have implemented LDC in a widely used LSM-tree KV storage engine, i.e., LevelDB. LDC can reduce the 99.9th percentile latency for 2.62 times, which improves the quality of service significantly for online big data applications.

(3) *Especially fitting new hardware like SSDs.* Our proposed LDC method is particularly suitable for the hardware features of SSDs very well, promoting the system performance and lengthening the lifetimes of SSDs significantly by cutting down the compaction I/Os by about 50%.

The rest of this paper is organized as follows. Section II introduces the background of our research and models the LSM-tree performance. In Section III, we describe the rationale of our proposed Lower-level Driven Compaction (LDC). Section IV exhibits the extensive experiments and

comparisons, followed by the related works in Section V. Finally we conclude this paper in Section VI.

II. Modeling LSM-tree Performance

In this section, we first introduce some background about LSM-tree KV stores (Section II-A), and then the performance model and the performance analysis of LSM-tree KV stores will be presented (Section II-B and II-C, respectively).

A. Background

Definition 2.1: **(LSM-tree)** The Log-Structured Merge (LSM) tree is a data structure composed of a series of data set C_i ($0 \leq i \leq N$), where C_i is a data structure in order, and $|C_i| < |C_{i+1}|$.

Definition 2.2: **(MemTable)** *MemTable* is the ordered data set in the first level (i.e., C_0) which is stored in memory for higher performance in an LSM-tree.

For a key-value store, a *MemTable* maintains key-value pairs ($k \rightarrow v$) according to the order of k. All the newly inserted or updated key-value pairs are first written into a *MemTable*. Only after a *MemTable* is full, it will then be written into the persistent storage, i.e., converting the slow fine-grained writing to the fast batched one for I/O devices.

Definition 2.3: **(SSTable)** *SSTable* is a data set (i.e., S_i, $1 \leq i \leq N$) located in persistent storage storing key-value pairs in the sequential order of keys.

Definition 2.4: **(Compaction)** Compaction is the behavior of merging all or part of the data in C_i to the sorted structure in lower levels, i.e., C_j ($j > i$).

In most cases, we merge the data in C_i into the next level C_{i+1}. Assuming $S_m^i (m \in [M_1, M_2])$ represents the *SSTables* in level i that are selected to be merged into the lower level $i+1$ and $S_n^{i+1} (n \in [N_1, N_2])$ are the *SSTables* that locate in level $i+1$ and have the overlapping key ranges with all S_m^i, the compaction operation first loads all S_m^i and S_n^{i+1} into memory, then sorts all the key-value pairs through a merge sort, and finally constructs some new *SSTables* in level $i+1$ and write these new *SSTables* into I/O devices.

Definition 2.5: **(Fan-out)** *Fan-out* is the capacity ratio of adjacent levels in an LSM-tree, i.e., $|C_{i+1}|/|C_i|$.

The size of the lower levels in LSM-trees usually grows exponentially. In practice, the total size ratios between adjacent layers of LSM-trees may deviate from the settings of *fan-out*. In this case, we can adjust the size distribution of all layers by selecting appropriate compaction targets.

Definition 2.6: **(Write Amplification)** Write amplification indicates the phenomenon that the physically performed I/Os are larger than the user's written data.

When merging the upper-layer data into the larger lower layer, the compaction will trigger reading all the related *SSTables* into memory for merge sort and writing all the sorted data into persistent storage again to form new *SSTables* in the lower layer. This procedure causes multiple times of additional I/Os compared with the original upper-layer data.

The write amplification rate caused by one compaction operation is $\Sigma_n|S_n^{i+1}|/\Sigma_m|S_m^i|$, and the value can usually be estimated as the *fan-out* of an LSM-tree. The I/ O amplification phenomenon usually happens several times when merging data down layer by layer; the total write amplification rate is $\Sigma_n|S_n^{i+1}|/\Sigma_m|S_m^i| \times H_{LSM}$, where H_{LSM} is the height of the LSM-tree.

Example 2.1: *LSM-tree Structure and Compaction Behavior in LevelDB.* As shown in Fig. 3, when serving write requests, LevelDB first sorts and places data in *MemTable*. When a MemTable reaches its capacity threshold (e.g., 2MB), it is marked as an *Immutable MemTable*, indicating the contents in it cannot be changed, and will later be dumped into the disk and stored as a file (i.e., *SSTable*). An SSTable stores the sorted key-value pairs and some auxiliary data like a Bloom filter to judge if a key exists in the SSTable with low overhead. In LevelDB, the top layer, i.e., Level 0, is simply a collection of the newly dumped SSTables to support fast insertion. The SSTables in Level 0 usually have overlapping key ranges with each other, while the other lower levels restrict their SSTables to be sorted and have no overlaps with each other.

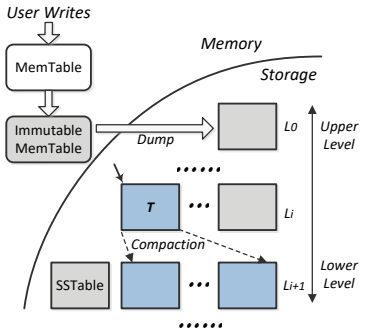


Fig. 3. Compaction mechanism in LSM-trees.

In an LSM-tree, compaction is an operation performing the most I/Os, so it is the performance bottleneck in LSM-tree KV stores. For instance, we used a performance analysis tool, i.e., *perf* [18], in Linux to record the total execution time of each functions of the LevelDB source codes when inserting 10 millions of key-pairs. The experiment was performed based on an enterprise-level SSD (see Section IV for more). Table I lists the top functions which consume the most time. The top two parts (i.e., the function *DoCompactionWork* and *file system*) consume the most time due to compaction jobs.

TABLE I
The most time-consuming functions in LevelDB.

Module	Percent of Time
DoCompactionWork	61.4%
file system (kernel)	20.9%
DoWrite	8.04%
Others	9.66%

B. Performance Model

LSM-tree Assumptions. We assume the fan-out of an LSM-tree is *k*, the size of each *SSTable* is *b*, and the total data amount of the whole LSM-tree is *n*. So the count of SSTables is n/b, and the number of LSM-tree levels is $\log_k(n/b)$ [20]. Assuming the first level (i.e., Level 0) contains u SSTables with overlapping key ranges with each other, read operations requires reading additional u SSTables to look for the target

key-value pairs. The symbols used in the following part are summarized in Table II.

TABLE II
SYMBOLS USED IN THE LSM-TREE PERFORMANCE MODEL.

Symbols	Descriptions
k	Fan-out of an LSM-tree
b	Size of an SSTable
n	Total data amount of the LSM-tree
u	Count of unsorted SSTables in the first level
th_w, th_r, th	Write, read, and total throughput of the LSM-tree
th_w^{ssd}, th_r^{ssd}	Write and read throughput of SSDs
a_w, a_r	Write and read amplification rates of the LSM-tree
r_w	Ratio of write requests in user workloads
tl_w	Tail latency of write operations
c	SSTable count in upper-level involved in compaction

Based on the above assumptions, the I/O amplification of write and read operations can be calculated as the following theorems.

Theorem 2.1: The write amplification rate of an LSM-tree coupled with the traditional upper-level driven compaction is $O(k \times \log_k(n/b))$.

Proof (sketch): The total number of *SSTables* is n/b and each level is k times larger than the previous one, so the height of the LSM-tree is equal to $\log_k(n/b)$. In each time of compaction, we assume one *SSTable* is first selected to perform compaction, i.e., merging it into the lower layer. Because the lower layer is k times larger than the current layer, the average count of *SSTables* with the overlapping key range with the selected *SSTable* in the lower level is O(k). Therefore, considering one *SSTable* newly written to the LSM-tree, it will be re-written for O($k \times \log_k(n/b)$) times before putting it in the lowest layer of the LSM-tree, i.e., the write amplification rate is O($k \times \log_k(n/b)$).

Theorem 2.2: The read amplification rate of an LSM-tree coupled with the traditional upper-level driven compaction is $O(\log_k(n/b) + u)$ when the key ranges of *SSTables* are maintained in in-memory indexes.

Proof (sketch): We assume the key ranges of all the SSTables are maintained in an in-memory index. Each layer of an LSM-tree is fully sorted, i.e., the key-value pairs in one SSTable is sorted and the key ranges of SSTables are not overlapped, but the SSTables in different layers may have overlapping key ranges. When serving a read request, we need to check each layer in a top-bottom manner to look for the target key. Furthermore, the first layer contains u unsorted SSTables, so we also need to check them first. Therefore, the read amplification rate of the LSM-tree is related to the height of the LSM-tree and the count of these unsorted SSTables in Level 0, i.e., $O(\log_k(n/b) + u)$.

In addition, it is a reasonable assumption that the SSTable key range index can be maintained in memory. For instance, assuming n=10TB, b=2MB, the SSTable count (i.e., n/b) will be 5 millions. Because the key range of one SSTable only contains two values of maximum and minimum keys, when the length of one key is 16 bytes, the total memory consumption of this index is only 160MB (i.e., 5M×2×16B).

Throughput. The write and read throughput of an LSM-tree KV store (i.e., th_w and th_r, respectively) are shown as (1), where th_w^{ssd} and th_r^{ssd} are the write and read throughput of SSDs, and a_w and a_r are the LSM-tree write and read amplification rates, respectively. Usually th_r^{ssd} is much larger than th_w^{ssd} for SSDs, leading to larger read throughput than the write one in most cases.

$$\begin{aligned} th_w &= th_w^{ssd}/a_w \\ th_r &= th_r^{ssd}/a_r \end{aligned} \tag{1}$$

Assuming r_w is the write request ratio of workloads, the total system throughput (i.e., th) is shown as (2).

$$th = \frac{1}{\frac{r_w}{th_w} + \frac{1-r_w}{th_r}} \tag{2}$$

Tail Latency. Recall Fig. 1 that the request latencies fluctuate significantly (from several us to more than 1ms). The reason lies in that the periodical heavy compaction operations blocks the requests, especially for write requests. If write requests are always allowed, even when a compaction is working, the unsorted *SSTables* in Level 0 will increase too rapidly to slow down the read performance too much (i.e., a too large value of u in Theorem 2.2). Therefore, the tail latency of LSM-tree KV stores mainly comes from write requests which are blocked by the ongoing compaction.

For the traditional compaction manner, each round of compaction involves k additional SSTables on average when moving one upper-level SSTable down. So the tail latency of write operations (i.e., tl_w) is presented as (3), where t_w is the consumed time of writing data into the *MemTable* which can be considered as a negligible constant (i.e., p), c is the number of the selected *SSTables* each time for compaction, and th_{read} is the average device bandwidth occupied by serving the read requests at the same time when performing the compaction.

$$\begin{aligned} tl_w &= t_{compaction} + t_w \\ &= \frac{(k+1) \times c \times b}{th_w^{ssd} - th_{read}} + p \end{aligned} \tag{3}$$

C. Performance Analysis

Based on the above performance model of LSM-tree KV stores, we can get the following points:

1) Existing lazy compaction solutions contribute to reduce write amplification rate, but enlarge the tail latency. According to Theorem 2.1, the LSM-tree write amplification comes from two aspects: the first one is the amplification happened in each round of compaction (proportional to *fan-out*, i.e., k), and the second one is the amplification of rewriting data layer by layer (proportional to the height of the LSM-tree, i.e., $\log_k(n/b)$). Existing lazy compaction solutions [15]–[17] usually aim to put off some compactions, i.e., skipping compactions in some layers, to reduce the total amplification rate. However, these methods increase the granularity of compaction operations, i.e., increase c in (3), usually leading to worse tail latency.

2) The key for throughput optimization lies in reducing the write amplification introduced by each round of compaction. According to Theorem 2.1, the traditional compaction algorithm introduces k more *SSTables* in the lower level for merge sort when writing one *SSTable* down to the lower level,

because the lower level is multiple (e.g., 10) times larger than the upper level. However, if we can involve data less than k *SSTables* in the compaction, the LSM-tree write amplification rate will be significantly reduced, leading to higher throughput.

3) Making a good balance between reading and writing to promote the total throughput. According to (2), a too small value of th_w will slow down the process of write requests significantly, blocking the following read requests and declining the total throughput. For example, when r_w is 0.5, th_r is 10MB/s, and th_w is only 1MB/s, the total throughput is only 1.82MB/s. If we can promote th_w to 2MB/s, even if th_r drops to 5MB/s, the total throughput can be improved to 2.86MB/s, 57% higher than the former, although the sum of th_r and th_w declines. Therefore, for new storage hardware like SSDs with unbalanced read/write performance, we should give higher priority to improving the write throughput of the LSM-tree to promote the overall performance.

4) Reducing the SSTables involved in each round of compaction is conductive to tail latency reduction. As (3) indicates, if we can reduce both c and a_w effectively, each round of compaction can be finished in a short time. Write operations will be blocked for less time and the tail latency will be reduced significantly. Therefore, the I/O reduction in each round of compaction is also conducive to reducing tail latency by shrinking each compaction job.

III. The Lower-Level Driven Compaction

In this section, we first present the basic idea of our proposed Lower-level Compaction (LDC) algorithm in Section III-A. Then the algorithm description will be given in Section III-B, followed by the performance analysis and some discussions in Section III-C and III-D, respectively.

A. Overview

If we perform the lower-level driven compaction directly in an LSM-tree, the compaction actions include first selecting an appropriate SSTable and then getting the upper-level SSTable with the same key range for merge sort. However, due to the capacity ratio k between adjacent layers, the involved lower-level data amount in each round of compaction is still k times larger compared with the upper-level data. Therefore, in order to reduce the write amplification rate, we should *accumulate more upper-level data for an SSTable for compaction*.

Based on the above idea, our proposed LDC algorithm splits the traditional LSM-tree compaction action into two separate steps. (1) When an upper-level SSTable is selected for being merged into the next level, we do not perform actual I/Os to execute the compaction immediately. Instead, we adopt a light-weighted *link* action to connect the segments of the upper-level SSTable, which are called *slices*, to the related lower-level SSTables according to their key range overlaps. Then we move the upper-level SSTable into the *frozen region*, i.e., out of the management of an LSM-tree. (2) Only when a lower-level SSTable has accumulated enough *slices* with the overlapping key ranges, the actual I/O operations of *merging* the upper-level contents down into the lower level are performed.

Example 3.1: *LDC contains two phases, i.e., link and merge.* As shown in Fig. 4 (a), when SSTable A is determined to be moved down, we just *link* its *slices* to the related lower-level SSTable with the same key ranges (i.e., SSTable B, C, and D). As Fig. 4 (b) plots, when one lower-level SSTable (e.g., C) has accumulated nearly the same amount of data as itself from multiple file slices in the upper level (e.g., A) and the *frozen region* (e.g., X and Y), actual I/Os will be performed at this time to complete the data merging.

For the traditional upper-level driven compaction (UDC), for example, when performing the compaction immediately in the case of Fig. 4 (a), the original compaction needs to perform additional reading and writing of three low-level SSTables (i.e., *B*, *C*, and *D*) in order to merge *A* into a sorted structure. LDC can reduce the additional I/O consumption to about the same amount of the necessary I/Os.

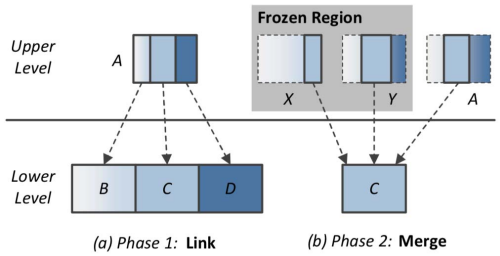


Fig. 4. Basic idea of Lower-level Driven Compaction (LDC).

B. Algorithm Description

The following Algorithm 1 exhibits the key parts of LDC (i.e., the link and the merge phases).

*1) **Link**:* When a compaction process starts, the level whose size exceeds the expected value the most severely according to the LSM-tree fan-out setting will be chosen for compaction. Then an SSTable in this table will be selected as the compaction target (i.e., s_u) according to methods like round-robin.

As Lines 1 $\sim$ 3 of Algorithm 1 indicate, the SSTables with the overlapping key ranges with s_u located in the next level of the LSM-tree will be identified as S_l, and s_u will be frozen, i.e., removed from the LSM-tree. Then, for each SSTable s_l in S_l, we make a *slice* from s_u with the same key range as s_l, and link this slice to s_l, as shown in Lines 4 $\sim$ 7. This link is referenced as a *SliceLink*. Finally, when the accumulated slice number of a lower-level SSTable reaches the specified threshold T_s, the merge phase of a lower-level driven compaction action will be triggered, as illustrated in Lines 8 $\sim$ 9.

In the link phase, we only establish a relationship between the slices in the upper-level SSTable and the lower-level SSTables with overlapping key ranges. No actual I/O operations are performed in this step; only some in-memory metadata needs to be processed at a fast speed. Therefore, linking is a lightweight action in LDC.

*2) **Merge**:* When the count of *SliceLinks* for a lower-level SSTable is no less than T_s, the merge action will start to perform a lower-level driven compaction, i.e., executing actual I/Os to move data down to the lower level and maintain the key-value pairs sorted in the order of their keys.

Algorithm 1 LDC: Link & Merge Operations

```
1:  function link(s_u) :
2:      S_l ← getOverlappedLowerSSTs(s_u);
3:      freeze(s_u);
4:      for each s_l ∈ S_l do
5:          slice ← createFileSlice(s_u, s_l);
6:          s_l.addSliceLink(slice);
7:          s_u.reference ← s_u.reference + 1;
8:          if getSlicesNum(s_l) ≥ T_s then
9:              merge(s_l);
10: function merge(s_l) :
11:     C ← getLinkedSlices(s_l);
12:     D ← loadData(s_l, C);
13:     M ← doMergeSort(D);
14:     S ← generateNewSSTs(M);
15:     for each s ∈ S do
16:         flush(s);
17:     removeSST(s_l);
18:     for each c ∈ C do
19:         s_u ← getSST(c);
20:         s_u.reference ← s_u.reference − 1;
21:         if s_u.reference = 0 then
22:             removeSST(s_u);
```

In the merge phase, shown as Lines 10 $\sim$ 17 of Algorithm 1, we first fetch the lower-level SSTable (i.e., s_l) and all its linked slices (i.e., C) into memory, sort all the loaded key-value pairs using a merge-sort, and write the newly generated SSTables into the underlying storage devices (e.g., SSDs). These new SSTables located in the same level as s_l, and the old s_l will be removed.

When the actual I/Os are accomplished, all the linked upper-level SSTables of s_l will have a decreased reference count. If any of these upper-level SSTables have no reference, it will be deleted to save storage space, as Lines 18 $\sim$ 22 exhibit.

*3) **Modification on Read Procedure**:* Besides the link and the merge phases, LDC requires some additional modification of the read procedure in the LSM-tree KV stores. According to LDC, the frozen SSTables break away from the normal management of the LSM-tree, and the data in these frozen SSTables belong to their linked lower-level SSTables. In an LSM-tree, because data in a higher level may be a newer version compared with that in the lower levels, linked slices have higher priority for reading than the corresponding SSTable.

Although the LDC mechanism may introduce some additional requirements in reading the linked slices, the cached indexes and Bloom filters of active SSTables can help locate the position of the target data accurately to avoid most of the I/Os. Moreover, LDC is used in SSD-based key stores; the random reading performance of SSDs is much more close to their sequential reading performance compared with hard disk drives. LDC usually achieves better total throughput according to (2) because of more balanced read and write performance of KV accesses (see §IV-C for more experimental results).

*4) **Self-Adaption of the SliceLink Threshold**:* When the *SliceLink* threshold of LDC is small, we have less linked slices to read, leading to higher read performance; but the write amplification rate gets larger, with the result of worse write performance. When the the *SliceLink* threshold is large, the write amplification rates will be reduced for linking more data to the same lower-level SSTable, resulting in higher write performance but worse read performance.

Therefore, the *SliceLink* threshold can be self-adaptive to fit the read/write request ratios of practical user workloads for higher performance dynamically. For the read-dominated applications, smaller *SliceLink* thresholds can be set in LDC until the total performance cannot be improved further. For the write-dominated workloads, the *SliceLink* thresholds can be set larger periodically until we get the optimal total performance.

Example 3.2: *Operations in the link phase.* Taking the case in Fig. 5 for example, for the selected compaction target, i.e., SSTable A, we first find the related lower-level SSTables with the overlapping key ranges (i.e., B, C, and D). Assuming that B is responsible for the key range from the smallest possible key (i.e., $k_{smallest}$) to the largest key of B (i.e., k_{BH}), marked as kr_1 in Fig. 5. Similarly, C and D are also related to the key range kr_2 and kr_3 containing the intervals. Then SSTable *A* is marked as a *frozen* SSTable, meaning it will not be chosen for another round of compaction. Reading data in *A* may cause additional slice reading. For the related low-level SSTables *B*, *C*, and *D*, a new in-memory item of *SliceLink* will be created for each SSTable to remember its linked slice in SSTable *A* (i.e., A_1, A_2, and A_3 respectively) and their overlapping key ranges.

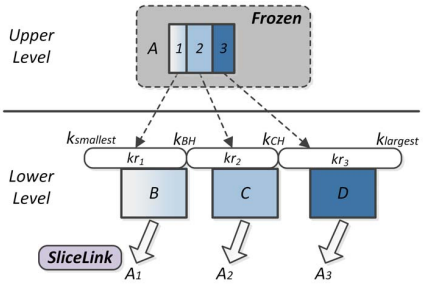


Fig. 5. An example of the *link* phase in LDC.

Example 3.3: *Operations in the merge phase.* As shown in Fig. 6, in the merge phase, we identify the corresponding *slices* in the *frozen* SSTables (i.e., A_3, B_1, and C_2) according to all the *SliceLinks* of the lower-level SSTable D. The data in the slices A_3, B_1, and C_2, not the whole SSTables A, B, and C, are loaded into memory, sorted, and then written into two new SSTables, i.e., D' and D'', in the lower level. Finally, the reference counts of the corresponding frozen SSTables will all decrease by 1. A frozen SSTable can be recycled when its reference count comes to 0.

C. Performance Analysis

Compared with the traditional UDC approach, our proposed LDC algorithm can effectively reduce the tail latency and improve the total system throughput.

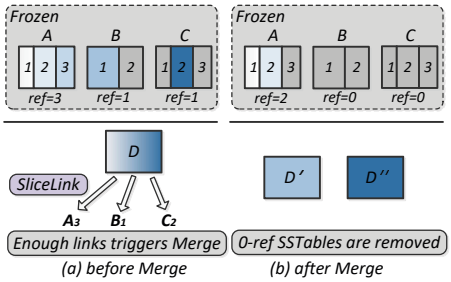


Fig. 6. An example of the *merge* phase in LDC.

Tail Latency. Recall (3) that the tail latency of the traditional UDC method is nearly in direct proportion to $(k+1)\times c\times b$, i.e., $O(k\times c)$, because b is a constant. The lazy compaction approaches [15]–[17] can skip some compaction actions but cannot reduce the write amplification of each round of compaction by following the same upper-level driven compaction mode (i.e., not reducing k). Furthermore, the lazy compaction schemes increase the compaction granularity (i.e., increasing c). Therefore, the tail latency will be enlarged by these lazy compaction schemes. On the contrary, our proposed LDC algorithm aims to reduce the granularity in each round of compaction (i.e., from $O(k)$ to $O(1)$), so the tail latency will be reduced significantly.

Throughput. LDC reduces the write amplification rate for k times compared with UDC (see Theorem 3.1), obtaining significantly improved LSM-tree write throughput. Although the read amplification rate increases (see Theorem 3.2), the total throughput will also be promoted based on higher write throughput and a bit lower read throughput since the underlying SSDs has much better read performance than the write one (please refer to the above (2)).

Theorem 3.1: The write amplification of an LSM-tree with LDC is $O(\log_k(n/b))$.

Proof (sketch): Compared with UDC, LDC does not change the fan-out (i.e., k) and the LSM-tree height (i.e., $\log_k(n/b)$). The condition of triggering lower-level driven compaction is the linked upper-level *slices* have nearly the same data amount compared with the lower-level *SSTable*. Therefore, the write amplification rate of each round compaction is $O(1)$; the total write amplification rate is $O(\log_k(n/b))$.

Theorem 3.2: The read amplification rate of an LSM-tree with LDC is $O(k\times\log_k(n/b)+u)$ when the key ranges of *SSTables* are maintained in in-memory indexes.

Proof (sketch): According to LDC, some SSTables have linked *slices*. When performing read operations on this kind of SSTables, we need to check all the linked *slices* first for the target key(s). Because a lower layer in an LSM-tree is usually k times larger than its upper layer, the data amount corresponding to the same key ranges in the lower layer will also be k times larger than those in the upper layer. In this case, considering one SSTable in the lower layer, each of its linked *slices* usually only contains $1/k$ size of an SSTable. So we need to check k *slices* additionally when performing reads. The read amplification rate of LDC will be enlarged to $O(k\times\log_k(n/b)+u)$.

However, Bloom filters are widely adopted in LSM-tree KV storage engines to help avoid reading unnecessary SSTables. Due to the high space efficiency of Bloom filters, the Bloom filters of most frequently accessed SSTables are usually cached in memory. Consequently, the practical read amplification rate of LDC will be much lower than $O(k\times\log_k(n/b)+u)$, even close to $O(\log_k(n/b)+u)$.

D. Discussions

Space Overhead of LDC is Small. Compared with UDC, LDC requires some additional temporary space, because some frozen SSTable may contain useless slices, whose data have already been merged into lower-level SSTables, e.g., the gray slices in Fig. 6. However, this overhead is acceptable due to the following reasons:

1) An SSTable with *SliceLinks* cannot be chosen for *link* to avoid making LDC too complicated. So overall, even if all the SSTables establish link relationships in pairs, the total size of all the frozen SSTables is less than 50%. On average, half of the slices in all frozen SSTables is useless, so the additional space that LDC does not release in time is at most 25% of the whole storage.

2) In fact, not all the SSTables are involved in the *link* relationship. It could be found from our experiments in §IV-J that LDC only consumes 3.37% $\sim$ 10.0% more space than the native compaction in LevelDB, 6.78% on average, much less than the above worst-case estimation. Furthermore, we can adopt a smaller *SliceLink* threshold to further reduce the additional space overhead of LDC.

Simply Adjusting LSM-trees does not Work. An intuitive idea of improving the tail latency and the throughput of an LSM-tree is to simply adjust some parameters of the LSM-tree, such as the size of *SSTables*, or fan-out of the LSM-tree. However, these methods cannot improve tail latency and throughput at the same time.

1) Setting Smaller SSTables. Smaller SSTable file size settings lead to small compaction granularity and small tail latency, but also lead to worse I/O performance due to smaller I/O granularity on storage devices.

2) Tuning Fan-outs of LSM-trees. When setting a smaller fan-out of an LSM-tree, we can reduce the average file number involved in each round of compaction (i.e., $O(k)$), but it also increases the tree depth, leading to more rounds of compactions. On the contrary, an LSM-tree with a larger fan-out has to handle more files each time (i.e., enlarged k), resulting in considerable I/O amplification. Therefore, tuning fan-outs of LSM-trees cannot effectively reduce I/O amplification and promote throughput, verified by our experimental results in Fig. 7 when the fan-out ranges from 3 to 100.

IV. Implementation and Evaluation

As a classical implementation of LSM-trees, LevelDB [19] is a popular open source key-value store library written in C++, inspired by Google's BigTable [1]. Our proposed Lower-level Driven Compaction (LDC) was implemented and integrated to LevelDB. Our work mainly includes the newly added or

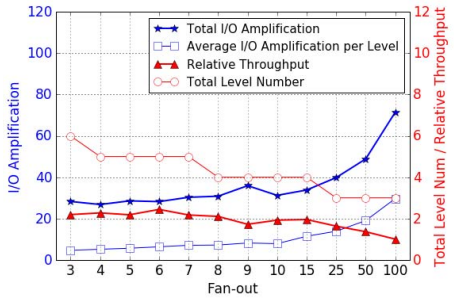


Fig. 7. Tuning *fan-out* cannot reduce amplification and promote throughput.

modified metadata to support our proposed new concepts, such as *slice*, *SliceLink*, *link*, and *merge*, the modifications on the compaction procedure, the read request processes including both GET and SCAN operations, and some necessary statistical functions.

A. Experimental Setup

The experiments were performed in a Linux Ubuntu 14.04.1 environment equipped with an enterprise-level 800GB Memblaze Q520 PCIe Solid State Drive (SSD). The widely used YCSB benchmark suite [18] is adopted in our experiments to provide workloads for LevelDB, with the uniform key distribution as the default setting. Each key-value pair is set to have a 16-B key and a 1-KB value. The evaluated workloads are constructed with different ratios of random insertions (i.e., write operations) mixed with random reads (i.e., point lookups) or range scans.

The workloads we listed in Table III include a write-only (*WO*) workload, a read-only (*RO*) workload, mixed workloads consisting of 70%, 50%, or 30% of writes and GET operations representing write-heavy (*WH*), read/write balanced (*RWB*), and read-heavy (*RH*). And there are similar workloads mixing writes with range queries (i.e., *SCN-WH*, *SCN-RWB*, and *SCN-RH*). Because the lazy compaction schemes introduce much larger tail latency, which does not suit online applications, we do not put them in the comparison target.

TABLE III
YCSB WORKLOADS USED IN EVALUATIONS.

Workload	Query Type	Category
WO	/	Write Only (100% writes)
WH	Point Lookups	Write Heavy (70% writes)
RWB	Point Lookups	Read/Write Balanced (50% writes)
RH	Point Lookups	Read Heavy (30% writes)
RO	Point Lookups	Read Only (100% reads)
SCN-WH	Range Queries	Write Heavy (70% writes)
SCN-RWB	Range Queries	Read/Write Balanced (50% writes)
SCN-RH	Range Queries	Read Heavy (30% writes)

B. Latency Reduction

Tail Latency. Fig. 8 plots the relative tail latencies of different percentiles (i.e., P90 $\sim$ P99.99) for UDC and LDC by performing 10 millions of random writes and 10 millions of random reads. Recall (3) and the tail latency analysis in Section III-C that LDC can effectively reduce the count of long latencies by breaking a large compaction job into efficient small pieces. For example, for the 99.9th percentile latency, LDC reduces the latency from 469.66 us to 179.53 us, declined

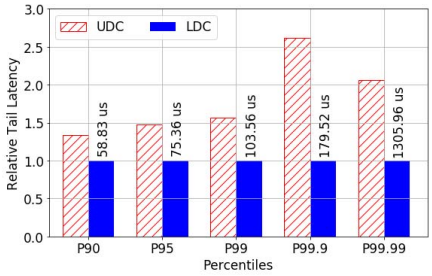


Fig. 8. The P90 $\sim$ P99.99 tail latency comparisons between UDC and LDC.

for 2.62 times. And the 99.99th percentile latency is declined from 2688.23 us to 1305.96 us by deploying LDC.

Average Latency. The same reason can explain the improvement of LDC on average latency. Fig.9 plots the average latency of UDC and LDC by running different workloads. For write-heavy workloads and read/write balanced workloads, the average latency drops to 43.3% and 45.6% from UDC to LDC. For read-heavy workloads, UDC and LDC achieve comparable average latency.

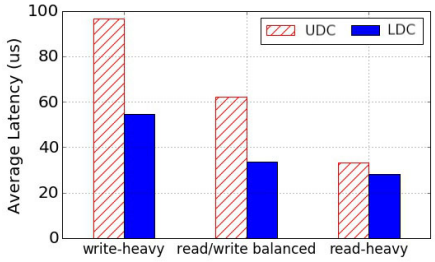


Fig. 9. The average latency of UDC and LDC by running different workloads.

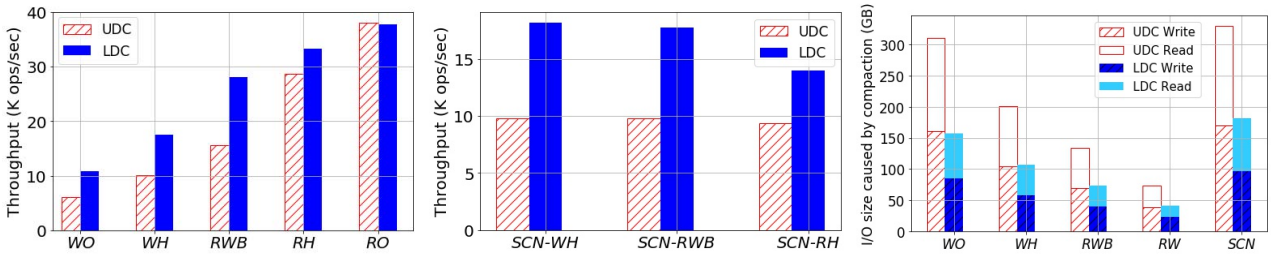
Therefore, LDC is effective in reducing tail latency and can achieve lower average latency under various workloads.

C. Throughput Improvement

Fig. 10 (a) and (b) plot the total throughput of UDC and LDC by running workloads containing queries composed of GET or SCAN, respectively. All the workloads include 10 millions of requests each under the uniform distribution.

***WO, WH, RWB, and RH* workloads**. In Fig. 10 (a), LDC achieves a 78.0% higher total throughput than UDC under the write-only workload (i.e., *WO*). For the write-heavy workloads (i.e., *WH*), LDC achieves a 73.7% higher throughput than UDC; for the read/write balanced cases (i.e., *RWB*), the improvement is 80.2%. As (2) indicates, when LDC balances the read and the write performance for SSD-based KV stores, the total throughput will be improved significantly. For the read-heavy workloads (i.e., *RH*), the improvement of LDC over UDC is 16% because write operations caused by compactions are not the majority of I/O operations. The average improvement of LDC over the traditional compaction manner can reach 56.7% on average for the above workloads including *WH*, *RWB*, and *RH*.

Read-only workloads. For the read-only workload, the throughput of LDC is nearly the same as UDC. Although LDC may trigger more read operations in order to reduce the tail latency and to promote the throughput at the same time, the SSTable-level Bloom filters and the self-adaption



(a) Throughput (Point Lookups) (b) Throughput (Range Queries) (c) I/O amount caused by compactions

Fig. 10. **Throughput Improvement**: our proposed Lower-level Driven Compaction (LDC) vs. traditional Upper-level Driven Compaction (UDC).

of the slicelink threshold make LDC achieve similar read performance as UDC. First, due to small space overhead, most Bloom filters of popular SSTables are usually in memory, avoiding many unnecessary I/Os of reading from slices. Second, when the workloads are read-dominated or read-only, the SliceLink threshold of LDC will be self-tuned to be very small, reducing the linked slice count for SSTables.

Range Queries. *SCN* workloads are composed of random insert operations and SCAN operations, each of which covers 100 key-value pairs on average. The throughput of running *SCN-WH*, *SCN-RWB*, and *SCN-RH* workloads with 10 millions of requests each are illustrated in Fig. 10 (b). LDC can promote the throughput for range queries. The throughput improvement are 86.2%, 81.1%, and 49.1%, respectively for *SCN-WH*, *SCN-RWB*, and *SCN-RH*, compared with UDC. The average improvement of LDC is up to 72.3%.

Recall Theorems 3.1 that LDC can reduce the write amplification rate of an LSM-tree. As the experimental results indicate, LDC is effective in promoting the system throughput of KV stores due to smaller write amplification rates. Note that the throughput of range-query workloads measured in ops/sec is usually lower than those of the workloads without range queries, because one range query that reads 100 key-value pairs on average only accounts for one operation in the measurement of throughput.

D. Compaction Efficiency Analysis

Fig. 10 (c) exhibits the total I/O amount caused by the compaction operations of LDC and UDC under different workloads. *SCN* in this figure indicates *SCN-RWB*. By adopting the lower-level driven compaction, the key-value store can save nearly half of the I/O requests during the compaction procedure under all kinds of workloads. Taking the *WH* workload for example, the read and write I/O sizes of UDC are 98.78 and 107.1GB, respectively, almost twice of LDC's, which are 50.38 and 58.78 GB.

Moreover, for key-value stores coupled with flash-based SSDs, which have limited write endurance, LDC can extend the SSD lifetimes by reducing writes caused by compactions, improving the system reliability and reducing the costs.

E. Uniform vs. Non-Uniform Distributions

In this part, we measure the system throughput under the uniform and the non-uniform distribution workloads, as shown in Fig. 11. Some *RWB* workloads of 20 million requests under the uniform and the Zipf distributions were performed in the experiments and the Zipf constant ranges from 1 to 5 (i.e., *Zipf1*, *Zipf2*, and *Zipf5* in Fig. 11). The larger the Zipf constant is, the accesses are more concentrated on some popular key-value pairs.

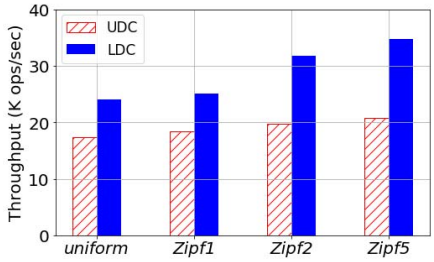


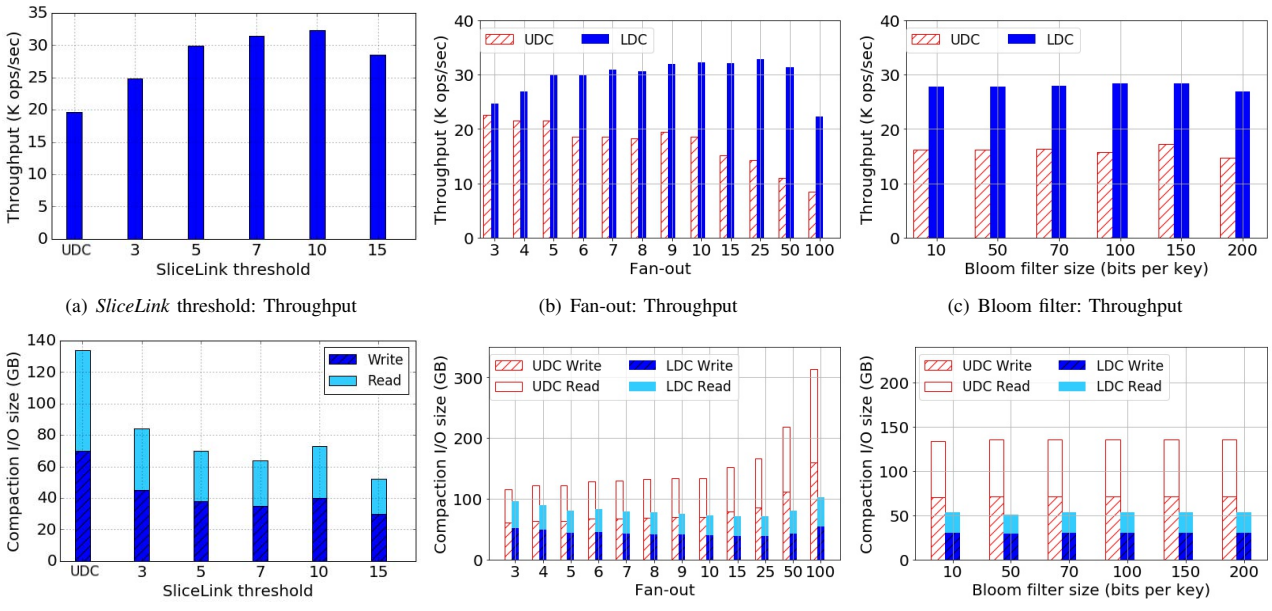
Fig. 11. The throughput under workloads of uniform and Zipf distributions.

Compared with the uniform distribution, Zipf distribution usually leads to higher hit ratios of in-memory cache. Moreover, the concentrated user accesses push more data in limited key ranges down for compaction, leading to smaller write amplification rates. So both UDC and LDC achieve higher performance when the Zipf constant gets larger.

Along with the increase of the Zipf constant, the performance promotion of LDC over UDC generally gets larger. For example, the performance improvement of the uniform distribution is 38.7%, while that of *Zipf5* is up to 67.3%. When the written data are more concentrated in part of key ranges under Zipf distribution with larger Zipf constant, it is much easier to reach the *SliceLink* threshold for the lower-level SSTables according to LDC. Therefore, LDC is more effective under the Zipf distribution, which is more close to the practical applications.

F. Impacts of SliceLink Threshold Settings

In this part, we measure the impacts of the *SliceLink* threshold of LDC under a uniform workload with 50% writes and 50% reads, illustrated in Fig. 12 (a) and (d). The *SliceLink* threshold determines the timing to trigger a *merge* phase. The results exhibit that the most suitable setting of the *SliceLink* threshold is the same as fan-out (e.g., 10 in this case). A small value will force the target SSTable to start merging too early with few linked slices, i.e., the percentage of the additional lower-level compaction I/Os will be enlarged. On the contrary, although a large *SliceLink* threshold generates smaller I/O amplification (see Fig. 12 (d)), it will introduce more data fragments, leading to loss of I/O performance.



(a) *SliceLink* threshold: Throughput (b) Fan-out: Throughput (c) Bloom filter: Throughput

(d) *SliceLink* threshold: Compaction I/O Size (e) Fan-out: Compaction I/O Size (f) Bloom filter: Compaction I/O Size

Fig. 12. Impacts of different *SliceLink* thresholds, fan-outs, and Bloom filter sizes.

G. Impacts of Fan-out Settings

Fig. 12 (b) and (e) respectively plot the throughput and the total compaction I/O sizes when the fan-out of the LSM-tree ranges from 3 to 100 under a uniform *RWB* workload. LDC achieves less compaction I/Os and higher performance than UDC in all the cases. The performance is improved by 8.8% $\sim$ 187.9%. The larger a fan-out value is, the greater advantage of LDC over UDC is. The reason lies in that LDC is designed to reduce the I/Os of each round compaction, which is extremely effective for a relatively fat LSM-tree.

For UDC, the fan-out value that achieves the best performance is 3, and the best fan-out for LDC is about 25. The hint for us is that we can adopt an appropriately fat LSM-tree to achieve high performance for LDC. The highest performance that LDC has even reached is still much better than UDC. In the other experiments, the fan-out values of UDC and LDC are both set to 10 by default.

H. Impacts of Bloom Filter Settings

***RWB* workload**. In Fig. 12 (c) and (f), we evaluate how different Bloom filter size affects the performance based on a read/write balanced workload under the uniform distribution. In order to reduce the I/O overhead of reading data, LSM-tree KV stores usually set a Bloom filter in each SSTable. If a key fails to pass the Bloom filter, it is surely not in this SSTable; if it hits the Bloom filter, the key may be in this SSTable. Therefore, the accuracy of Bloom filters is related to their sizes. As shown in Fig. 12 (c) and (f), when the sizes of Bloom filters are assigned to 10 to 200 bits per key, the system performance does not fluctuate much for both UDC and LDC, indicating that Bloom filters in the level of 10 bits/key are enough to provide high accuracy to judge whether a target key is contained in an SSTable.

***RO* workload**. Because of the design of *SliceLinks*, LDC may need to check more SSTables than UDC when performing a read request. However, the Bloom filters can exclude the majority of to-be-checked SSTables for a target key to reduce actual I/Os, especially when the accuracy of Bloom filters is high. The effects of Bloom filters with different sizes are further evaluated and shown in Fig. 13 under a 10-million read-only workload. This figure shows the relationship between the size of Bloom filters and their effects on reducing the count of reading data blocks in SSTables from SSDs. Because we always need to check a Bloom filter first when processing a read requests, if the accuracy of the Bloom filter is 100%, the read counts of data blocks should be exactly 10 millions. Fig. 13 exhibits that when the bits-per-key of Bloom filters reaches 16 or higher, the effects of reducing read counts of data blocks is not obvious, i.e., increasing the Bloom filter size is hard to promote the accuracy of Bloom filters further. So a value of bits-per-key between 8 and 16 is usually enough.

In addition, the relationship between the Bloom filter size of each SSTable and the setting of bits-per-key is also illustrated in Fig. 13. When the value of bits-per-key ranges from 8 to 128, the size of Bloom filter increases from 11.3 to 67.3 KB. The above analysis tells us that we should adopt Bloom filters of 8 $\sim$ 16 bits/key, i.e., the Bloom filter in each 2-MB SSTable is only a bit more than 10 KB. The 0.5% additional space overhead is not a heavy burden and most of the Bloom filters can be cached in memory to promote the performance significantly.

I. Scalability

Fig. 14 shows the comparison between LDC and UDC when conducting different amount of requests with 50% writes and 50% reads under the uniform distribution. As the request count

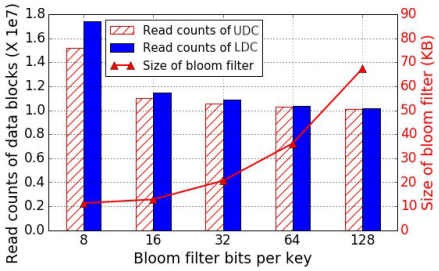


Fig. 13. Impacts of Bloom filter size.

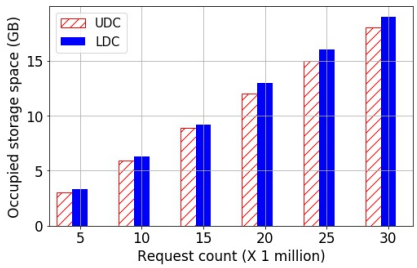


Fig. 14. Impacts of different data scale.

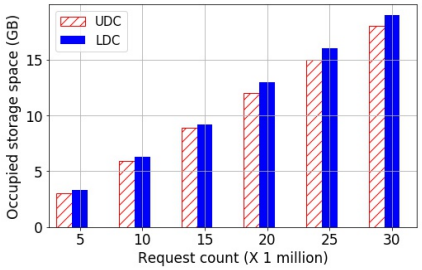


Fig. 15. LDC does not bring much space overhead.

ranges from 5 to 30 millions, LDC always maintains a 39% $\sim$ 65% higher throughput and saves 43.3% $\sim$ 46.7% of the compaction I/O consumptions than the mechanism, indicating good scalability of LDC.

J. Space Efficiency

To avoid frequent I/O operations, LDC adopts a delayed garbage collection mechanism for recycling the useless *slices* in frozen SSTables. This may lead to more space consumption in SSDs. Fig. 15 plots the final consumed storage space in the KV store to perform the uniform *RWB* workloads with total request counts ranging from 5 to 30 millions. The results reveal that LDC only consumes 3.37% $\sim$ 10.0% more space compared with UDC. Hence, the additional capacity of LDC is not a heavy burden for modern storage systems.

V. Related Work

In this part, we summarize existing research work related to the LSM-tree optimization and SSD-based key-value stores in three categories:

Lazy Compaction Schemes. Existing work about the LSM-tree compaction mechanism improvement mainly aims to reduce the write amplification of compactions through accumulating more data in each round of compaction, i.e., increasing the compaction granularity. Except for the previously introduced RocksDB [15] and dCompaction [16], there are also some other approaches:

The approach of the LSM-Tree maintenance in Cassandra is called size-tiered compaction [20]. The size of the SSTable is divided into many tiered of exponential growth. Several smaller SSTables would be compacted into a larger SSTable. For example, if the fan-out is 4, four SSTables of 16-MB data would be compacted into a new SSTable of 64MB. And if the amount of the SSTables in 64-MB size reach 4, these four SSTables would be compacted into an SSTable of 256MB.

PebblesDB [17] proposed a new structure called *Guard* which contains multiple SSTables with overlapping key ranges in each layer of an LSM-tree. It also extends the compaction action by tolerating some unsorted data in a layer.

Although these lazy compaction schemes can lower the write amplification rates, the enlarged compaction granularity usually leads to serious performance fluctuation and large tail latency. In addition, the large compaction granularity requires a huge space cost including both the input and output data of the compaction behavior. For example, for the universal compaction of RocksDB, under an extreme circumstance that all the SSTables are involved in the compaction, the KV store has to prepare twice the storage capacity of all the stored data.

Moreover, the existing approaches are orthogonal with LDC. Some existing lazy solutions skip the compaction operations at some levels of the LSM-tree, while some other methods put several sorted data sets in one level of the LSM-tree. Different with all of them, LDC optimizes the microscopic action in each round of compaction. Therefore, LDC can be integrated with them to further improve the throughput and tail latency. Besides LSM-tree, LDC can also be applied in some other similar data structures. For example, in the partitioned B-tree [21], some independent B-tree partitions are allowed except for the main partition. In this case, the write performance, especially the bulk writes, are significantly improved, with the cost of declined read performance due to looking up multiple sorted partitions. When the data in the small partitions are merged into the main partition, LDC can be integrated to both shrink the granularity of data merging for smaller tail latency and accumulate more data in small partitions for less write amplification and higher throughput.

Other LSM-tree improvement except for compaction mechanism. Some works accelerate compactions by making use of hardware parallelism such as CPUs and I/O devices. RocksDB [15] supports configuring an arbitrary number of *MemTables*, and can be configured to issue concurrent compaction requests from multiple threads. PCP [22] also decomposes compactions into several sub-tasks, and exploits the parallelism of I/O and CPU resources to process them in a pipeline manner. LOCS [23] extends LevelDB to exploit the parallelism of customized open-channel SSDs and optimizes the scheduling and dispatching policies to improve efficiency.

Some other works reduce the compaction I/O amplification effectively for some types of workloads. Atlas [24] and WiscKey [25] separate keys and values, using an LSM-tree to manage keys and the pointers to their values and appending values into logs, which is effective on large objects since compactions only need to concern I/O amplification of keys. LSM-trie [26] uses a tree prefixed by hash values of the keys to organize data in the store and replace merge sorting with hash sorting in compaction, which reduces the compaction I/O amplification effectively, but sacrifices the important range scan performance of LSM-trees. Monkey [27] focuses on the LSM-tree parameters optimization for performance acceleration. Light-weight compaction [28] first compacts some

sorted SSTables into temporary DTables which possess several overlapped segments and shared metadata, and then merge some DTables into sorted SSTables. This approach introduces extra I/Os performed on DTables, without effective reduction on compaction I/O overhead.

SSD-based key-value stores. Flash-based SSDs provide higher performance than hard disk drives and have been deployed in many enterprise storage systems. However, SSDs are faced with the write durability problem, i.e. the device will become unreliable after a certain times of writing [13]. For the flash-based KV stores, many systems organize key-value pairs in log-structures and establish hash tables for quick lookup, e.g., FAWN [29] and Flashstore [30], which require large memory to keep the in-memory indexes. However, stores based on hash indexes are difficult to provide satisfactory range query performance.

VI. Conclusion

For the widely used LSM-tree key-value storage engines in many NoSQL and SQL systems, the batched write operations (i.e., compaction) can promote the system throughput, but also introduce the problems of performance fluctuation and long tail latency. Some existing lazy compaction schemes can reduce the count of performed compaction operations to promote system throughput, but they also enlarge the compaction granularity and worsen the tail latency. In fact, the difficulty of improving both the throughput and the tail latency of LSM-tree key-value stores lies in the traditional upper-level driven compaction manner, which adds much more lower-level data into an I/O batch of compaction when merging the upper-level data down, enlarging the compaction granularity and the I/O amplification.

Therefore, in this paper, we propose a novel Lower-level Driven Compaction (LDC) method. By means of separating an consecutive compaction action into two steps, i.e., *link* and *merge*, LDC performs actual I/Os driven by a lower-level data set that has enough links with upper-level data. Therefore, LDC can effectively reduce the additional I/O amplification and shrink the compaction granularity, achieving improved tail latency and system throughput at the same time.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), National Natural Science Foundation of China (No. 61732014, 61472427, and 61572353), Beijing Natural Science Foundation (No. 4172031), Natural Science Foundation of Tianjin (17JCYBJC15400), and Open research program of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Science (No. CARCH201702). This work is also supported by SenseTime Young Scholars Research Fund.

References

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: a distributed storage system for structured data. *TOCS*, 26(2):15–15, 2008.

[2] Avinash Lakshman and Prashant Malik. Cassandra:a decentralized structured storage system. *Acm Sigops Operating Systems Review*, 44(2):35–40, 2010.

[3] Apache hbase, 2017. http://hbase.apache.org/.

[4] Hawq, 2018. http://hawq.apache.org/.

[5] Tidb is a distributed htap database compatible with the mysql protocol, 2018. https://github.com/pingcap/tidb.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[7] Rifki Sadikin, Andria Arisal, Rofithah Omar, and Nur Hidayah Mazni. Processing next generation sequencing data in map-reduce framework using hadoop-bam in a computer cluster. In *ICITISEE*, pages 421–425. IEEE, 2017.

[8] Belén Vela, José María Cavero, Paloma Cáceres, Almudena Sierra-Alonso, and Carlos E Cuesta. Using a nosql graph oriented database to store accessible transport routes. In *EDBT/ICDT Workshops*, pages 62–66, 2018.

[9] Ashish Kumar Gupta, Prashant Varshney, Abhishek Kumar, Bakshi Rohit Prasad, and Sonali Agarwal. Evaluation of mapreduce-based distributed parallel machine learning algorithms. In *Advances in Big Data and Cloud Computing*, pages 101–111. Springer, 2018.

[10] Innodb, 2018. https://en.wikipedia.org/wiki/InnoDB.

[11] Mysql, 2018. https://www.mysql.com/cn/.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.

[13] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *FAST*, pages 9–9, 2010.

[14] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of nand flash memory. In *FAST*, pages 2–2. USENIX Association, 2012.

[15] Under the hood: Building and open-sourcing rocksdb, 2017. http://goo.gl/9xulVB.

[16] Feng-Feng Pan, Yin-Liang Yue, and Jin Xiong. dcompaction: Speeding up compaction of the lsm-tree via delayed compaction. *JCST*, 32(1):41–54, 2017.

[17] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*, pages 497–514. ACM, 2017.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, pages 143–154, 2010.

[19] Leveldb - a fast and lightweight key/value database library by google, 2017. http://code.google.com/p/leveldb.

[20] Bradley C Kuszmaul. A comparison of fractal trees to log-structured merge (lsm) trees. *White Paper*, 2014.

[21] Goetz Graefe. Sorting and indexing with partitioned b-trees. In *CIDR*, volume 3, pages 5–8, 2003.

[22] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. Pipelined compaction for the lsm-tree. In *IPDPS*, pages 777–786, 2014.

[23] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *EuroSys*, page 16. ACM, 2014.

[24] Chunbo Lai, Song Jiang, Liqiong Yang, and Shiding Lin. Atlas: Baidu's key-value storage system for cloud data. In *MSST*, pages 1–14, 2015.

[25] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *FAST*, pages 133–148, 2016.

[26] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. In *ATC*, pages 71–82, 2015.

[27] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD*, pages 79–94. ACM, 2017.

[28] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *MSST*, 2017.

[29] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. *CACM*, 54(7):101–109, 2011.

[30] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *VLDB*, 3(1-2):1414–1425, 2010.