Reducing Write Amplification of LSM-Tree with Block-Grained Compaction

Xiaoliang Wang[1], Peiquan Jin[1*], Bei Hua[1], Hai Long[2], Wei Huang[2]
[1]*University of Science and Technology of China, Hefei, China*
[2]*Huawei Technologies Co., Ltd., Shenzhen, China*
*jpq@ustc.edu.cn

Abstract*—LSM-tree has been widely used as a write-optimized storage engine in many key-value stores, such as LevelDB and RocksDB. However, conventional compaction operations on the LSM-tree need to read, merge, and write many SSTables, which we call *Table Compaction* in this paper. Table Compaction will cause two major problems, namely write amplification and block-cache invalidation. They will lower both write and read performance of the LSM-tree. To address these issues, we propose a novel compaction scheme named *Block Compaction* that adopts a block-grained merging policy to perform compaction operations on the LSM-tree. Block Compaction identifies the boundaries of data blocks and tries to avoid reusing data blocks, which not only reduces the write amplification but also alleviates the block-cache invalidation. We present cost analysis to theoretically demonstrate that Block Compaction is more efficient than the existing Table Compaction. Furthermore, we analyze the side-effects of Block Compaction and present three optimizations: (1) Selective Compaction is to reduce the space amplification of Block Compaction by integrating Table Compaction with Block Compaction. (2) Parallel Merging divides a compaction task into several sub-tasks and uses multiple workers to accomplish sub-tasks in parallel. (3) Lazy Deletion mitigates the overhead caused by traversing files at the tail of compaction operations. We implement a new key-value store named BlockDB based on Block Compaction and its optimizations. Then, we compare BlockDB with LevelDB, RocksDB, and L2SM using the YCSB benchmark. The results show that BlockDB can reduce write amplification up to 32% and running time by up to 43.6%, compared to its competitors. In addition, it can maintain the high performance for point lookups and range scans.

***Index Terms*—Write amplification, Block compaction, LSM-tree**

I. Introduction

LSM-tree [1] is widely used in modern key-value stores, such as RocksDB [2], HBase [3], and Cassandra [4], to deal with modern data-intensive applications. LSM-tree is a leveled data structure, and each level is composed of several sorted files called SSTables. The key innovative idea of the LSM-tree is to transform small random writes into large sequential writes, which is mainly accomplished by compaction operations [5]. A conventional compaction operation in current LSM-tree-based key-value stores, e.g., LevelDB [6] and RocksDB [2], uses an SSTable-grained scheme to merge one or multiple SSTables in a low level into new SSTables in a high level [7].

Unfortunately, conventional SSTable-grained compaction will cause serious write amplification [8]. Write amplification will consume a majority of disk bandwidths, which will severely lower the system throughput and even cause write stalls [9]–[12]. Also, compaction will invalidate the block cache [13], because old SSTables will be merged into new SSTables, which will make the cached blocks associated with the old SSTables invalid and worsen the search performance on the LSM-tree [14].

So far, a few methods have been proposed to address the write amplification problem on the LSM-tree. For example, Cassandra [4] and HBase [3] adopted the tiering compaction strategy that can accumulate more key-value pairs in each compaction. In this way, LSM-tree can avoid rewriting the key-value pairs at the same level repeatedly. However, it will incur additional read costs and high space overhead [15]. In general, many existing approaches addressed the write amplification issue by sacrificing read performance or other functions (e.g., sacrificing range queries [16]). These schemes are not acceptable to read-write-mixed workloads and read-intensive workloads [14], [17], [18]. As a result, it has been a trend to devise a new approach to solve the write amplification problem of the LSM-tree without sacrificing the read performance [19].

In this paper, we propose a new compaction method called *Block Compaction*. Unlike the conventional SSTable-grained compaction (we call it *Table Compaction* in this paper) that is adopted by many key-value stores like LevelDB and RocksDB, Block Compaction only reads those data blocks involved by a compaction operation, rather than reading the whole SSTable. Further, it only appends data blocks to SSTables, rather than writing back whole SSTables. Thus, with the adoption of block-grained merging units, Block compaction can significantly reduce write amplification. Moreover, with Block Compaction, many data blocks need not be rebuilt, meaning that most blocks in the block cache will keep valid after Block Compaction. Therefore, Block Compaction can alleviate the invalidation problem of the block cache caused by compaction on the LSM-tree. Briefly, we make the following contributions in this paper.

- We propose a novel Block Compaction that adopts a block-grained merging policy to perform compaction operations on the LSM-tree. Compared to the traditional SSTable-grained Table Compaction scheme, Block Compaction can not only significantly reduce write amplification but also efficiently alleviate the block-cache invalidation problem on the LSM-tree.
- We propose to a hybrid compaction scheme named Se-

lective Compaction to integrate Block Compaction with Table Compaction. The selective compaction can choose one of the two compaction schemes dynamically based on several parameters, such as valid ratio and valid size. It can further reduce the write amplification of Block Compaction and also avoid large space consumption by Table Compaction, achieving a better trade-off between Block Compaction and Table Compaction.

- We propose to three individual optimizations, *Parallel Merging* and *Lazy Deletion*, to accelerate Block Compaction. *Parallel Merging* divides a single compaction task into several individual sub-tasks and uses multiple workers to finish sub-tasks in parallel, which can fully exploit the idle CPU resources and the parallelism of storage devices. *Lazy Deletion* mitigates the overhead caused by traversing files at the tail of compaction operations. We experimentally show that these optimizations are effective for speeding up Block Compaction.
- Block Compaction and its optimizations on LevelDB (version 1.20), and the new key-value store is called BlockDB. We conduct extensive experiments on the YCSB workloads to compare BlockDB with LevelDB, RocksDB, and L2SM. The results show that BlockDB can reduce write amplification up to 32% and running time by up to 43.6%, compared to its competitors. In addition, it can maintain high performance for point lookups and range scans.

The rest of the paper is structured as follows. Section II introduces the background of LSM-tree and related work. Section III details the design of Block Compaction. Section IV presents the optimizations of Block Compaction. Section V reports the experimental results, and finally, in Section VI, we conclude the paper and discuss the future work.

II. Related Work

LSM-tree is a multi-level data structure designed primarily for block-based storage devices to process write-intensive workloads [1], [20], which transforms small random writes into large sequential writes to improve write throughput. LSM-tree is composed of multiple components termed C_i consisting of Sorted String Tables (SSTables). The C_0 component consists of MemTable and Immutable MemTable, and both of them reside in memory. The $C_{i(i>0)}$ composed of multiple ordered SSTables is disk-resident. These components are organized in levels whose sizes increase exponentially.

For two adjacent levels on the LSM-tree, we define L_i at the upper level as *parent level* and L_{i+1} at the lower level as *child level*. A compaction operation on the LSM-tree is to merge the key-value pairs in the parent and child levels into the child level, which is triggered when the parent level reaches its capacity threshold. The conventional compaction in LevelDB and RocksDB adopts an SSTable-grained compaction scheme, which needs to read, merge, and write all influenced SSTables. To this end, we regard the conventional compaction as *Table Compaction*. Figure 1 shows an example of Table Compaction happened between two adjacent levels. First, it selects an

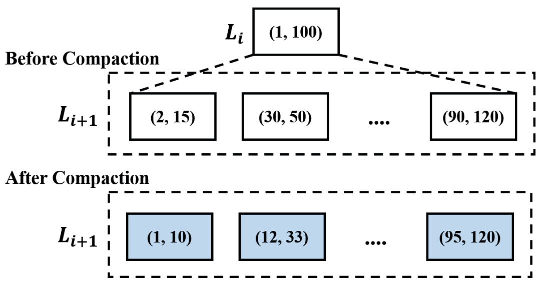


Fig. 1. Illustration of *Table Compaction*.

SSTable at the parent level by a specified strategy (e.g., using a round-robin approach in LevelDB). Then, it chooses the qualified SSTables at the child level, whose key ranges must be overlapped with the key range of the selected SSTable. Next, LSM-tree loads these SSTables into main memory and sorts all key-value pairs through a merge-sort algorithm. After that, LSM-tree builds new SSTables and writes them into the child level by sequential I/Os. Finally, all old SSTables become obsolete, and LSM-tree deletes them instantly.

Table Compaction regards an SSTable file as an operation unit (read/merge/write) to accomplish compaction operations. This approach can keep data blocks sorted and compacted, which helps LSM-tree reduce space amplification and improve range-query performance. However, Table Compaction has to rewrite all key-value pairs in the SSTables, which will incur serious write amplification (i.e., the ratio of physical write I/Os to the size of the data set). However, write amplification is a critical metric because high write amplification will occupy disk bandwidth, which damages the read/write performance of the LSM-tree, and shorten the lifetime of storage devices, especially for SSDs.

In recent years, there are a few studies focusing on the write amplification problem of the LSM-tree [20]–[26]. dCompaction [24] introduced the concepts of Virtual SSTable and Virtual Compaction. During Virtual Compaction, dCompaction just merges metadata of normal SSTables to build virtual SSTables without reorganizing data blocks. For virtual SSTables, dCompaction performs a real compaction operation by merging both metadata and data blocks. This design delays the behavior of merging data blocks. Thus, it can avoid repeated I/Os and reduce write amplification. However, the data structure of virtual SSTables increases the length of the search path of query processing.

PebblesDB [25] proposed a novel data structure that integrated skip-lists with the LSM-tree. It introduces *Guards* that partition the key range of each level into multiple small regions. In every region, the SSTables with overlapped key ranges can co-exist. When an SSTable is moved from L_i to L_{i+1}, it will be divided into multiple parts according to the guards in L_{i+1}, and then these parts are written to L_{i+1}. With this mechanism, PebblesDB can avoid rewriting the key-value pairs in L_{i+1} during compaction. However, SSTables may contain overlapped key ranges, which will hurt the read performance because additional read costs need to be paid for searching the overlapped key ranges. As a result, PebblesDB reduces write amplification at a high expense of read performance. VT-tree [21] proposed to extend the LSM-tree to efficiently handle sequential workloads by using a stitching technique to reuse data blocks. However, it will incur additional overhead of garbage collection. L2SM [26] proposed to isolate key-value items that bring a disruptive effect on the LSM-tree by using a multi-level log structure. L2SM identifies the SSTables that contain many hot keys and moves them to the log structure, which can avoid the write amplification caused by updates. Wisckey [22] proposed to change the data layout of the LSM-tree by separating keys with values. They only store keys and pointers (used to index values) on the LSM-tree and maintain values in a specific log-structured component. Compared to traditional LSM-trees, Wisckey can avoid the repeated writing of values and reduce write amplification. However, the key-value separation policy sacrifices read performance, especially for scan performance.

III. Block Compaction

A. Main Idea

Table Compaction adopted by conventional LSM-tree-based key-value stores like LevelDB and RocksDB regards an SSTable file as a basic operation unit for read/merge/write operations during compaction operations. However, we observe that many data blocks in the new SSTables are the same as those in the merged SSTables. That means Table Compaction has to write a significant number of unnecessary data blocks, which will increase the ratio of write amplification. To handle this problem, we propose to adopt a block-grained unit, e.g., 4 KB or 8 KB, to perform compaction operations. With this mechanism, we just need to focus on the affected data blocks (dirty blocks) and merge key-value pairs within those blocks during a compaction process. Compared with Table Compaction, Block Compaction needs not to read/write those unaffected data blocks. Thus, it can significantly reduce the number of data blocks rewritten and, in turn, reduce write amplification. Moreover, Block Compaction has other advantages. It rewrites few data blocks, implying that many data blocks are still valid in the block cache after compaction. Therefore, it can alleviate the block-cache invalidation problem and improve the read performance of the LSM-tree.

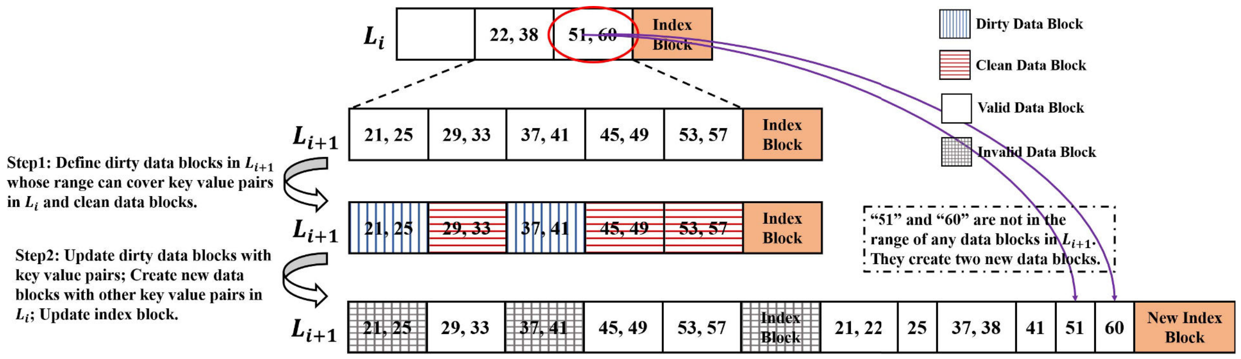
Since Block Compaction only reads and writes the affected data blocks, it need not create a new SSTable for compaction. Instead, it appends the modified data blocks to the tail of the SSTables at the lower level. The append-based writing scheme can ensure sequential I/Os, making full use of the disk bandwidth. To search the key-value pairs in the modified SSTables, we rebuild a new index block for SSTables to locate all valid data blocks, including the modified data blocks and the unaffected data blocks.

B. Data Structure

The process of Block Compaction is illustrated in Fig. 2. When L_i (the parent level) becomes full, we select an SSTable in L_i, which is similar to that in Table Compaction. Then, we merge it with the overlapped SSTables in L_{i+1} (the child level). For each overlapped SSTable in L_{i+1}, we classify their data blocks into two types, *clean blocks* and *dirty blocks*, according to whether a data block needs to be rewritten. If the key range of a data block can cover multiple keys in the selected SSTable, we regard it as a dirty block (e.g., the first and the third data block in L_{i+1} in Fig. 2), which needs to be rewritten later. If not, it is regarded as a clean block, and we can reuse it later. During compaction, for every dirty block, we load it into the main memory and merge it with the corresponding key-value pairs in the selected SSTable that are within the key range of the dirty block. Then, we create new data blocks and write the merged key-value pairs to the tail of the SSTable. For clean blocks, we do not rewrite them. Thus, clean blocks can stay valid in the block cache. After rewriting all dirty blocks, we build a new index block for all valid data blocks, including the old data blocks (e.g., the second and the fourth data blocks in L_{i+1} in Fig. 2) and the new data blocks. Finally, we create a new footer to store the offset of the index block and other metadata, such as Bloom filters. The example in Fig. 2 shows that Block Compaction only needs to rewrite a few data blocks instead of the whole SSTable.

One special case is that one or multiple key-value pairs in the selected SSTable are not within the key ranges of any data blocks in L_{i+1}. In this case, we use these key-value pairs to form a new data block directly rather than rewriting the data blocks in L_{i+1}. In Fig. 2, key "51" or "60" in the selected SSTable are not within the key ranges of any data blocks in L_{i+1}. Therefore, two small data blocks are generated, and we need not rewrite any blocks in L_{i+1}. The index handles of these data blocks are stored in the index block to keep the SSTable sorted logically. This strategy further helps reduce the re-writes to data blocks, which is helpful to reduce write amplification.

An index entry of a conventional index block in LevelDB and RocksDB only stores either the largest key or the smallest key of a data block. To identify the type of a data block at the child level, we add both the smallest key and the largest key of a data block into the index entry of the block. Moreover, we notice that the smallest key and the largest key in a data block usually have a long common prefix. Therefore, instead of storing the full string of a key, we only store the non-shared part of the smallest key to reduce the space overhead. Figure 3 shows the data structure of the index block in our design. An index entry is composed of several fields representing necessary information for a data block. The field *Key String* stores the largest key of the data block. *Key Size* stores the size of *Key String*. *Shared Size* stores the length of the common prefix shared by the smallest key and the largest key. In addition, *Non-Shared String* stores the non-common part of the smallest key, and *Non-Shared Size* stores the size of *Non-Shared String*. The field *Value Size* and *Offset* store the length and the offset of the data block, respectively. Such a revised index-block structure can improve the performance of point queries because it can filter more candidate data blocks without disk I/Os.



L_i
22, 38
51, 60
Index Block
L_{i+1}
21, 25
29, 33
37, 41
45, 49
53, 57
Index Block
Step1: Define dirty data blocks in L_{i+1} whose range can cover key value pairs in L_i and clean data blocks.
Step2: Update dirty data blocks with key value pairs; Create new data blocks with other key value pairs in L_i; Update index block.
Dirty Data Block
Clean Data Block
Valid Data Block
Invalid Data Block
"51" and "60" are not in the range of any data blocks in L_{i+1}. They create two new data blocks.
21, 22
25
37, 38
41
51
60
New Index Block

Fig. 2. Illustration of Block Compaction.

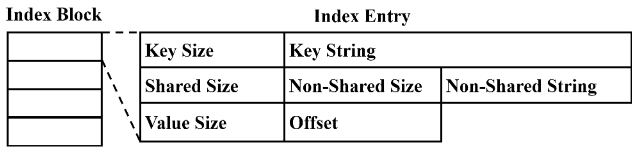


Fig. 3. Data structure of an index block.

C. Algorithms

Algorithm 1 shows the process of Block Compaction. We use *kv_iter* to traverse all the key-value pairs in the selected SSTable in L_i. We get an index iterator *idx_iter* of *S* to traverse the index entries of the index block in the overlapped SSTable. *idx_iter* has the boundaries of data blocks. Then, we skip data blocks indicated by *idx_iter* until the largest key of the data block in L_{i+1} is over the key pointed by *kv_iter*. As the skipped data blocks are clean blocks, we can directly reuse the index entries pointing to them. Then, we traverse and skip the key-value pairs with *kv_iter* until the key is larger than the smallest key of the data block in L_{i+1}. The skipped key-value pairs are not within any data blocks in the overlapped SSTable. Thus, we gather them to create one or multiple data blocks. At this moment, if the key pointed by *kv_iter* is smaller than the largest key of the data block, we perform the *UpdateBlock* function to rewrite this data block. If not, we take the *idx_iter* one step forward and enter the next loop until the *idx_iter* or *kv_iter* is invalid. Algorithm 2 shows *UpdateBlock* function. We use the merge sort algorithm to sort key-value pairs.

Note that reading dirty blocks into memory may introduce extra random-read I/Os because dirty blocks are usually distributed in the overlapped SSTable randomly. To address this issue, we first obtain the offsets and sizes of the dirty blocks in the overlapped SSTable in advance through the *FindDirtyBlocks* function, which is described in Algorithm 3 and looks like a mini-version of the *BlockCompaction* algorithm. After that, we read these dirty blocks concurrently using multi-threads. Such a design can fully exploit the massive internal parallelism of SSDs.

In summary, compared with Table Compaction, Block Compaction adopts a block-grained operation unit to accomplish compaction operations. Furthermore, Block Compaction identifies the boundaries of data blocks and tries to avoid rewritten data blocks. Thus, it can only rewrite few dirty blocks. With this mechanism, we can significantly reduce the I/Os caused by one compaction operation, which can not only reduce write amplification but save both CPU resources and disk bandwidth. Moreover, Block Compaction keeps clean blocks valid after compaction, ensuring the validity of the blocks in the block cache. Thus, it also can partially solve the block-

Algorithm 1: BlockCompaction

```
Input: kv_iter, key-value iterator for a selected SSTable; S,
       an overlapped SSTable
/* Step 1: Create a new index block.                */
$E$=AllocateNewIndexBlock();
/* Used to traverse index entries                   */
$idx\_iter$=$S$.NewIndexIterator();
while idx_iter is valid and kv_iter is valid do
    /* Step 2: Traverse index entries and
       store the entries of clean blocks.
       */
    while idx_iter.largest_key <kv_iter.key do
        $E$.addData($idx\_iter$);
        $idx\_iter$.next();
    end
    /* Step 3: Traverse key-value pairs and
       append new blocks.                           */
    $N$=AllocateNewDataBlock();
    /* Extract the smallest key                     */
    $k_{min}$=FirstKey($idx\_iter$.value);
    while $kv\_iter.key < k_{min}$ do
        $N$.addData($kv\_iter$.data);
        $kv\_iter$.next();
    end
    $S$.append($N$);
    /* Step 4: Update the data block with
       key-value pairs.                             */
    $B$=GetDataBlock($idx\_iter$.value);
    UpdateBlock($kv\_iter$, $B$);
    $idx\_iter$.next();
end
/* Add the remaining index entries.                 */
while idx_iter is valid do
    $E$.addData(idx_iter);
    idx_iter.next();
end
return
```

Algorithm 2: UpdateBlock

```
Input: kv_iter, key-value iterator; B, target data block.
bk_iter=B.NewIterator();
N=AllocateNewDataBlock();
/* Merge sort */
while kv_iter is valid and bk_iter is valid do
    if kv_iter.key < bk_iter.key then
        N.addData(kv_iter.data);
        kv_iter.next();
    else
        N.addData(bk_iter.data);
        bk_iter.next();
    end
    if N is full then
        AppendDataBlock(N);
        N=AllocateNewDataBlock();
    end
end
/* Add remaining key-value pairs */
while bk_iter is valid do
    N.addData(bk_iter.data);
    bk_iter.next();
    if N is full then
        AppendDataBlock(N);
        N=AllocateNewDataBlock();
    end
end
if N is not empty then
    AppendDataBlock(N);
end
```

Algorithm 3: FindDirtyBlocks

```
Input: kv_iter, the key-value iterator; S, the overlapped
       SSTable.
Output: R, the offset array of dirty blocks
/* Used to traverse index entries */
idx_iter=S.NewIndexIterator();
while idx_iter is valid and kv_iter is valid do
    /* Step 1: Traverse index entries. */
    while idx_iter.largest_key < kv_iter.key do
        idx_iter.next();
    end
    /* Step 2: Traverse key-value pairs. */
    /* Extract the smallest key */
    $k_{min}$=FirstKey(idx_iter.value);
    while kv_iter.key < $k_{min}$ do
        kv_iter.next();
    end
    /* Step3: Add the location of the dirty
       block into the location array R. */
    R.add(idx_iter.value);
    if kv_iter.key < idx_iter.key then
        while kv_iter.key < idx_iter.key do
            kv_iter.next();
        end
    end
    idx_iter.next();
end
return R;
```

cache invalidation problem and improve the read performance of the LSM-tree.

TABLE I
NOTATIONS FOR THE LSM-TREE.

Notation	Definition	Example
D	data set size	40 GB
B	data block size	4 KB
M	L_0 size	10 MB
k	key-value pair size	1 KB
N	number of levels	7
a	amplification ratio	10

D. Cost Analysis

In this section, we analyze the cost of Block Compaction with a comparison to Table Compaction. The notations used are shown in Table I. L_i has a capacity of $M \cdot a^i$. Thus, the amount of levels can be calculated by Eq. 1.

$$N = \left\lceil \log_a \left(\frac{D}{M} \cdot \frac{a-1}{a} \right) \right\rceil \tag{1}$$

Each key-value pair is first written into the disk by the flush operation and merged multiple times to the last level. Here, we consider the worst case. The average flush cost is $\frac{k}{B}$. For the conventional Table Compaction, to merge a key-value pair from L_0 to L_N, the average merging cost is $\frac{k}{B} \cdot (a+1) \cdot N$. Therefore, we can calculate the average write cost of Table Compaction by Eq. 2.

$$WriteCost_{table} = \frac{k}{B} + \frac{k}{B} \cdot (a+1) \cdot N \tag{2}$$

For Block Compaction, the average flush cost does not change and is still $\frac{k}{B}$. The number of dirty blocks is related to the number of the key-value pairs in the selected SSTable. In the worst case, every key-value pair in the selected SSTable corresponds to a dirty block in the overlapped SSTables. Thus, the average merging cost is$\frac{k}{B} \cdot (\frac{B}{k} + 1) \cdot N$. Then, we can get the average write cost of Block Compaction by Eq. 3.

$$WriteCost_{block} = \frac{k}{B} + \frac{k}{B} \cdot (\frac{B}{k} + 1) \cdot N \tag{3}$$

By comparing Eq. 1 and Eq. 2, we can see that Table Compaction is much sensitive to the amplification ratio, while Block Compaction does not have this feature. As shown in Table I, when we set the key-value pair size k to 1024 B, the block size B to 4 KB, and the amplification ratio a to 10. We can get the following observation, as shown in Eq. 4.

$$WriteCost_{block} < WriteCost_{table} \tag{4}$$

However, Block Compaction is sensitive to the amount of key-value pairs in one data block. Therefore, When meeting small data, Block Compaction may degenerate into Table Compaction (but not worse) and cause additional space overhead. In the next section, we will introduce other optimizations to alleviate this issue for Block Compaction.

E. Advantages of Block Compaction

Compared to the conventional Table Compaction adopted by LevelDB and RocksDB, our proposed Block Compaction has the following advantages.

Low Write Amplification. Block Compaction adopts a block-grained operation unit to accomplish compaction operations. Thus, it can rewrite a few dirty blocks and reuse clean blocks, which can significantly reduce the I/Os caused by one compaction operation and reduce write amplification.

Block-Cache Friendly. Block Compaction keeps clean blocks valid after compaction, which can ensure the validity of the blocks in the block cache. Thus, it can partially solve the block-cache invalidation problem and improve the read performance of the LSM-tree. As a result, Block Compaction is expected to lower block-cache misses than RocksDB, which has been verified by the experimental results in Section V.

High Efficiency. Block Compaction writes fewer data than conventional Table Compaction. Thus, it can save both CPU resources and disk bandwidth, meaning that compaction operations can be finished much faster than before, specifically for write-intensive workloads. As a result, Block Compaction can deliver a higher throughput than the conventional Table Compaction, and our experimental results in Section V will demonstrate this claim.

IV. Optimizations of Block Compaction

As a coin has two sides, Block Compaction suffers from some side effects. First, it incurs additional space consumption. After appending dirty blocks at the tail of the SSTable, selected data blocks at the lower level become obsolete. In the worst case, if all the data blocks in the overlapped SSTable are dirty, Block Compaction will have similar write amplification as Table Compaction but cause 2x space overhead. Second, after several Block Compaction operations, valid data blocks are randomly distributed in the SSTable. This is not friendly to range queries. On the contrary, Table Compaction can remove obsolete SSTable files after each compaction operation immediately.

In this section, we propose three optimizations for Block Compaction, including *Selective Compaction*, *Parallel Merging*, and *Lazy Deletion*, to reduce the side-effects and improve the performance of Block Compaction. In addition, we will discuss how to equip Block Compaction with Bloom filters.

A. Selective Compaction

We note that if most data blocks in the overlapped SSTable are dirty, Block Compaction will also incur similar write amplification as Table Compaction but cause additional space overheads. Thus, We propose to combine Block Compaction with Table Compaction to fully exploit the characteristics of different levels and the benefits of both Block Compaction and Table Compaction.

Figure 4 shows the main idea of *Selective Compaction*. In a compaction operation, for each overlapped SSTable, we determine its compaction type according to three parameters, including *dirty ratio, valid ratio*, and *valid size*. Below, we present the details of these parameters.

Dirty Ratio. The dirty ratio is the percentage of the dirty blocks in the overlapped SSTable. If the dirty ratio is higher than a specific threshold (*MAX_DIRTY_RATIO*), we adopt Table Compaction to avoid large space consumption. Otherwise, we choose Block Compaction to reduce write amplification. In Fig. 4, the left-most SSTable in L_{i+1} has a low dirty ratio. Thus, it will be compacted by Block Compaction. On the other hand, the left-second SSTable has a high dirty ratio, and it will be modified by Table Compaction. Here, we use the *FindDirtyBlocks* algorithm (Algorithm 3) to get the size of the dirty blocks in the overlapped SSTable.

Valid Ratio. The valid ratio is the percentage of the valid blocks in the overlapped SSTable. If the valid ratio is less than a specified threshold (*MIN_VALID_RATIO*), we use Table Compaction to remove the obsolete data blocks in the overlapped SSTable. For example, the left-third SSTable in L_{i+1} in Fig. 4 has a low valid ratio. Thus, it will be compacted by Table Compaction.

Valid Size. The valid size refers to the size of the valid data blocks in the overlapped SSTable. This ratio is relevant to the scale of the compaction operation. A large valid size exceeding the threshold (*MAX_VALID_SIZE*) means that the data blocks are likely out of order. Thus, we use Table Compaction to split the SSTable into several small ordered SSTable files. The right-most SSTable in L_{i+1} in Fig. 4 shows an example.

We use Selective Compaction for all levels except L_0 and L_1. This is because L_0 stores the SSTables flushed from the main memory, and different SSTables usually contain overlapped key ranges. Therefore, a compaction operation between L_0 and L_1 may involve all the SSTables at the levels. To this end, it is useless to adopt Block Compaction, and Table Compaction is more suitable.

Although all the levels except L_0 and L_1 can use Selective Compaction, we set different triggering thresholds for different levels. Such a design is based on two reasons. First, we notice that all key-value pairs are compacted from top levels to bottom levels until they reach the bottom-most level. Thus, for upper levels, we prefer to use Block Compaction to minimize the write amplification at upper levels. Second, when answering range queries, LSM-tree needs to traverse the key-value pairs of many levels, and may scan the bottom-most level because it stores the majority of key-value pairs. Thus, to improve the efficiency of range query processing, we prefer to perform Table Compaction to keep the data blocks in the lower levels sorted.

Algorithm 4 shows the process of *Selective Compaction*. We employ Table Compaction to create small SSTables when the valid size exceeds a threshold (Line 1-4). We also employ Table Compaction to perform garbage collection when the valid ratio is smaller than a threshold (Line 5-8). In addition, we employ Block Compaction to avoid write amplification when the dirty ratio is smaller than a threshold (Line 10-14). In fact, the dirty ratio is a trade off between space amplification and write amplification. For $L_{i(i<N)}$, we aim to avoid write amplification, while for L_N, we aim to avoid space amplification.

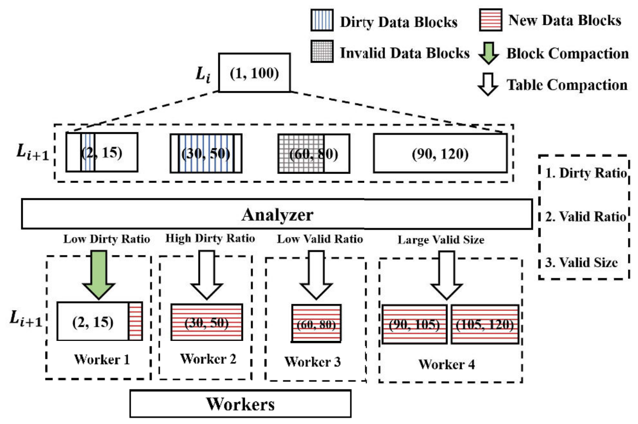


Fig. 4. Illustration of *Selective Compaction* and *Parallel Merging*.

Algorithm 4: SelectiveCompaction

```
Input: le, level; kv_iter, key-value iterator of selected
       SSTable; S, target SSTable;
/* SSTable valid size is large.                          */
1  if S.size() < max_file_size[le] then
2      TableCompaction(kv_iter, S);
3      return;
4  end
/* Obsolete data blocks are many.                        */
5  if S.valid_size() / S.size() < min_valid_ratio[le] then
6      TableCompaction(kv_iter, S);
7      return;
8  end
/* Dirty blocks are many.                                */
9  R=FindDirtyBlocks(kv_iter, S);
10 if R.size() / S.size() > max_dirty_ratio[le] then
11     TableCompaction(kv_iter, S);
12     return;
13 end
14 BlockCompaction(kv_iter, S);
15 return;
```

B. Parallel Merging

A compaction operation can be regarded as a task. And, a task deals with multiple SSTables that come from two adjacent levels (i.e., L_i and L_{i+1}). For most key-value stores (e.g., LevelDB and RocksDB), a task is only executed by a background thread. Here, we propose to utilize multiple threads to develop a *Parallel Merging* scheme to further accelerate Block Compaction.

Figure 4 shows the mechanism of *Parallel Merging*. We maintain a thread pool and a work queue in the main memory. When L_i becomes full, and a compaction operation is triggered, we identify a selected SSTable and multiple overlapped SSTables like before. Then, we divide the compaction task into several independent sub-tasks according to the number of the overlapped SSTables in L_{i+1}. Then, we let a sub-task handles one overlapped SSTable in L_{i+1}. We put these sub-tasks into the task queue. When there is an idle worker in the thread pool, the worker is waked up to accomplish a sub-task assigned by the task queue. In this way, we can execute multiple sub-tasks in parallel and exploit idle CPU resources and the parallelism of storage devices for accelerating Block Compaction.

TABLE II
RUNNING TIME (S) ON DIFFERENT DATASETS.

Type	40 GB	80 GB
LevelDB	7318	18228
LevelDB(+Lazy Deletion)	6689	15003

RocksDB has a similar mechanism named Sub-Compaction. However, Sub-Compaction is only adopted in L_0, not all levels. Otherwise, it may create many small SSTable files. This is because RocksDB does not allow obsolete data blocks in an SSTable and fixes the SSTable size strictly. However, our design allows an SSTable to grow in size up to *MAX_FILE_SIZE*, which makes *Parallel Merging* available in Block Compaction.

C. Lazy Deletion

In the current implementation of the LSM-tree, the clean function removing obsolete SSTable files from the file system will be executed after each compaction operation. However, this approach will cause a performance drop. Because this function needs to read the working directory file and check the validity of all files, including SSTable files, log files, and other files in the working directory through a set structure (e.g., std::set in LevelDB). To deal with this problem, we propose *Lazy Deletion* on top of Block Compaction to change the triggering mechanism of the file clean function. We count the total size of all the obsolete SSTable files. When the size reaches a specific threshold (e.g., 200 MB in our experiments), we call the *DeleteObsoleteFiles* function to remove these obsolete SSTable files together. Thus, we can reduce the frequency of traversing files and the disk bandwidth use of reading the directory file. Table II shows the effectiveness of Lazy Deletion when loading different sizes of datasets. We can see that this strategy improves the performance of LevelDB on 40 GB data and 80 GB data by up to 8% and 17%, respectively.

D. Bloom Filter

Bloom filters are used to improve the read performance of the LSM-tree. LevelDB and RocksDB always construct new Bloom filters for a newly created SSTable. Once Bloom filters are created, they will become immutable. However, Block Compaction only rewrites dirty blocks and appends new data blocks at the tail of SSTables. Thus, updating an SSTable will cause the re-construction of Bloom filters, which will incur additional costs. To alleviate this problem, in Block Compaction, we reserve extra bits for Bloom filters to reduce the cost of re-constructing Bloom filters. First, when an SSTable is created and new Bloom filters are constructed, we allocate some extra bits for the new Bloom filters. When new keys are appended to an SSTable, we insert the new keys into the Bloom filters using the reserved bits. Second, as the reserved bits will incur additional space overheads, we propose to reserve different-length bits for different levels on the LSM-tree. To be more specific, for the level $L_i(1< i < N)$, the reserved bits can support inserting 40% of the keys in an SSTable, while for the last level L_N, the reserved bits can only undergo inserting 10% of the keys in an SSTable.

V. Performance Evaluation

A. System Implementation

We implemented the proposed Block Compaction and its three optimizations on LevelDB (version 1.20). LevelDB is a popular open-source key-value store adopting LSM-tree as the storage engine. It was developed by Google and written in C++. Many previous works [22], [25] on improving LSM-tree were implemented on LevelDB. In addition, we name our system BlockDB and compare it with other systems using a system-to-system comparing scheme.

Below we briefly describe how various operations are implemented in BlockDB. Similar to many LSM-tree-based engines, BlockDB supports basic API interfaces like Put, Get, Delete, Range Query.

Put. To put a key-value pair, BlockDB first inserts the key-value pair into the Memtable. A key-value pair will be flushed to an SSTable file in L_0 when a flush operation is triggered to flush the Immutable Memtable to L_0. The key-value pairs in L_0 will be merged into lower levels through compaction operations.

Get. BlockDB first searches the Memtable and the Immutable Memtable. Then, it starts to search from L_0 to the last level. Since the SSTables in L_0 may be overlapped with other SSTables, BlockDB needs to search from the newest SSTable to the oldest one. For other levels, all SSTables are sorted with non-overlapped key ranges, BlockDB only needs to search one targeted SSTable. For this SSTable, the Bloom filters of the SSTable will be checked first, followed by an search on the index block if the search key is found in the Bloom filters. Note that several index blocks may exist for one SSTable in BlockDB. However, only the last-created index block is valid to locate valid data blocks. Thus, for each SSTable, BlockDB only needs to search one index block and one data block.

Delete. The key deletion operation in BlockDB is similar to that in LevelDB and RocksDB. When a key is deleted, BlockDB inserts the key with a flag marking that it has been deleted. When the SSTable containing the deleted key is merged by a compaction operation, the deleted key will be removed from the SSTable.

Range Scan. To perform a range scan, BlockDB uses an LSM-tree iterator and positions the iterator to the starting key by calling the `seek` method. Then, it scans the given range using the `next` and `value` methods.

B. Experimental Setting

To achieve fair comparison, we equip all competitor with the same settings. First, we set the default Memtable size to 16 MB. Second, we set *level0_slowdown_writes_trigger* to 12 and *level0_stop_writes_trigger* to 16. Third, we set the block cache to 4 GB. Fourth, we enable Bloom filters for all the systems and the Bloom-filters bits per key is set to 10. Furthermore, to avoid the influence of the page cache, we open all SSTable files with the *O_DIRECT* flag and we disable compression mechanisms. By default, we set the SSTable size to 16 MB. Note that LSM-tree is sensitive to the SSTable file size (we will evaluate the impact of the SSTable size on performance in Section V-I). In addition, the size of L_0 is set to 8x SSTable size.

TABLE III
YCSB WORKLOADS.

Workload	Description
Write-Only (WO)	100% writes
Write-Heavy (WH)	80% writes and 20% reads
Balanced (RW)	50% writes and 50% reads
Read-Heavy (RH)	20% writes and 80% reads
Read-Only (RO)	100% reads

We conduct all experiments on a server equipped with two Intel-Xeon CPUs and 128 GB DDR4 memory. The operating system is Ubuntu-20 with the 64-bit Linux 4.15 kernel, and the file system is Ext4. The storage device is a 1.92 TB Intel SSD D3-S4610 Series, which has a 560 MB/s top sequential-read speed and a 510 MB/s top sequential-write speed.

We use the YCSB (Yahoo! Cloud Serving Benchmark) benchmark as the workloads. Each key is set to 32 B and each value is set to 1 KB. Following the same approach of the previous work [8], we configure the YCSB workloads with different ratios of writes, point queries, or range scans. Consequently, we construct four YCSB workloads, which are listed in Table III. The read requests in all workloads contains point queries and range scans. We will use point queries when evaluating the performance of point lookups. On the other hand, we will use range scans when evaluating the range-query performance. As shown in Table III, the ratio of read requests varies from 0%, 20%, 50%, 80%, to 100%, and all read requests satisfy a Zipfan distribution (*zipf*=0.9) by default. In addition, the write requests contains insertions and updates. Insertions mean that users put non-existing key-value pairs into the LSM-tree and updates mean that users put existing key-value pairs into the LSM-tree.

We compare BlockDB with three existing LSM-tree-based key-value stores, including LevelDB (version 1.20) [6], RocksDB (version 6.16.5) [2], and L2SM [26]. LevelDB and RocksDB are the representatives of the Table Compaction strategy. L2SM is a state-of-the-art LSM-tree-based engine proposed in 2021. In the experiments, we used the public source codes of L2SM [27].

C. Write Performance

In this experiment, we load different scales of datasets (i.e., 40 GB and 80 GB) uniformly using the write-only workload. As Fig. 5 shows, both LevelDB and RocksDB show poor write performance because they both use the Table Compaction, which causes large write amplification. L2SM adopts Table compaction in LSM-tree and suffers from computing the hotness and density of SSTables. The running time of L2SM is roughly the same as that of RocksDB, even increased by 7.7% (40 GB). BlockDB can reduce write amplification efficiently,

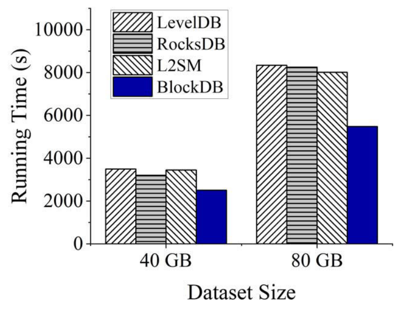


Fig. 5. Write performance of LSM-trees.

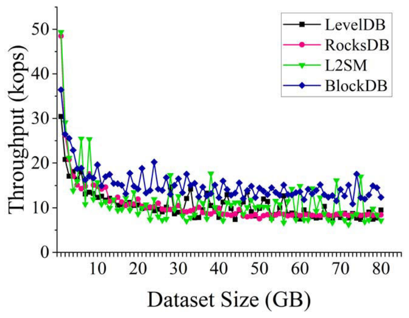


Fig. 6. Throughput of inserting 80 million key-value pairs (80 GB).

owing to its Block Compaction strategy. As a result, BlockDB performs better than other systems and decreases the running time by up to 28%, compared to LevelDB.

Note that L2SM shows poor write performance in the experiment. L2SM highly relies on the characteristics of the access pattern of keys in the workload. It works well when updates are concentrated on a small number of SSTables. However, when all key-value pairs are inserted uniformly into SSTables, it will fail to identify the hot SSTables that are updated frequently. Thus, the multi-level log in L2SM cannot benefit the LSM-tree. When the SSTable in the multi-level log is moved to the LSM-tree, L2SM also adopts the Table Compaction and needs to rewrite all the overlapped SSTables, which will cause large write amplification and worsen write performance. Also, L2SM has an extra overhead of computing the hotness and density of SSTables.

Figure 6 shows the throughput curves when inserting 80 million key-value pairs from scratch. We can see that the throughput curves of RocksDB and LevelDB are similar, indicating that they have similar write performance. On the other hand, BlockDB shows the best average write throughput among all the systems, owing to the reduction of the rewritten data blocks.

D. Write Amplification

Figure 7 shows the write amplification when ingesting different datasets. LevelDB and RocksDB use Table Compaction and show similar write amplification. Compared with LevelDB and RocksDB, BlockDB can reduce write amplification by up to 22.7% (40 GB) and 24.2% (80 GB). L2SM adopts a multi-level log, which can help reduce write amplification. However, L2SM highly relies on the writing pattern of keys. When the key-value pairs are inserted uniformly, all SSTables will have similar hotness and density, and the multi-level log in L2SM will not benefit the write amplification. In addition, L2SM also uses Table Compaction, which has similar write amplification as LevelDB and RocksDB when performing compaction. BlockDB adopts Block Compaction and only needs to write a small number of data blocks during compaction, which reduces write amplification.

Further, we measure the write amplification of each level on the LSM-tree, and the results are shown in Fig. 8. In this experiment, LSM-tree has five levels ($L_0 \sim L_4$) after inserting 40 million key-value pairs (40 GB). As L_4 triggers few compaction operations, we do not include L_4 in Fig. 8. BlockDB has the same write traffic in L_1 as LevelDB because it selects Table Compaction between L_0 and L_1, which is determined by the *Selective Compaction* strategy in BlockDB. The L2SM's write amplification in L_1 is also similar to that of LevelDB because it also uses the Table Compaction strategy. For other levels, BlockDB selects Block Compaction and reduces the write traffic by up to 42.2% (L2) and up to 34.6% (L3), compared with LevelDB.

When LSM-tree involves more levels with the increase of the dataset, the newly-inserted key-value pairs have to go through more levels to reach the targeted level. In this situation, BlockDB will have more chances to use Block Compaction than Table Compaction at middle levels, which can reduce more writes, yielding more improvements on the write amplification.

E. Space Amplification

Space amplification is a crucial factor that reflects the space usability of the LSM-tree. To evaluate space amplification, we first load 40/80 million key-value pairs and then update them uniformly. When performing updates, we monitor the space usage of the LSM-tree and record the maximum value. As shown in Fig. 9, LevelDB and RocksDB can timely remove obsolete SSTables through Table Compaction and achieve the lowest space amplification. However, L2SM needs to move SSTables to the multi-level log for postponing the updates to the LSM-tree, which incurs additional space amplification.

BlockDB will generate some obsolete data blocks, which will be collected and merged into SSTables by subsequent table compaction operations. As Fig. 9 shows, BlockDB only increases the space usage by up to 19.6% (40 GB) and 15.6% (80 GB), compared to RocksDB. Note that BlockDB mainly aims to reduce the write amplification of the LSM-tree. Compared to space amplification, write amplification will lower the performance of the LSM-tree directly because it will incur many extra I/Os. On the other hand, space amplification indirectly influences performance, which can be alleviated by the index blocks on the LSM-tree. This is because there are always index blocks associated with each SSTable. Thus, we can read data blocks by searching index blocks first. In a word, BlockDB improves the write amplification with some

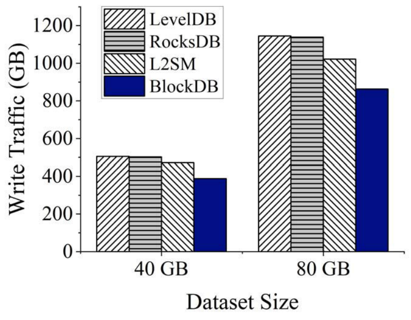


Fig. 7. Write amplification of LSM-trees.

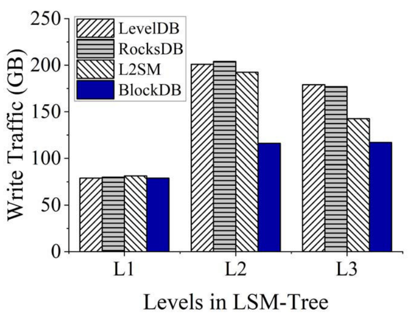


Fig. 8. Write amplification at each level.

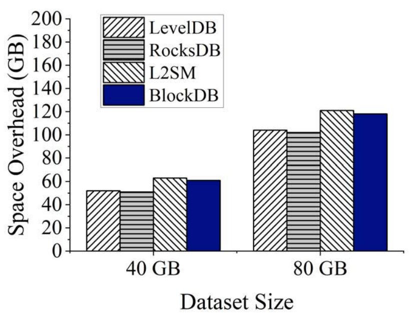
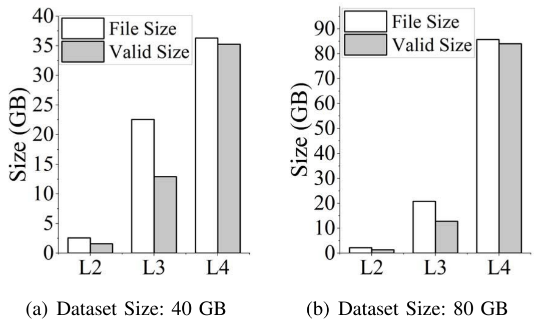


Fig. 9. Space amplification.



(a) Dataset Size: 40 GB (b) Dataset Size: 80 GB

Fig. 10. Space amplification at each level in BlockDB.

additional space amplification, but such a design can benefit the overall performance of LSM-tree-based engines. Generally, the best way to optimize the LSM-tree is to reduce both write and space amplification, and we will remain this issue as one of our future research directions.

Further, we measure the space amplification of different levels in BlockDB, and the results are shown in Fig. 10. We can see that most space amplification in BlockDB happens at middle levels, and the last level L_4 only introduces a little extra space. At middle levels, BlockDB is more likely to use Block Compaction to reduce write amplification, while for the last level, Table Compaction will be triggered more frequently, leading to the low space amplification at the last level.

F. Point Query

To evaluate the performance of point queries, we use the workloads that have different read/write ratios, including RO (read-only), RH (read-heavy), RW (write-read balanced), WH (write-heavy), and WO (write-only). The read operations are point queries, and the write operations are insertions or updates. In these workloads, point queries and updates follow a Zipfan distribution (*zipf*=0.9). In modern implementations, LSM-tree is usually equipped the block cache and Bloom filters to fast point queries. Thus, for all systems, we add a 4 GB block cache (10% of the dataset) and enable the Bloom filters (*bits_per_key* is 10).

We first load 40 million key-value pairs, and then issue 40 million requests by using 16 threads. Figure 11 shows the performance of the LSM-tree under the workloads mixing point queries and insertions. For the RO workload, LevelDB and RocksDB perform better than others because of their low read amplification. The read procedure in BlockDB is the same as that of LevelDB. Along with the increase of the ratio of insertions, the advantages of BlockDB are gradually emerging. For the RW and WH workloads, the performance improvements of BlockDB over RocksDB are 31.4% and 36.2%, respectively. In addition, L2SM does not benefit from the separated multi-level Log component. There are two reasons. First, random insertions make L2SM difficult to isolate key-value items that have a disruptive effect on the LSM-tree. Second, there are overlapped SSTables in the multi-level Log, which causes read amplification.

Figure 12 shows the performance of the LSM-tree under the workloads mixing point queries and updates. Compared with RocksDB, BlockDB can improve the performance by up to 13.4% (RH), 14.5% (RH), 20.6% (RW), and 24.2% (WH). We also evaluate all competitors with various zipf values. A larger zipf value means that the access is more skewed. The results show that, RocksDB perform better than others. For a high zipf value (0.99), the performance of BlockDB is similar to that of RocksDB. For others, BlockDB can improve the performance by up to 14.5% and 20.3%.

The number of the block cache misses are shown in Fig. 14. We can see that BlockDB has fewer block-cache misses than other competitors because Block Compaction can reduce the number of the rewritten data blocks, which alleviates the invalidation problem of the block cache. Especially for other read/write workloads, BlockDB can reduce the block-cache misses by up to 7.9% (RH), 10.9% (RW) and 10.2% (WH), respectively.

L2SM isolates the SSTables that are updated frequently to improve the performance of the LSM-tree. However, it requires that the access patterns of keys has high space locality. The best case is that all the frequently updated keys are within a small range and stored in one SSTable. On the other hand, if the frequently updated keys are uniformly distributed in all SSTables, the performance of L2SM will degrade.

G. Range Scan

To evaluate the performance of range scans, we prepare four new workloads with different scan/write ratios. Such

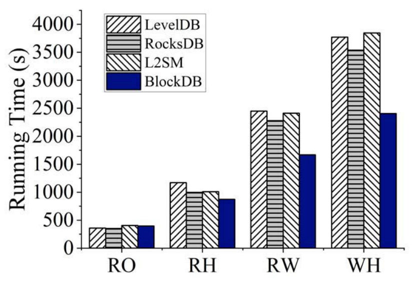


Fig. 11. Read performance (point queries + insertions).

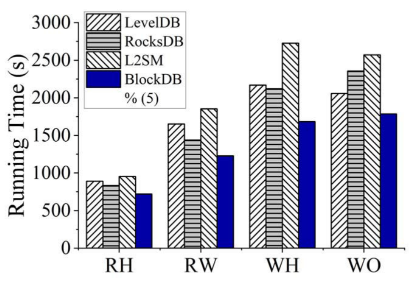


Fig. 12. Read performance (point queries + updates).

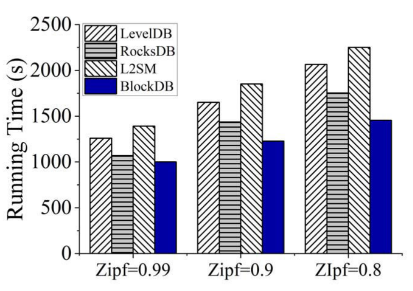


Fig. 13. Read performance under different Zipfan distributions.

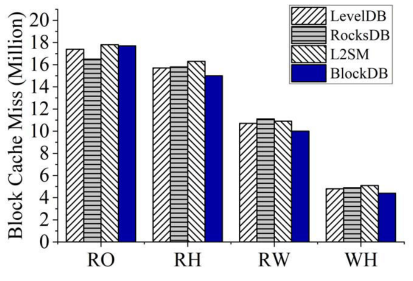


Fig. 14. Block cache misses.

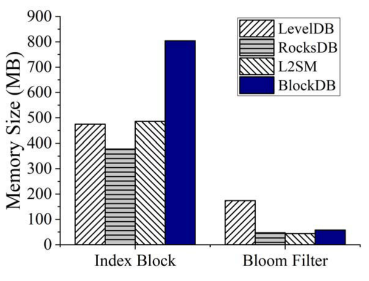


Fig. 15. Memory cost of the table cache.

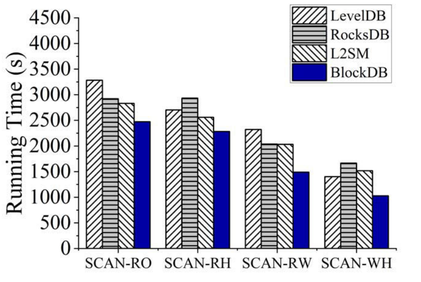


Fig. 16. Range scan performance.

a way to generate range-scan workloads is also used by previous work [8]. Consequently, we generate four range-scan workloads, which are denoted as SCAN-RO, SCAN-RH, SCAN-BA, and SCAN-WH (the definitions of Ro, RH, BA, and WH are listed in Table III). The length of scan operations is uniformly distributed between 1 and 100. The start keys of the range queries follow a Zipfan distribution (*zipf*=0.9). In addition, the write operations in all range-scan workloads are insertions. We first load 40 million key-value pairs and then issue 10 million requests using 16 threads.

Figure 16 shows the running time of each workload. We can see that BlockDB outperforms LevelDB, RocksDB, and L2SM on all workloads. Note that LevelDB supports *Seek Compaction* [25], [28]. If LevelDB detects that an SSTable has been read frequently (over a threshold), it will trigger compaction to merge the SSTable into the next level. As the start keys of range queries are distributed in the entire key range, we can infer that after performing a number of range queries, all the SSTables in upper levels, e.g., L_0 and L_1, will trigger *Seek Compaction*, resulting in the decrease of the tree levels. In our experiment, we found that the levels of LevelDB decreased from five to two after executing the SCAN-RO workload.

Meanwhile, as L2SM and BlockDB are built on top of LevelDB, they also support *Seek Compaction*. However, RocksDB does not use *Seek Compaction*, meaning that range queries will invoke no compaction. Therefore, the LSM-tree height in RocksDB remains stable in all range-scan experiments. Accordingly, RocksDB needs to traverse more levels than BlockDB when answering range queries, which leads to its worse performance than BlockDB. On the other hand, although LevelDB and L2SM can trigger compaction to lower the tree levels (which is friendly to range queries), the invoked table compaction will incur additional table-compaction cost and the disk-bandwidth contention with range queries, which makes LevelDB and L2SM perform worse than BlockDB.

H. Memory Cost

The memory consumption of the LSM-tree consists of following parts: (1) Memtable and Immutable Memtable; (2) Block Cache; (3) Table Cache. Among these parts, the table cache limits the number of opened SSTable files and stores index blocks and Bloom filters. As Memtable and the block cache are with fixed sizes, we focus on the memory cost of the table cache.

Figure 15 shows the memory cost of the table cache. We can see that BlockDB uses more memories for the index blocks than others. This is because BlockDB may create many small data blocks (less than 4 KB), which increase the total size of the index blocks. To solve this issue, we can enlarge the block size to reduce the memory consumption of the index blocks. For Bloom filters, LevelDB adopts blocks-based filters that need to store the offset of each data block. Thus, LevelDB incurs large memory consumption. RocksDB, L2SM, and BlockDB all use table-based filters to skip the SSTables that do not contain the search key. The filters of BlockDB incur more memory overheads than RocksDB because of the additional space costs of the reserved bits in BlockDB.

I. Varying the SSTable Size

In this experiment, we compare the performance of all systems under different SSTable sizes. In RocksDB, the Memtable size is the same as the SSTable size, and the L_1 size is the

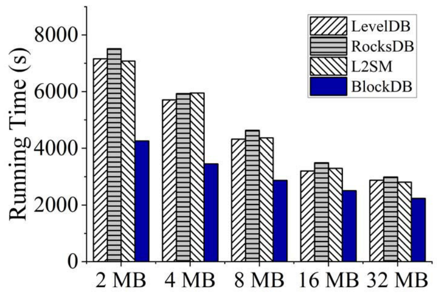


Fig. 17. Running time with different SSTable sizes (both L_0 and L_1 are eight times the SSTable size).

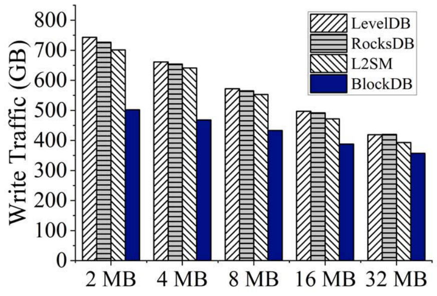


Fig. 18. Write amplification with different SSTable sizes (both L_0 and L_1 are eight times the SSTable size).

same as the L_0 size by default. To make the comparison fair enough, we also use the same settings for all the systems, i.e., the Memtable size equals the SSTable size, and L_1 has the same size as L_0, which is eight times the SSTable size. The size ratio between L_i and L_{i+1} is set to 10. Then, we vary the SSTable size and load 40 million key-value pairs to compare the running time and write amplification of all systems.

Figures 17 and 18 show the running time and write amplification, respectively. With the increase of the SSTable size, both the write performance and write amplification are improved. The reasons are two-fold. First, larger SSTables increase the capacity of L_0, which can decrease the tree height and reduce the write amplification caused by insertions. Second, larger SSTables will increase the number of key-value pairs involved by a compaction operation, which can help reduce the compaction frequency. As shown in the figures, BlockDB can reduce the running time by up to 43.6% and the write traffic by up to 32% when the SSTable size varies.

As the size of L_1 is set to eight times the SSTable size for all systems, we can infer that small SSTables will lead to a high LSM-tree. For example, suppose that the LSM-tree has six levels when the SSTable size is 2 MB, it may have only five levels when the SSTable size is increased to 8 MB because a larger SSTable can hold more key-value pairs.

Note that using small SSTables in LevelDB and RocksDB cannot reduce write amplification because they always need to rewrite all the SSTables in L_{i+1}, which are overlapped with the selected SSTable in L_i.

Unlike LevelDB and RocksDB, BlockDB only needs to write back the affected data blocks in L_{i+1} rather than the overlapped SSTables. Even if SSTables are small enough, e.g., only have one data block containing two key-value pairs, BlockDB can also reduce more write amplification than LevelDB and RocksDB. For example, if the selected SSTable file in L_i has two keys, e.g., "1" and "100", and the number of overlapped SSTables in L_{i+1} is 10, whose keys range from "2" to "99", LevelDB and RocksDB have to rewrite 10 SSTables during compaction, while BlockDB only needs to write two blocks because key "1" is only overlapped with one data block and so is key "100". Although LevelDB and RocksDB support trivial compaction [29] (i.e., if an SSTable in L_i does not overlap with any SSTables in L_{i+1}, no SSTable in L_{i+1} needs to be rewritten during compaction), it does not affect the comparative results shown in Fig. 17 and Fig. 18 because BlockDB also supports trivial compaction.

VI. Conclusions and Future Work

In this paper, we addressed the write amplification and the block-cache invalidation problems on the LSM-tree. We first proposed a new compaction scheme called Block Compaction that adopted a block-based merging policy to perform compaction operations. Compared to the traditional Table Compaction using an SSTable-grained merging policy, Block Compaction can avoid a great number of write I/Os and meanwhile alleviate the block-cache invalidation problem. Further, we proposed three optimizations to Block Compaction, which can accelerate the merging process and reduce the side effects of Block Compaction. We presented a cost analysis to demonstrate that Block Compaction can reduce write amplification without sacrificing read performance. Finally, We implemented Block Compaction and its optimizations on LevelDB (version 1.20), forming a new key-value store called BlockDB. We conducted extensive experiments on the YCSB workloads to compare BlockDB with three LSM-tree-based key-value stores, including LevelDB, RocksDB, and L2SM. The results showed that BlockDB can reduce write amplification and running time significantly compared to its competitors. In addition, it can maintain high performance for point lookups and range scans.

In the future, we will consider a few research directions. First, we will explore Block Compaction on new storage devices, such as Zoned Namespaces (ZNS) SSDs [30] and persistent memory [31]. There are several challenges in the key-value stores on new storage devices [10], [32], [33], and it is potential to use Block Compaction to develop new solutions. Second, we will investigate efficient methods to reduce the space amplification caused by Block Compaction. Finally, we will consider improving the read performance of BlockDB, e.g., using a learned prefetching policy for the block cache [14] or building hotness-aware LSM-trees [34].

Acknowledgement

This work is partially supported by the National Science Foundation of China (No. 62072419) and Huawei Technologies Co., Ltd. Peiquan Jin is the corresponding author.

References

[1] P. E. O'Neil and et al., "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[2] Facebook. (2021) RocksDB. [Online]. Available: https://rocksdb.org/

[3] Apache Software Foundation. (2021) Apache HBase. [Online]. Available: https://hbase.apache.org

[4] A. S. Foundation. (2021) Apache Cassandra. [Online]. Available: https://cassandra.apache.org

[5] S. Idreos and M. Callaghan, "Key-value storage engines," in *SIGMOD*, 2020, pp. 2667–2672.

[6] Google. (2021) LevelDB. [Online]. Available: https://github.com/google/leveldb

[7] A. Fevgas, L. Akritidis, P. Bozanis, and Y. Manolopoulos, "Indexing in flash storage devices: a survey on challenges, current approaches, and future trends," *VLDB J.*, vol. 29, no. 1, pp. 273–311, 2020.

[8] Y. Chai, Y. Chai, X. Wang, H. Wei, N. Bao, and Y. Liang, "LDC: A lower-level driven compaction method to optimize SSD-oriented key-value stores," in *ICDE*, 2019, pp. 722–733.

[9] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in log-structured merge key-value stores," in *USENIX ATC*, 2019, pp. 753–766.

[10] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *USENIX ATC*, 2020, pp. 17–31.

[11] C. Luo and M. Carey, "On performance stability in LSM-based storage systems," *Proc. VLDB Endow.*, vol. 13, no. 4, pp. 449–462, 2019.

[12] P. Jin, J. Li, and H. Long, "DLC: A new compaction scheme for lsm-tree with high stability and low latency," in *EDBT*, 2021, pp. 547–557.

[13] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang, "LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *ICDCS*, 2017, pp. 68–79.

[14] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang, "Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 1976–1989, 2020.

[15] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *SoCC*, 2018, pp. 477–489.

[16] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *USENIX ATC*, 2018, pp. 1007–1019.

[17] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *FAST*, 2020, pp. 209–223.

[18] F. Wu, M. Yang, B. Zhang, and D. H. C. Du, "AC-Key: Adaptive caching for LSM-based key-value stores," in *USENIX ATC*, 2020, pp. 603–615.

[19] C. Luo and M. J. Carey, "Breaking down memory walls: Adaptive memory management in LSM-based storage systems," *Proc. VLDB Endow.*, vol. 14, no. 3, pp. 241–254, 2020.

[20] C. Luo and M. Carey, "LSM-based storage techniques: a survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, 2020.

[21] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees," in *FAST*, 2013, pp. 17–30.

[22] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *FAST*, 2016, pp. 133–148.

[23] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *CIDR*, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[24] F. Pan, Y. Yue, and J. Xiong, "dCompaction: Speeding up compaction of the LSM-tree via delayed compaction," *J. Comput. Sci. Technol.*, vol. 32, no. 1, pp. 41–54, 2017.

[25] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *SOSP*, 2017, pp. 497–514.

[26] K. Huang, Z. Jia, Z. Shen, Z. Shao, and F. Chen, "Less is more: De-amplifying i/os for key-value stores with a log-assisted lsm-tree," in *ICDE*, 2021, pp. 612–623.

[27] L2SM. [Online]. Available: https://github.com/ericaloha/L2SM

[28] Google. DB Iterator. [Online]. Available: https://github.com/google/leveldb/blob/main/db/db_iter.cc

[29] Facebook. Trivial Compaction in RocksDB. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Compaction-Trivial-Move

[30] G. Choi, K. Lee, M. Oh, J. Choi, J. Jhin, and Y. Oh, "A new LSM-style garbage collection scheme for ZNS SSDs," in *HotStorage*, 2020, pp. 1–6.

[31] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for OLAP workloads," in *SIGMOD*, 2021, pp. 339–351.

[32] P. Jin, X. Zhuang, Y. Luo, and M. Lu, "Exploring index structures for zoned namespaces SSDs," in *IEEE BigData*, 2021, pp. 5919–5922.

[33] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid PMEM-DRAM key-value store," *Proc. VLDB Endow.*, vol. 14, no. 9, pp. 1544–1556, 2021.

[34] Y. Wang, P. Jin, and S. Wan, "Hotkey-lsm: A hotness-aware lsm-tree for big data storage," in *IEEE BigData*. IEEE, 2020, pp. 5849–5851.