Pan FF, Yue YL, Xiong J. dCompaction: Speeding up compaction of the LSM-tree via delayed compaction. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(1): 41–54 Jan. 2017. DOI 10.1007/s11390-017-1704-4

dCompaction: Speeding up Compaction of the LSM-Tree via Delayed Compaction

Feng-Feng Pan[1,2], *Student Member, CCF, ACM, IEEE*, Yin-Liang Yue[3], *Member, CCF, ACM, IEEE* and Jin Xiong[1,2], *Senior Member, CCF, Member, ACM, IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*

E-mail: panfengfeng@ict.ac.cn; yueyinliang@iie.ac.cn; xiongjin@ict.ac.cn

Received August 1, 2016; revised November 9, 2016.

Abstract Key-value (KV) stores have become a backbone of large-scale applications in today's data centers. Write-optimized data structures like the Log-Structured Merge-tree (LSM-tree) and their variants are widely used in KV storage systems like BigTable and RocksDB. Conventional LSM-tree organizes KV items into multiple, successively larger components, and uses compaction to push KV items from one smaller component to another adjacent larger component until the KV items reach the largest component. Unfortunately, current compaction scheme incurs significant write amplification due to repeated KV item reads and writes, and then results in poor throughput. We propose a new compaction scheme, delayed compaction (dCompaction) that decreases write amplification. dCompaction postpones some compactions and gathers them into the following compaction. In this way, it avoids KV item reads and writes during compaction, and consequently improves the throughput of LSM-tree based KV stores. We implement dCompaction on RocksDB, and conduct extensive experiments. Validation using YCSB framework shows that compared with RocksDB, dCompaction has about 40% write performance improvements and also comparable read performance.

Keywords key-value store, Log-Structured Merge-tree (LSM-tree), write amplification, delayed compaction

1 Introduction

Key-value (KV) store plays a vital role in today's large-scale, high-performance, data-intensive applications in data centers. Handling write-dominated interactive workloads is becoming increasingly important for KV stores. On the one hand, typical KV store workloads have transitioned from 80%∼90% reads in 2010 to only 50% reads in 2012[1], so that the ratio of writes is increasing in comparison with that of reads in typical low-latency workloads. On the other hand, writes are more likely to be the performance bottleneck of back-end storage systems, since most read requests can be absorbed by multi-level caches[2], implemented by Web browser's cache, CDN, Redis①, Memcached[3] and OS page cache, while data must be written to persistent storage devices to ensure data persistence. The analysis results of photo caching mechanism in Facebook[2] show that 90.1% of read requests are served by multi-layer cache and only 9.9% of read requests are served by back-end storage.

Since caches in the entire system absorb most of read requests, back-end storage tends to be write-dominated. Therefore, write-optimized data structures like the Log-Structured Merge-tree (LSM-tree)[4] and their variants have attracted much attention and have widely been used in KV storage systems, including distributed KV stores, such as BigTable[5], Cassandra[6],

Regular Paper

A preliminary version of the paper was published in the Proceedings of NPC 2016.

This work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000202 and the National Natural Science Foundation of China under Grant Nos. 61303056 and 61379042.

① Redis. http://redis.io, July 2016.

HBase[7], HyperDex[8], Pnuts[9], and local KV stores like LevelDB[2], RocksDB[3].

The LSM-tree organizes data into multiple, successively larger components (or levels) named C_0, C_1, ..., C_k. The first component C_0 is memory-resident and the others are disk-resident. When one component is filled up, its KV items are pushed to the adjacent larger component via a procedure called compaction. The compaction procedure is extremely I/O-intensive. Briefly, if $M = (\frac{Size\,(C_{i+1})}{Size\,(C_i)})$($Size$() means the size of component C_i), in order to push B bytes of data from component C_i to component C_{i+1}, a compaction performs both $(M{+}1)\times B$ bytes read I/O and $(M{+}1)\times B$ bytes write I/O. Both the basic data structures and the compaction procedure of LSM-tree are described in Section 2. Obviously, serious write amplification has a large impact on the throughput achievable using LSM-tree.

Some research has been done to improve the performance of LSM-tree based KV stores. VT-tree[10], bLSM[1], PE[11], and PCP[12] are proposed to improve the write performance. VT-tree uses the stitching technique to avoid unnecessary data movement. The replacement-selection sort algorithm is adopted by bLSM to decrease the compaction frequency. PE and bLSM partition the key range to confine compactions in hot key ranges. PCP uses a pipelined compaction procedure to fully utilize both CPUs and I/O devices, in order to speed up the compaction procedure. All the research focuses on decreasing the compaction frequency, accelerating the speed of compaction or confining compactions on hot data key-ranges. However, they have not considered repeated KV item reads and writes caused by KV items component-by-component movement from smaller components to larger components, which leads to serious write amplification. In Section 2, we show that the write amplification can be over 30 times.

We present a new compaction scheme called delayed compaction (dCompaction) for reducing the amount of I/O involved in maintaining an LSM-tree, which is widely used in KV storage systems. dCompaction replaces some real compactions which generate large amounts of I/O by writing only meta-data in the form of virtual SSTables with each virtual SSTable pointing to the unsorted real SSTables, which is called virtual compaction. Since several virtual compactions can be combined into one real compaction, the total amount of I/O can be reduced. The trade-off is that reads are slower if they involve virtually compacted blocks. In order to overcome these challenges, we introduce two parameters called *VCT* and *VSMT* into dCompaction.

More concretely, our contributions are as follows.

- We make a deep experimental and qualitative analysis of write amplification in LSM-tree based KV stores under write-intensive workloads.
- We propose a new compaction scheme called dCompaction that improves write performance.
- We introduce two parameters called *VCT* and *VSMT*, which ensure the trade-off between write and read performance.
- We implement dCompaction on RocksDB, and the experiment shows that dCompaction has about 40% write performance improvements and provides reasonable read performance compared with RocksDB.

The rest of this paper is organized as follows. The background and the motivation are presented in Section 2. In Section 3, we discuss details of the design of dCompaction. The performance evaluation is described in Section 4. We present related work in Section 5 and conclude this paper in Section 6 by summarizing our main contributions.

2 Background and Motivation

We firstly present the background of LSM-tree, and then detail the performance issues of LSM-tree under write-intensive workloads with the experiments on RocksDB which motivates the design of delayed compaction.

2.1 LSM-Tree

LSM-tree is a disk-based data structure designed to provide low-cost indexing for the workloads experiencing a high rate of record updates (inserts and deletes) over an extended period. LSM-tree consists of multiple components C_i. The C_0 component is memory-resident and the remaining components reside on disk, as shown in Fig.1(a).

We use a representative design named 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 hold recently inserted and sorted KV items. When an MemTable exceeds its capacity threshold, it will be dumped into disk as an SSTable, such as T_{11} which refers to an SSTable in Fig.1(a). Each disk component is composed of many

[2] LevelDB. http://code.google.com/p/LevelDB, May 2014.

[3] RocksDB. http://rocksdb.org, June 2016.

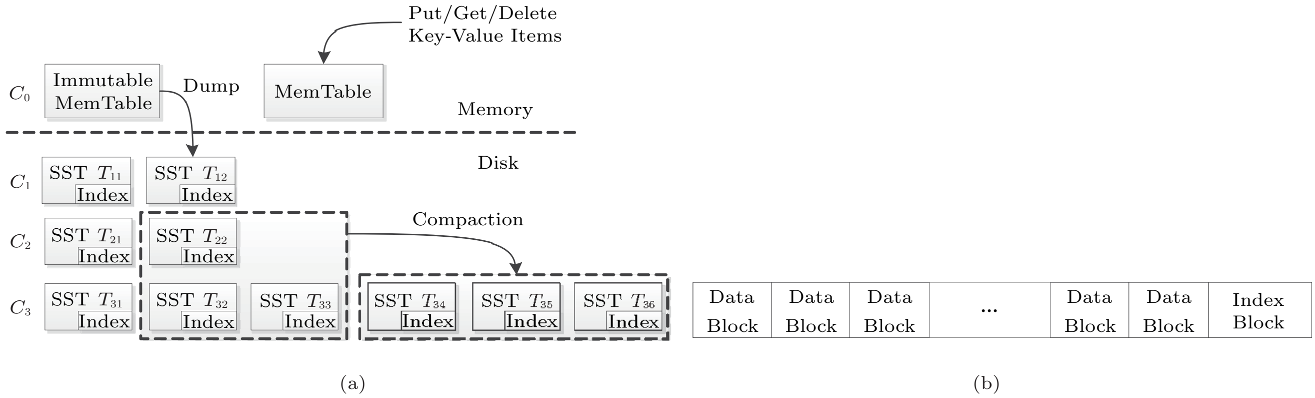


Fig.1. (a) Basic LSM-tree data structure and (b) SSTable layout.

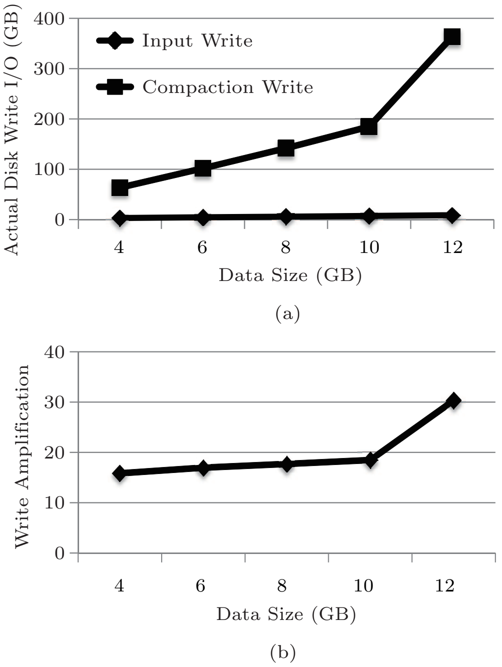
SSTables, whose key ranges do not overlap with each other except those in C_1. Fig.1(b) presents the layout of one SSTable. Each SSTable contains multiple data blocks and one index block. The data block contains the sorted KV items, while the index block contains both indexes and bloom filters of each data block.

2.2 Compaction Procedure and Write Amplification in LSM-Tree

LSM-tree defers and batches index updates, cascading the changes from a memory-based component through one or more disk components. For one KV item, it is firstly inserted into the in-memory buffer component C_0, then dumped into C_1 and pushed into C_2, C_3, ..., C_k in sequence during the compaction procedure. KV items of both C_i and C_{i+1} within the same key range are firstly read into memory, then merge-sorted, and finally written back to C_{i+1} in the fix-sized SSTables during the compaction procedure. For example, as shown in Fig.1(a), KV items of T_{22} in C_2 have the same key range with those of T_{32} and T_{33} in C_3, and they are involved in one compaction procedure. The compacted KV items are sequentially stored into T_{34}, T_{35} and T_{36}, which are located in C_3.

Unfortunately, the compaction procedure is extremely I/O-intensive. Assume that each SSTable is sized to V, and in LSM-tree, C_{i+1} has around M times the total size of C_i, i.e., $M = \frac{Size\,(C_{i+1})}{Size\,(C_i)}$ (where $i = 0$, 1,...). M is named as amplification factor (AF), which is 10 in RocksDB by default. We define the write amplification ratio (WAR) as the proportion between the actual write amount to the disk and the amount of data written by users. The number of SSTables in C_{i+1} that contain keys overlapping with the keys in SSTables in C_i is also close to M. This means moving one SSTable from C_i to C_{i+1} during a compaction requires $(M+1)\times V$ bytes to be read and $(M+1)\times V$ bytes to be written to disk; thus we can say that the write amplification factor is 11, i.e., $WAR = M + 1$. Recall that for a KV item flowing from C_0 to C_k during the compaction procedure, the WAR can reach up to $K\times(M + 1)$[13-14]. We conduct experiments on RocksDB to measure the write amplification of LSM-tree based KV stores by using 100% write workload generated from Yahoo! Cloud Serving Benchmark (YCSB)[15] with 256 B value size, random key and Zipfian distribution. We execute the run phase with data size of 4 GB, 6 GB, 8 GB, 10 GB and 12 GB separately. From Fig.2(a) we can see that when the input data size is 4 GB (in fact 2.96 GB after compression), the actual disk write I/O is 63.39 GB. In this case, the write amplification is 15.85. From Fig.2(b) we can see that WAR is increasing up to 30.30 with an average of 19.86. In addition, the write amplification becomes more severe with the increase of data size.

The serious write amplification makes compaction I/O dominate the disk I/O. Without considering scan operation, the disk I/O of LSM-tree based KV stores includes 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 get the specific KV items from disk-resident SSTables. Compaction I/O includes reading KV items from disk-resident SSTables and writing KV items that were sorted and merged to SSTables. We conduct experiments on RocksDB to explore the proportion of the three types of I/O by using a dataset generated from YCSB with 1 KB value size, random key and Zipfian distribution. In this experiment, both update and get operations are incorporated and we vary the get proportion from 0% to 90%. After loading 2 GB data, we execute the run phase with 2 000 000 operations, including update and get operations.



(a)

(b)

Fig.2. (a) Comparison between input data volume and actual disk write I/O and (b) write amplification of RocksDB.

From Fig.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%. Thus we can conclude that serious write amplification greatly consumes most of the disk I/O bandwidth, and leaves little for servicing the fronted application requests.

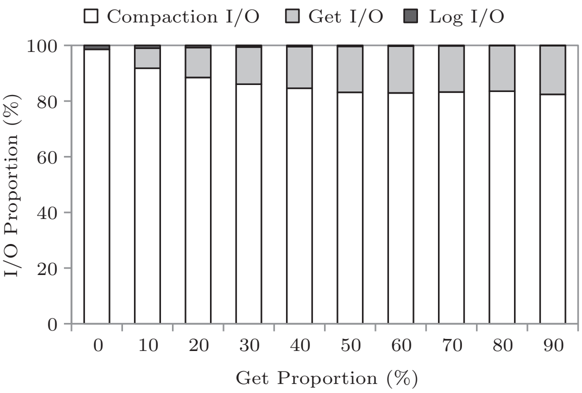


Fig.3. Comparison of compaction I/O, get I/O and log I/O.

2.3 Motivation

For some KV items in LSM-tree to reach C_k, it would be involved in the compaction between C_0 and C_1 to reach C_1 first, and then involved in the compaction between C_1 and C_2 to reach C_2, and so on. Finally, it would be involved in the compaction between C_{k-1} and C_k to arrive at the destination component C_k, which incurs repeated reads and writes for these KV items during the compaction procedure, and results in superfluous I/O overheads.

Fig.4 shows two conventional compaction procedures. SSTables T_{34}, T_{35} and T_{36} are generated by one compaction of T_{22}, T_{32} and T_{33}, and then SSTables T_{44}, T_{45}, T_{46}, T_{47}, T_{48} are generated by another compaction of T_{34}, T_{35}, T_{36}, T_{42} and T_{43}. During the compaction procedure, we assume that there is no repeated KV items between SSTables, which means that the KV items in SSTables are just merge-sorted by the compaction (in the real environment, there always exist repeated KV items, thereby the compaction will delete redundant KV items and merge-sort the remaining items), so that we can get two relationships between compaction read and write I/O as (1) and (2):

$$W(T_{34}+T_{35}+T_{36}) = R(T_{22}+T_{32}+T_{33}), \quad (1)$$

$$\begin{aligned} & W(T_{44}+T_{45}+T_{46}+T_{47}+T_{48}) \\ = \ & R(T_{34}+T_{35}+T_{36}+T_{42}+T_{43}), \quad (2) \end{aligned}$$

where $W()$ and $R()$ are two functions. $W()$ means the total size of write I/O during compaction procedure and $R()$ means the total size of read I/O during compaction procedure. From these two equations, we can conclude two observations as follows.

1) KV items in T_{34}, T_{35} and T_{36} can be read from T_{22}, T_{32} and T_{33}. Similarly, KV items in T_{44}, T_{45}, T_{46}, T_{47} and T_{48} can also be read from T_{34}, T_{35}, T_{36}, T_{42} and T_{43}.

2) There exist repeated reads and writes during these two compaction procedures. KV items in T_{22}, T_{32} and T_{33} are read and written at least twice during the two compaction procedures. This needs to be stressed that the repeated reads and writes are also in the real environment, which is shown in Fig.2(b).

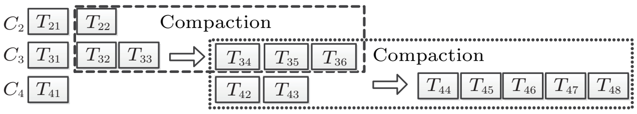


Fig.4. Conventional compaction procedures.

Motivated by the above two observations, we propose dCompaction, instead of conventional compaction scheme.

3 Design of dCompaction

This section introduces the design of dCompaction optimized for write-intensive performance. Firstly, the main idea of dCompaction is described in more detail in Subsection 3.1. Traditional dCompaction has a great effect on the read performance of KV stores, which is explained in Subsection 3.2. In order to overcome these challenges, we introduce two parameters named virtual compaction threshold (VCT) and virtual SSTable merge threshold (VSMT) to support comparable read performance while maintaining high write performance, and they are illustrated in Subsection 3.3 and Subsection 3.4 respectively.

3.1 Main Idea

As shown in Fig.4, we call the compaction real compaction, i.e., the chosen SSTables are merged into new and non-overlapped SSTables, which is displayed in Fig.5(a), and all the SSTable sizes are identical no matter what level they are in or among. We define the SSTables generated by real compaction as real SSTables, e.g., SSTables T_{34}, T_{35} and T_{36}. For each real SSTable, there exists a corresponding SSTable metadata including the smallest key, the largest key, file number, and file size. Here we define the stage of generating metadata as metadata generation. Therefore there are two stages in real compaction named data generation and metadata generation. Most of I/O overheads are caused by data generation, while metadata generation has less I/O overheads.

In contrast to real compaction, virtual compaction has less compaction I/O overheads since there is only one stage, metadata generation. Virtual SSTable's metadata is composed of the smallest key, the largest key, file number, file size and parentSSTables. The parentSSTables store the relationship between the real SSTable and the virtual SSTable. For example, the virtual SSTables VT_{34}, VT_{35} and VT_{36} are derived from real SSTables T_{22}, T_{32} and T_{33} in Fig.5(b); thus the parentSSTables of VT_{34}, VT_{35} and VT_{36} are the same, i.e., the file number of T_{22}, T_{32} and T_{33}.

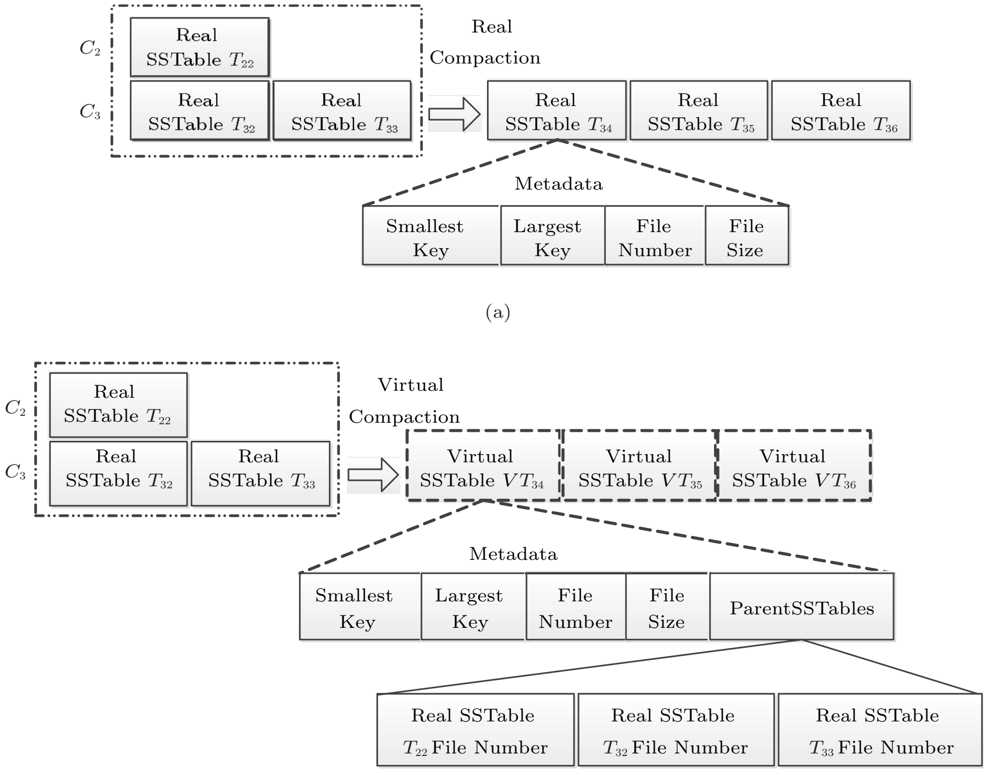


Fig.5. (a) Real compaction and real SSTable's metadata and (b) virtual compaction and virtual SSTable's metadata.

According to the observations in Subsection 2.3 and the combination of real compaction and virtual compaction, we present dCompaction instead of conventional compaction scheme. dCompaction postpones real compaction through virtual compaction to avoid repeated KV item reads and writes.

Fig.6 shows the basic process of dCompaction. T_{22}, T_{32} and T_{33} are converted to VT_{34}, VT_{35} and VT_{36} via virtual compaction and only metadata are recorded in VT_{34}, VT_{35} and VT_{36}; thus it skips compaction I/O overheads resulted from real compaction, such as compaction shown in Fig.4. In the following real compaction, it merges T_{42}, T_{43} with other SSTables which VT_{34}, VT_{35} and VT_{36} are derived from, i.e., T_{22}, T_{32} and T_{33}, into T_{44}, T_{45}, T_{46}, T_{47} and T_{48}. Compared with conventional compaction scheme, T_{22}, T_{32} and T_{33} are read and written only once in dCompaction; hence the repeated I/O overheads can be reduced, and write amplification can be greatly reduced.

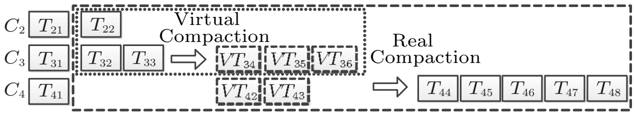


Fig.6. Delayed compaction procedure.

Also, we can use a simple mathematical model to compute the I/O bandwidth saved by dCompaction which is described in Subsection 2.2. For example, AF is set to 10, which means when an SSTable is merged into the next level, 10 SSTables in next level (in the worst case) whose entire key range matches the key range of the SSTable will be read into memory. If an SSTable T_{21} is merged into the next level (level 3), 10 SSTables, i.e., T_{31}, T_{32}, ..., T_{39} and T_{310} will be read and merged into T_{311}, T_{312}, ..., and T_{321}. When T_{311} is merged into the next level (level 4), T_{41}, T_{42}, ..., and T_{410} will be read.

Using the conventional compaction scheme, after T_{21} is merged into level 4, 132 (=(1+10)+(11+110)) SSTables are read and written. Using the proposed dCompaction, after T_{21} is merged into level 4, 121 (=1+10+110) SSTables are read and written, since T_{311}, T_{312}, ..., and T_{321} become VT_{311}, VT_{312}, ..., and VT_{321} which are not read and written. In general, $(132-121)\times 2/(132\times 2) = 8.3\%$ I/O bandwidth is saved by dCompaction from the above mathematical model.

Above all, dCompaction uses virtual compaction to reduce the compaction I/O overheads, so that the write performance can be improved. However, everything has both sides. With the advantages of virtual compaction on write performance, it also has an even greater influence on read performance, and the read process of virtual SSTable is discussed next.

3.2 Read Process

When a read operation falls in one real SSTable, it will search the target KV item in it which is shown in Fig.7(a). When a read operation falls in one virtual SSTable, since there is no data in the virtual SSTable, it will also search the target KV item in real SSTables which the virtual SSTable is derived from. For example, in Fig.7(b), we can see when one read operation falls in VT_{34}, firstly, it will determine whether the key is in the key range of VT_{34}. If so, it will read its metadata (parentSSTables) to get the real SSTables which VT_{34} is derived from, i.e., T_{22}, T_{32} and T_{33}, and then it will search the target KV item in these real SSTables in order.

Through the analysis of the read process of virtual SSTable, we can observe that the read path is increased by searching target KV item in many real SSTables; thus the number of real SSTables involved in the virtual SSTable is significantly important. In other words, more frequent virtual compaction means higher write performance but lower read performance. On the contrary, more frequent real compaction means lower write performance but higher read performance. In conclusion, the condition of triggering real compaction or virtual compaction is crucial to write and read performance, and more details about them are described as follows.

3.3 Trigger Condition: *VCT*

As mentioned above, the frequency of triggering real compaction or virtual compaction makes great influence on write and read performance. On the one hand, write performance can be speeded up by virtual compaction because of the reduction of compaction I/O overheads; on the other hand, through real compaction, the problem of the read latency of the virtual SSTable can be eased, since the read path is reduced. In order to take advantages of real compaction and virtual compaction, we introduce one parameter called VCT into dCompaction to trigger real compaction or virtual compaction.

VCT defines a threshold (the number of real SSTables during compaction) to trigger either virtual com-

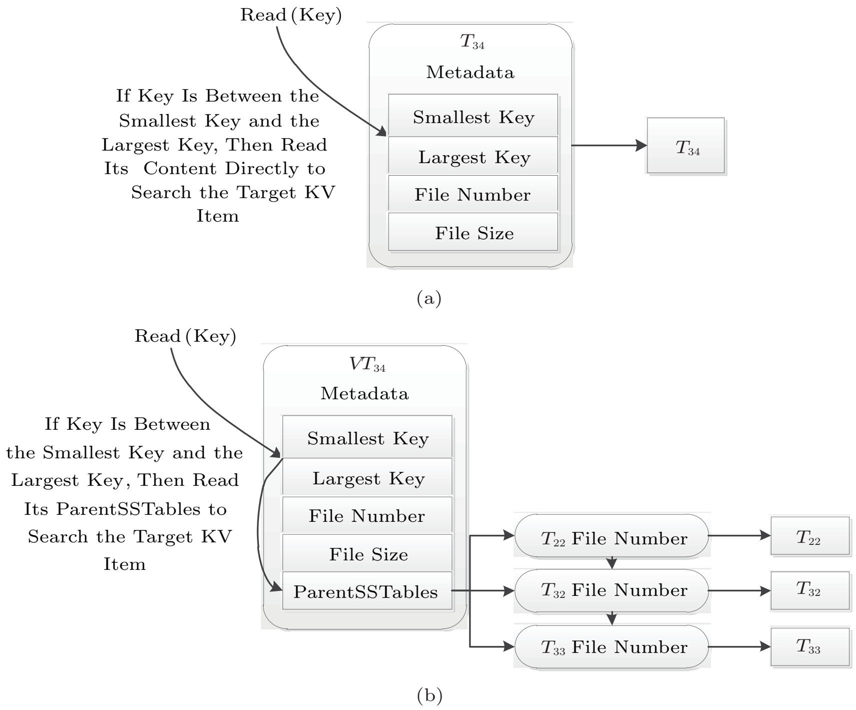


Fig.7. (a) Read process in real SSTable and (b) read process in virtual SSTable.

paction or real compaction. Firstly, we count the number of real SSTables, and then compare it with VCT to determine either virtual compaction or real compaction. The detailed algorithm is described in Algorithm 1. We can see that triggering either real compaction or virtual compaction depends on VCT, and it can make a large influence on the system performance.

Algorithm 1. Triggering Real Compaction or Virtual Compaction

```
Compaction starts;
Choose SSTables needing to be compacted;
Count the number of real SSTables in this compaction;
$Count$ = the number of real SSTables;
if $Count < VCT$ then
    Trigger the virtual compaction;
else
    Trigger the real compaction;
end
```

If VCT is set to 0, it means that there exists no virtual compaction in dCompaction which is the same with the conventional compaction scheme. Along with VCT increasing, virtual compaction is triggered more frequently than real compaction, while the total data volume of compactions decreases so that compaction I/O overheads can be reduced. However, large VCT also has negative effects on system performance.

Firstly, larger VCT will incur more virtual compaction and less real compaction; however once real compaction is triggered, larger VCT leads more real SSTables involved in this real compaction to be compacted, and it has a great impact on the foreground write/read performance. Namely, larger VCT prolongs the subsequent real compaction to lead bursty write throughput.

Secondly, larger VCT also means that the virtual SSTable contains more real SSTables; the read latency will be increased because it will search the target KV item in these real SSTables which the virtual SSTable is derived from. Thus VCT should be limited in a certain range from the perspective of read performance.

In summary, there are many important factors about the value of VCT including write amplification, real compaction time, and read performance. Through many experiments we find VCT should be set from 8 to 16 which is described in Section 4 in detail. In this range, we can get a significant write performance improvement and reasonable read performance.

3.4 *VSMT*

Although we limit VCT to ease its influence on read performance, we still make good use of bloom filter and real SSTables' key range to skip some real SSTables quickly. But when one virtual SSTable is in the critical path of read process, i.e., many read operations will meet the virtual SSTable to read its real SSTables, it has a large impact on the read performance. Thus if this virtual SSTable is merged into a real SSTable, the problem can be solved.

Based on above analysis, we introduce another parameter called *VSMT* into the read process. *VSMT* means the number of real SSTables which the virtual SSTable is derived from. During the read process, if read operation meets a virtual SSTable and the number of its real SSTables is larger than *VSMT*, thus we trigger the virtual SSTable merge, i.e., the virtual SSTable VT_{34} will be replaced by the real SSTable T_{34} as shown in Fig.8, and the following reads in this virtual SSTable will fall in the real SSTable. Here, not only the single virtual SSTable but also its adjacent virtual SSTable should be also merged into the real SSTable. For example, in Fig.8, when VT_{34} is merged into T_{34}, all the three real SSTables T_{22}, T_{32} and T_{33} should be loaded to memory which can be merged into three SSTables T_{34}, T_{35} and T_{36}. Thus the virtual SSTables VT_{35} and VT_{36} should be also merged into T_{35} and T_{36}. Otherwise, it is a waste of I/O bandwidth. The virtual SSTable merge reduces the lookup I/O, and improves the read performance. Triggering the virtual SSTable merge process is described in Algorithm 2.

Algorithm 2. Triggering the Virtual SSTable Merge Process

```
if $SSTable.type == virtualSSTable$ then
    Count the number of real SSTables in the virtual SSTable;
    $Count$ = the number of real SSTables;
    if $Count \geqslant VSMT$ then
        Merge virtual SSTable into real SSTables;
    end
end
```

In the read process, there may exist many virtual SSTables in the read path as shown in Fig.9. Along with the read path, if VT_{21} and VT_{31} are merged into real SSTables in order, it will increase read latency, so that only the first virtual SSTable that meets the above conditions can be merged.

VSMT also has positive and negative impacts on the read performance. When *VSMT* is small, triggering the virtual SSTable merge is very frequent so that it results in more merge overheads. Although virtual SSTable merge is good for the subsequent read operations, the frequent virtual SSTable merge can incur huge disk I/O, sacrifice the current read latency and reduce the dCompaction's effects on write amplification. When *VSMT* is large, less virtual SSTables are merged; thus there is no improvement in read performance. Therefore, setting the value of *VSMT* should take read latency and virtual SSTable merge overheads into consideration. Extensive experimental results show that *VSMT* should be set from 3 to 6 which is described in Section 4 in details. In this range, we can get the largest performance improvement for the read.

4 Evaluation

We integrate dCompaction into RocksDB, and compare its performance with that of original RocksDB. Also, we make a deep comparison on both solid state drives (SSDs) and hard disks.

Implementation Details. dCompaction is implemented based on RocksDB which is a representative LSM-tree implementation from Facebook. We leave out the details of RocksDB, and focus on components specific to our optimizations instead.

In total, we modify or add three components about 800 lines of C++ code in RocksDB as follows.

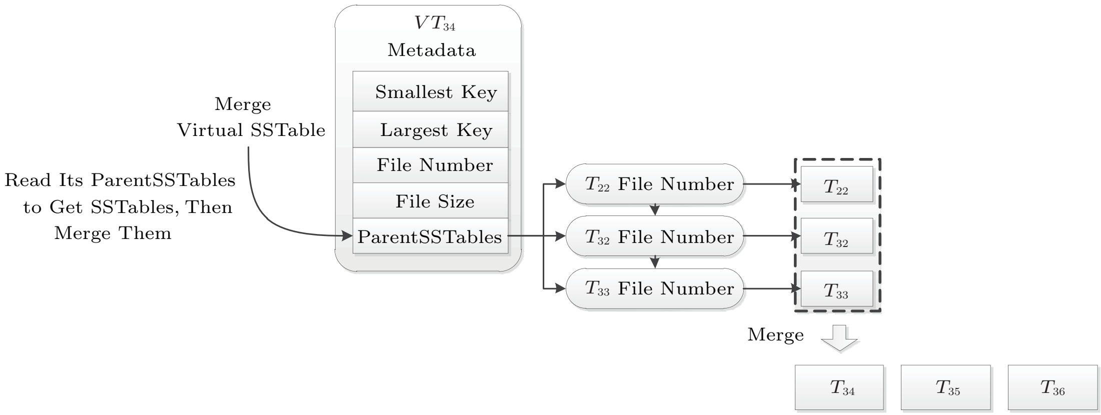


Fig.8. Merge virtual SSTable.

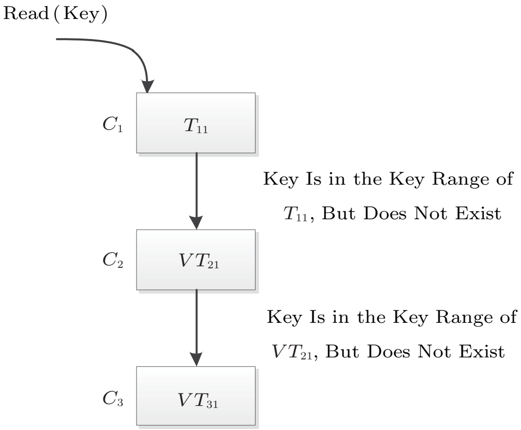

Read (Key)
C_1
T_{11}
Key Is in the Key Range of
T_{11}, But Does Not Exist
C_2
VT_{21}
Key Is in the Key Range of
VT_{21}, But Does Not Exist
C_3
VT_{31}

Fig.9. Many virtual SSTables exist in read path.

1) File metadata (the representation of virtual SSTable and real SSTable): we add one field, i.e., parentSSTables, into the original file metadata of RocksDB to distinguish between a virtual SSTable and a real SSTable, and we use this field to store the real SSTable's file number for the virtual SSTable, so that when the length of this field of one SSTable is larger than 0, this SSTable is a virtual SSTable; otherwise it is a real SSTable.

2) Compaction (supporting virtual compaction): we modify the original compaction procedure to support virtual compaction, and the main role of virtual compaction is generating the virtual SSTable's metadata.

3) Read procedure (virtual SSTable merge): we add the virtual SSTable merge process into the original read procedure to improve the subsequent read performance.

Experimental Setup. We conduct experiments on one machine running Linux CentOS 6.5 final with kernel 2.6.32. The machine includes a two-socket Intel® Xeon® E5645 (6 cores with hyper-threading, 2.4 GHz, 12 MB L3 cache). The machine has 2 GB of DRAM, and two 1 TB 7200RPM SATA III disks (avg. seek/rotational time: 8.5 ms/4.2 ms, sustained transfer rate: 150 MB/s). Moreover, it has one 480 GB SATA III Intel 520 solid state drive (avg. read/write latency: 80 µs/85 µs, sequential read/write rate: 550/520 MB/s). We drop all the caches before running any benchmark.

For the configuration of RocksDB and dCompaction, we use the default parameter values for all experiments. That is, the MemTable size is 4 MB, the SSTable size is 2 MB and the data block size is 4 KB.

Workload Generation. We use YCSB to generate workload traces, which are replayed in a light-weight workload generator. YCSB generates synthetic workloads with varying degrees of read/write ratio, statistical distributions and value size. Here, the dataset used is denoted as (O, W, R, V, S) where O is the number of total operations, W is the ratio of write operations, R is the ratio of read operations, V is the record size and S is the total size of the dataset. We configure it to generate different datasets that are described in Subsection 4.1 and Subsection 4.2, and make use of Zipfian request distributions for experiments.

4.1 Overall Evaluation

Firstly, the datasets used are D_1: (2.4×10^7, 1, 0, 256 B, 6 GB), D_2: (2.4×10^7, 0.8, 0.2, 256 B, 5 GB), D_3: (2.4×10^7, 0.2, 0.8, 256 B, 2 GB), D_4: (2.4×10^7, 0, 1, 256 B, 6 GB) for 256 B records. Here, VCT and $VSMT$ of dCompaction are 12 and 5 respectively. Fig.10 shows the IOPS of RocksDB and dCompaction under these four datasets based on SSD, and several important observations stand out.

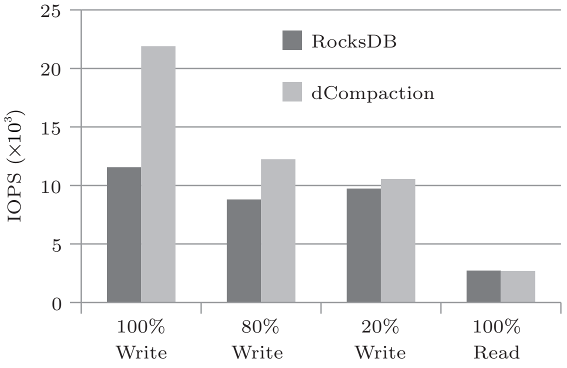


Fig.10. Comparison of dCompaction with RocksDB under D_1, D_2, D_3, D_4 on SSD.

1) For the write-100% workload (i.e., D_1), compared with RocksDB, write IOPS of dCompaction improves by 89.47%. The reason is that virtual compaction in dCompaction delays compaction to reduce the repeated I/O overheads and write amplification during the compaction procedure. In order to see the effect of virtual compaction, we evaluate the compaction data volume during the runtime of this workload as shown in Fig.11. From Fig.11, one can see that the I/O-savings on compaction read and compaction write by dCompaction from RocksDB are 48.27% and 51.23% respectively.

2) For the read-100% workload (i.e., D_4), we firstly load 6 GB data into RocksDB and dCompaction separately and then process read operations. As shown in Fig.10, the read operations have similar performance on read IOPS in both systems: 2 730 ops/sec

in RocksDB and 2 694 ops/sec in dCompaction (the read IOPS of dCompaction is only 1.32% fewer than that of RocksDB), and their read latency is also similar, namely 366.3 µs and 371.21 µs respectively (the read latency of dCompaction increases by only 1.34%, compared with RocksDB). This is because through reasonable *VCT* and virtual SSTable merger process, the number of real SSTables involved in the virtual SSTable is reduced during the read procedure, so that the read performance of dCompaction can be matched with that of RocksDB.

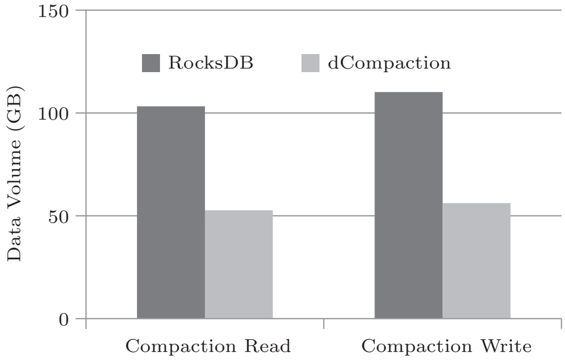


Fig.11. Compaction data volume of dCompaction and RocksDB based on SSD.

3) For the write-80% workload (i.e., D_2), due to the read operations, the total IOPS's improvement by dCompaction from RocksDB decreases to 39.01%, compared with D_1. Fig.12 shows the read latency and the write latency. As Fig.12 shows, compared with RocksDB, the write latency of dCompaction decreases by 45.28% due to the virtual compaction, while dCompaction and RocksDB have similar read performance, namely 232.33 µs and 238.42 µs respectively (the read latency of dCompaction increases by only 2.62%, compared with RocksDB).

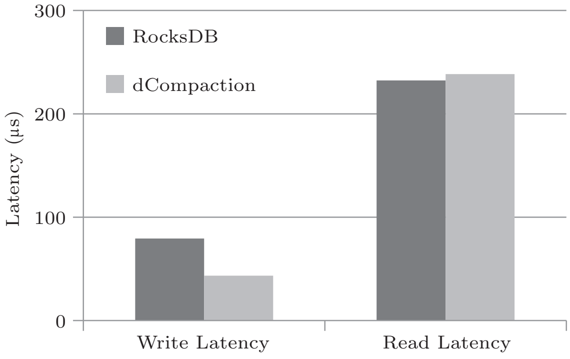


Fig.12. Read and write latency of dCompaction and RocksDB based on SSD.

4) For the write-20% workload (i.e., D_3), dCompaction and RocksDB perform similarly, namely 10 555 ops/sec and 9 737 ops/sec, since the runtime for this workload is mainly dominated by the read operations, just like D_4.

4.2 Sensitivity Study

To examine the impacts of data volume, record size, *VCT* and *VSMT*, we conduct a series of sensitivity studies through experiments about IOPS, latency of RocksDB and dCompaction.

4.2.1 Data Volume

To examine the impact of data volume, we conduct experiments on different data volumes (4 GB, 5 GB, 6 GB, 7 GB, 8 GB) with 256 B record size of write-100% workload and subsequent read-100% workload based on SSD. Fig.13 shows the write IOPS of dCompaction and RocksDB. For the write-100% workload, as the data volume increases, the write IOPS of dCompaction and RocksDB remains unchanged, and the write IOPS of dCompaction under different data volumes increases by 88.41%, 87.89%, 89.47%, 88.92%, 90.12% respectively compared with RocksDB, as a result of compaction I/O overheads decreases from virtual compaction which is shown in Fig.14(a).

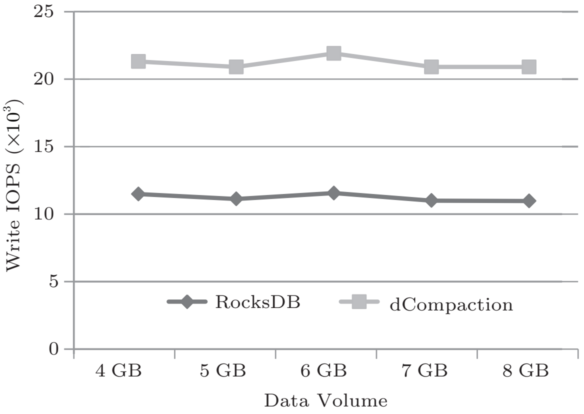


Fig.13. Write IOPS of dCompaction and RocksDB under different data volumes.

In the subsequent read-100% workload from Fig.14(b), one can see that as the data volume increases, the read performance of dCompaction and RocksDB slightly decreases due to the depth increase of LSM-tree. Also we can find dCompaction has similar read performance to RocksDB under different data volumes, and the performance gaps between them are very small, namely 2.11%, 1.79%, 1.32%, 2.72% and 2.29% respectively. The reason is that through the virtual SSTable merge, the average depth of read operations, i.e., the number of real SSTables involved in read process, is reduced in dCompaction.

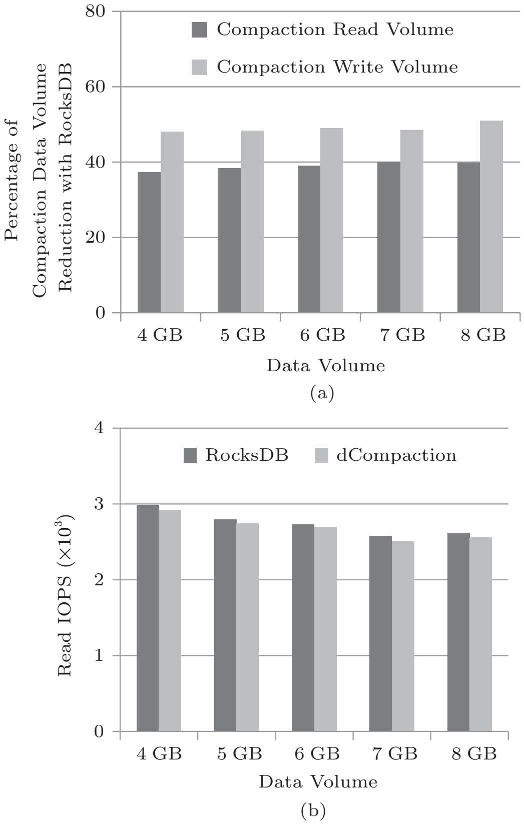


Fig.14. (a) Percentage of compaction data volume reduction, compared with RocksDB, based on SSD. (b) Read IOPS of dCompaction and RocksDB under different data volumes on SSD.

4.2.2 Record Size

To examine the impact of the record size, we conduct experiments on different record sizes (128 B, 256 B, 384 B, 512 B, 640 B) with the same 6 GB data volume of write-100% workload and subsequent read-100% workload based on SSD.

Fig.15 exhibits the write and the read IOPS of both dCompaction and RocksDB. For the write-100% workload, as the record size increases, the write IOPS of both dCompaction and RocksDB decreases because within the same data volume, the larger the record size, the more the overheads of compaction. However, compared with RocksDB, the write IOPS by dCompaction increases by 92.41%, 89.47%, 86.93%, 83.11%, 79.04% due to the virtual compaction. For the read-100% workload, as the record size increases, the read IOPS of both dCompaction and RocksDB remains unchanged, and the read performance gap between dCompaction and RocksDB is similar.

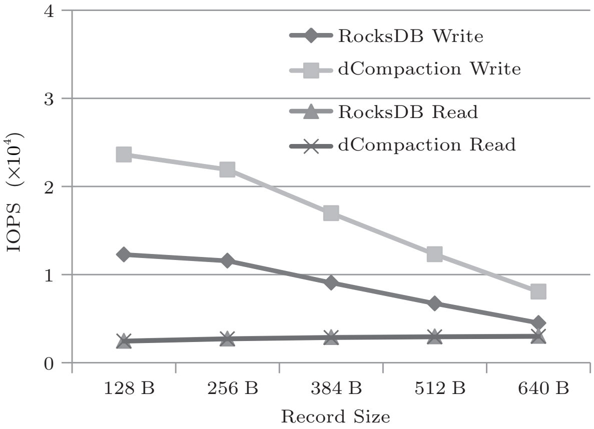


Fig.15. Write IOPS and read IOPS of dCompaction and RocksDB under different record sizes on SSD.

4.2.3 VCT

To examine the impact of *VCT*, we conduct experiments on different *VCT*s (4, 8, 12, 16, 20) with 256 B record size, 6 GB data volume of write-100% workload based on SSD. Fig.16 demonstrates the write latency with varying the value of *VCT* of dCompaction. In Fig.16, one can see that as *VCT* increases, the write latency of dCompaction firstly decreases and then increases. There are two reasons as follows.

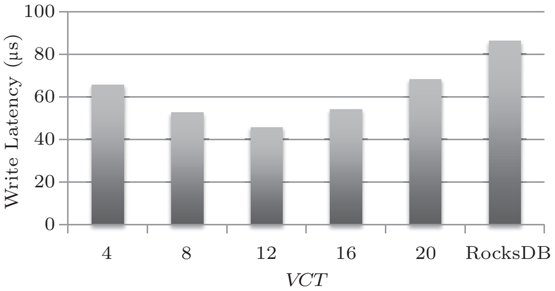


Fig.16. Write latency of dCompaction under different *VCT*s on SSD.

As mentioned above, the larger *VCT* is, the more the compaction I/O overheads can be saved. Consequently as *VCT* increases, the compaction read data volume and write data volume are decreased, which is shown in Fig.17(a). From Fig.17(a), when *VCT* is 12, its compaction read data volume and write data volume are both smaller than the other *VCT*s compaction read data volume and write data volume.

However, when *VCT* is large and the real compaction is triggered, there are many real SSTables in this real compaction that will prolong compaction to increase the compaction time as shown in Fig.17(b). In Fig.17(b), when *VCT* is 20, its compaction time is larger than some other *VCT*s due to the overheads of real compaction.

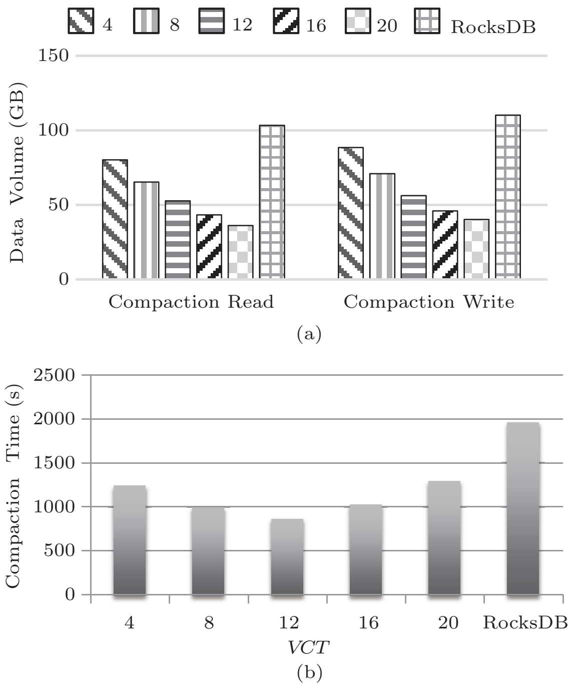


Fig.17. (a) Compaction read data volume and write data volume of dCompaction. (b) Compaction time of dCompaction under different *VCT*s on SSD.

4.2.4 *VSMT*

After playing the write-100% workload, we conduct experiments on different *VSMT*s (3, 5, 7, 9, 11) with 256 B record size, 6 GB data volume of read-100% workload based on SSD to examine the impact of *VSMT*. Fig.18(a) shows the read latency with varying the value of *VSMT* of dCompaction. From Fig.18(a), we can find as *VSMT* increases, the read latency of dCompaction increases, because the larger *VSMT* is, the less the virtual SSTables can be merged into real SSTables, thereby the number of real SSTables involved in read process becomes large, and it prolongs read latency. Fig.18(b) illustrates the average number of real SSTables involved in read process under different *VSMT*s. From Fig.18(b), one can see that the less *VSMT*, the less real SSTables to be read, and it is almost the same with original RocksDB's number of real SSTables to be read to when *VSMT* is set from 3 to 6.

4.3 Experiments on Hard Disks

The experiments described above are based on SSD. In order to study the performance of dCompaction based on disk, we also conduct some experiments to compare dCompaction with RocksDB. Similarly, we use four datasets and the same configuration of dCompaction and RocksDB which are described in Subsection 4.1, and the results are shown in Fig.19.

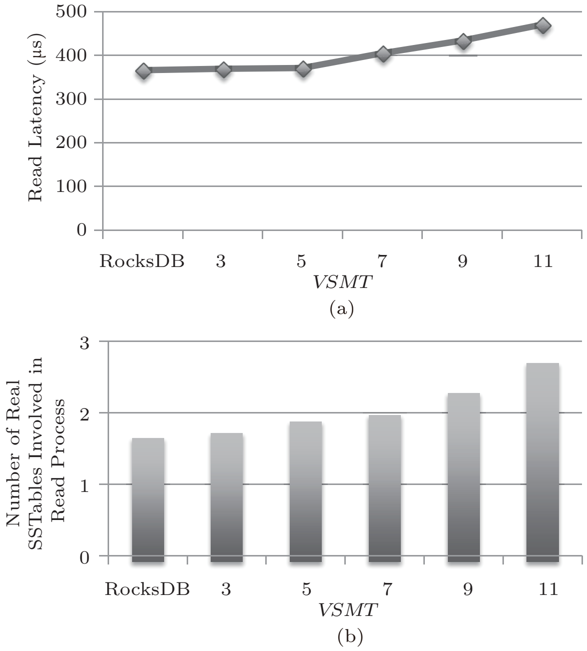


Fig.18. (a) Read latency of dCompaction and (b) number of real SSTables involved in read process under different *VSMT*s on SSD.

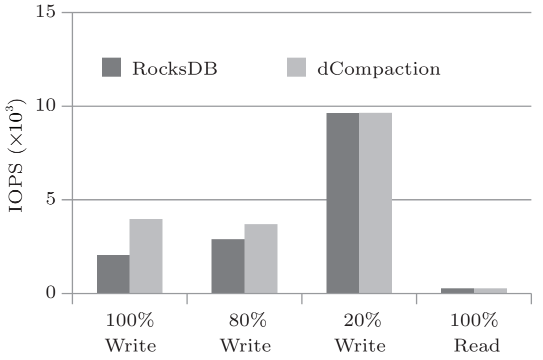


Fig.19. Comparison of dCompaction with RocksDB under D_1, D_2, D_3, D_4 on hard disks.

From Fig.19, we can get the similar observations that in the write-intensive workloads, e.g., D_1, D_2, the performance of dCompaction is better than that of RocksDB, and in the read-heavy workloads, e.g., D_3, D_4, the performance of dCompaction and RocksDB is similar.

5 Related Work

We categorize related work into two categories, improvement on read and write performance of KV-store (particularly on LSM-tree).

Improvement on Write Performance. bLSM[1] uses the replacement-selection sort algorithm in the compaction procedure of component C_0 to increase the length of sorted KV items, which decreases the frequency of compactions for all the components. VT-tree[10] looks for any block at a component whose key range does not overlap with that of blocks at the other component during the merge-sorting of the two components' KV items. In this way, it can avoid unnecessary disk I/O for sorted and non-overlap key range to speed up compactions. However the effectiveness of this method relies on the probability of having non-overlapping blocks and also the stitching technique of VT-tree may incur fragmentation. LSM-trie[13] introduces linear growth pattern to minimize the compaction cost for LSM-tree based KV systems; however LSM-tree uses hash functions to organize its data and accordingly does not support range search. bLSM and PE[11] partition the key range into multiple sub-key ranges and confine compactions in hot data key ranges, which accelerate the data flow. PCP[12] uses a pipelined compaction procedure to fully utilize both CPUs and I/O devices to speed up the compaction procedure. GTSSL[16] develops techniques to adapt to changing read-write ratios and adapts to hybrid disk-flash systems.

Improvement on Read Performance. With respect to read performance, LSM-tree requires each component to be checked for a read. To improve performance, datastores typically have four methods as follows:

1) caching frequently accessed data in memory;

2) protecting components with bloom filter[17] to avoid disk I/O for the level which does not contain the target KV item, e.g., RocksDB, Cassandra[6], bLSM[1];

3) using fractional cascading[18], where partial results from searching one component are used to speed up searching following components, e.g., Cache-Oblivious Lookahead Arrays (COLA)[19] and FD-tree[20];

4) other optimizations: GTSSL[16] uses the reclamation and re-insert techniques to put KV items in upper levels to decrease the count of levels that point queries go through. The partitioned exponential file[11] partitions the key range into multiple sub-key ranges, and then point queries just search one smaller sub-key range.

6 Conclusions

The LSM-tree incurs large I/O costs when data compaction is performed, and the repeated KV item reads and writes during compaction incur significant write amplification and then result in poor overall throughput. dCompaction replaces some compaction operations, which generate large amounts of I/O, with so-called virtual compaction. Since several virtual compactions can be combined into one real compaction, the total amount of I/O can be reduced. Hence dCompaction is able to significantly reduce repeated KV item reads and writes and decrease the write amplification. Further, in order to minimize the virtual SSTables' impact on read performance, we proposed two parameters named VCT and $VSMT$ to control the frequency of virtual compaction and limit the number of virtual SSTables respectively, and finally support comparable read performance while maintaining high writing performance. Extensive benchmark and real-world workloads driven experimental results demonstrated that compared with RocksDB, dCompaction has about more than 40% write performance improvement and matchable read performance when handling write-intensive workloads (e.g., write-80% workload), and also has comparable read performance when handling read-intensive workload (e.g., read-100% workload).

References

[1] Sears R, Ramakrishnan R. bLSM: A general purpose log-structured merge tree. In *Proc. the ACM SIGMOD International Conference on Management of Data*, May 2012, pp.217-228.

[2] Huang Q, Birman K, van Renesse R, Lloyd W, Kumar S, Li H C. An analysis of Facebook photo caching. In *Proc. the 24th ACM Symposium on Operating Systems Principles* (*SOSP*), Nov. 2013, pp.167-181.

[3] Atikoglu B, Xu Y, Frachtenberg E *et al.* Workload analysis of a large-scale key-value store. In *Proc. ACM SIGMETRICS*, Jun. 2012, pp.53-64.

[4] O'Neil P, Cheng E, Gawlick D *et al.* The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996, 33(4): 351-385.

[5] Chang F, Dean J, Ghemawat S, Hsieh W, Wallach D, Burrows M, Chandra T, Fikes A, Gruber R. Bigtable: A distributed storage system for structured data. In *Proc. the 7th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI*), Nov. 2006, pp.205-218.

[6] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010, 44(2): 35-40.

[7] George L. HBase: The Definitive Guide. O'Reilly Media, 2011.

[8] Escriva R, Wong B, Sirer E. HyperDex: A distributed, searchable key-value store. In *Proc. ACM SIGCOMM Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug. 2012, pp.25-36.

[9] Cooper B, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen H, Puz N, Weaver D, Yerneni R. PNUTS: Yahoo! hosted data serving platform. *Proc. the VLDB Endowment*, 2008, 1(2): 1277-1288.

[10] Shetty P, Spillane R, Malpani R *et al.* Building workload-independent storage with VT-trees. In *Proc. the 11th USENIX Conference on File and Storage Technologies* (*FAST*), Feb. 2013, pp.17-30.

[11] Jermaine C, Omiecinski E, Yee W G. The partitioned exponential file for database storage management. *The VLDB Journal*, 2007, 16(4): 417-437.

[12] Zhong Z, Yue Y, He B *et al.* Pipelined compaction for the LSM-tree. In *Proc. the 28th International Parallel and Distributed Processing Symposium* (*IPDPS*), May 2014, pp.777-786.

[13] Wu X, Xu Y, Shao Z *et al.* LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proc. the USENIX Annual Technical Conference* (*ATC*), Jul. 2015, pp.71-82.

[14] Amur H, Andersen D, Kaminsky M *et al.* Design of a write-optimized data store. Technical Report GIT-CERCS-13-08, Georgia Tech CERCS, 2013.

[15] Cooper B, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In *Proc. the 1st ACM Symposium on Cloud Computing* (*SoCC*), Jun. 2010, pp.143-154.

[16] Spillane R, Shetty P, Zadok E, Dixit S, Archak S. An efficient multi-tier tablet server storage architecture. In *Proc. the 2nd ACM Symposium on Cloud Computing in Conjunction with SOSP* (*SoCC*), Oct. 2011, pp.1-14.

[17] Bloom H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970, 13(7): 422-426.

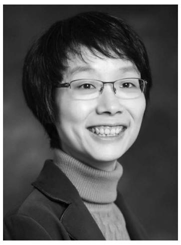
[18] Chazelle B, Guibas L. Fractional cascading: A data structuring technique with geometric applications. In *Proc. the 12th International Colloquium on Automata, Languages, and Programming* (*ICALP*), Jul. 1985, pp.90-100.

[19] Bender M, Farach-Colton M, Fineman J, Fogel Y, Kuszmaul B, Nelson J. Cache-oblivious streaming B-trees. In *Proc. the 19th Annual ACM Symposium on Parallel Algorithms and Architectures* (*SPAA*), Jun. 2007, pp.81-92.

[20] Li Y, He B, Yang R J *et al.* Tree indexing on solid state drives. *Proc. the VLDB Endowment*, 2010, 3(1/2): 1195-1206.

Feng-Feng Pan received his B.S. degree in computer science and technology from Central South University, Changsha, in 2010. He is now a Ph.D. candidate in the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include big data storage and management, storage systems, and big data analysis.

Yin-Liang Yue received his B.S. degree in computer science from Harbin Institute of Technology, Harbin, in 2005, and his Ph.D. degree in computer science from Huazhong University of Science and Technology, Wuhan, in 2011. He is currently an associate professor of Institute of Information Engineering, Chinese Academy of Sciences, Beijing. His current research interests include big data storage system and big data analysis. He is a member of CCF, ACM, and IEEE.



Jin Xiong received her B.S. degree from Sichuan University, Chengdu, in 1990, her M.S. degree from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1993, and her Ph.D. degree from Graduate University of Chinese Academy of Sciences, Beijing, in 2006, all in computer science. She is currently a professor at ICT, CAS, Beijing. Her research interests include big data storage and management, storage systems, and file systems. She is a senior member of CCF and a member of ACM and IEEE.