

On Reducing Space Amplification with Multi-Column Compaction in Apache IoTDB

Chenguang Fang Tsinghua University fcg19@mails.tsinghua.edu.cn

Xiangdong Huang Tsinghua University huangxdong@tsinghua.edu.cn Zijie Chen Tsinghua University chenzj23@mails.tsinghua.edu.cn

Chen Wang Tsinghua University wang_chen@tsinghua.edu.cn Shaoxu Song* Tsinghua University sxsong@tsinghua.edu.cn

Jianmin Wang Tsinghua University jimwang@tsinghua.edu.cn

ABSTRACT

Log-structured merge trees (LSM-trees) are commonly employed as the storage engines for write-intensive workloads in modern time series databases including Apache IoTDB. Following append-only principle, LSM-trees can handle intensive writes and updates, but consequently suffer high space amplification (SA). To reduce SA in LSM-tree, compaction is triggered periodically to reorganize a large number of immutable files on disk to eliminate redundancy. This issue is further complicated in the Internet of Things (IoT) scenarios, where frequent out-of-order data insertions and data updates introduce duplicated keys, obsolete values and overlapping bitmaps in multi-column data, thereby exacerbating SA concerns.

To mitigate SA in such contexts, this paper presents a Multi-Column Compaction (MCC) strategy in Apache IoTDB, an opensource time series database utilizing LSM-tree architecture and supporting multi-column storage. We take into consideration both the separate insertions (out-of-order data) and updates of multicolumn data, and analyze the hardness of selecting proper files with the maximum space reduction in compaction. We then propose a heuristic method designed to improve the file selection, thus reducing SA. To enhance the efficiency of this approach, we further devise File Prefetcher and Compaction Cache. The proposed MCC has been implemented in Apache IoTDB. Experimental results demonstrate that our proposed MCC achieves better performance in reducing space amplification.

PVLDB Reference Format:

Chenguang Fang, Zijie Chen, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. On Reducing Space Amplification with Multi-Column Compaction in Apache IoTDB. PVLDB, 17(11): 2974 - 2986, 2024. doi:10.14778/3681954.3681977

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/column-compaction/column-compaction.

1 INTRODUCTION

Apache IoTDB¹ is an open-source time series database with high performance for IoT applications [41]. It employs log-structured merge trees (LSM-trees) [32] as storage engine. LSM-trees can efficiently manage write-intensive workloads by following an appendonly principle to handle writes, making it naturally suitable for time series databases. However, this consequently leads to high space amplification (SA) [30, 36]. To mitigate SA in LSM-tree, compactions are periodically triggered to reorganize the immutable files on disk. Compactions aim to eliminate redundancy, thereby fundamentally improving the performance of the database, including reducing SA. Nevertheless, the workloads are further complicated in the Internet of Things (IoT) scenarios, especially with frequent out-of-order data insertions and data updates, introducing new challenges to IoTDB in terms of effectively managing SA.

1.1 Challenges in Apache IoTDB

While SA issues are prevalent, this section highlights unique challenges in Apache IoTDB regarding SA, which motivate the intuition of considering multi-column compaction in this study.

Multi-column storage with bitmaps. Apache IoTDB supports multi-column storage, a feature that is also prevalent in several LSMtree based stores, e.g., Cassandra [1] and Apache HBase [2]. This structure allows multiple columns to share a key column, reducing the space for storing redundant keys. Owing to unsynchronized data ingestions and updates, null values are inevitable in multicolumn data. It then requires extra space for representing null values. To manage null values, unlike HBase [2] and InfluxDB [3] that record indexes to mark values, IoTDB (1) supports multicolumn storage and (2) employs bitmaps to manage null values [26] for multi-column data. The bitmaps track the position of multicolumn values associated with the shared key column (i.e., indexes the position of the values). This facilitates contiguous storage of values within files for space efficiency. However, the bitmaps might overlap with each other when the data and updates are delayed. Figure 1 illustrates the real-world data example in the SAMSUNG [42] dataset, which encounters data delays as shown in Figure 1(a). Figure 1(b) shows the corresponding files in Apache IoTDB, where multi-column data are collected by sensors with severe delay issues. The white cells represent null values corresponding to '0' in bitmap. Different colors of cells indicate the arrival order of each data, and these cells are actually recorded with '1' in bitmap. The files

^{*}Shaoxu Song (https://sxsong.github.io/) is the corresponding author. This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097. doi:10.14778/3681954.3681977

¹https://iotdb.apache.org/



Figure 1: Real-world example from SAMSUNG [42] dataset.

(SSTables) in Level 0 of LSM-tree in Figure 1(b) are flushed to the disk with arrival time. When certain data of f_1 and f_2 are delayed in other files, the corresponding cells with identical keys are recorded repeatedly. As a result, the files in Level 0 exhibit duplicated keys under the design of multi-column storage, leading to high SA.

Our previous study [26] has improved the storage efficiency during the flushing stage by considering the bitmaps. Nevertheless, the issue of the bitmap still persists during the compaction stage, which motivates us to consider the compaction strategy for multicolumn data in IoTDB.

Out-of-order insertions and updates. Out-of-order insertions (i.e., delayed data) and updates are prevalent especially in IoT applications. For instance, data from various sensors on mobile devices are transmitted to a data collection engine for storage via networks. However, network latency [42] can cause out-of-order data arrivals resulting in different degrees of delay. Figure 1(a) illustrates the data delays in the SAMSUNG [42] dataset collected from real-world mobile devices. The x-axis denotes the arrival order of data points. The fluctuating time of the arrived data points means that they are received out-of-order. These issues challenge the design of time series databases. Our previous works focus on mitigating the impact of the delayed issues in IoTDB, regarding data sorting [45] and write amplification (WA) [29]. However, the issue of SA in LSM-tree is still severe.

Recall that compaction is periodically triggered, selecting some files to sort-merge and compacting them to the next level. The delayed data ingestion and updates pose challenges to compaction, especially in the stage of selecting files. Existing compaction strategies, such as Round-robin strategy [22] and Oldest strategy [4, 38], select files in a predefined order, but neglect the delay and update issues of multi-column data. Even worse, SA is often neglected in the design of existing LSM-tree stores and optimization techniques [30]. For instance, to conduct compaction in Figure 1(b), the traditional Oldest policy chooses f_1 , f_2 and f_3 to compact, as shown in Figure 1(c). However, f_4 has more overlapping data with f_1 and f_2 than f_3 , indicating that there is an opportunity for improving space usage and mitigating SA.

1.2 Motivation

To this end, it is necessary to devise a file selection strategy tailored to multi-column compaction in Apache IoTDB. Intuitively, we can merge the files with their delayed and updated values towards lower SA. The primary avenues for space reduction include the following aspects: (1) Duplicated keys: For multi-column data, duplicated keys are common when data from different columns do not arrive simultaneously. For example, the files f_1 and f_2 in Figure 1(b) have duplicated keys (2,3 and 4). Compacting these files would eliminate such redundancies. (2) Obsolete values: Data updates may also generate obsolete values and result in high SA. The values of key 2 in Figure 1(b) provide an example of such data updates. (3) Overlapping bitmap: The space cost of bitmaps is also essential for multi-column data. For both delayed and updated data in Figure 1(b), we can merge them for a more complete bitmap. Hence, for the files in Figure 1(b), a better choice is to compact f_1 , f_2 and f_4 , given their overlapping keys and complementary columns. Figure 1(c) presents the results. In this sense, the extra space cost for keys, values and bitmaps is saved.

Following the intuition, we introduce MCC, a compaction strategy in Apache IoTDB. By taking into consideration both the separate insertions (out-of-order data) and updates of multi-column data, MCC has the ability to reduce the space cost during the file selection of the LSM-tree compaction. The experiments demonstrate that MCC achieves better performance in reducing SA.

1.3 Contribution

Our major contributions in this study are summarized as follows.

(1) We formalize the problem of selecting files for multi-column compaction and analyze the NP-completeness of the problem in Theorem 1 in Section 4.2. We also devise directed acyclic graph (DAG) constraint. It guarantees different versions of data are correctly compacted in LSM-tree, ensuring the correctness of the compaction.

(2) We devise a Multi-Column Compaction (MCC) strategy tailored to multi-column storage in IoTDB in Section 5. It adopts a heuristic algorithm of file selection for better performance. File Prefetcher and Compaction Cache are also dedicatedly devised for efficient decisions regarding file selection in Section 6.

(3) We deploy the proposed MCC in Apache IoTDB [4], an opensource time series database. The code is included in the official GitHub repository of the system [5].

(4) We conduct experiments to validate that MCC successfully reduces the SA in IoTDB. Moreover, we further implement the proposed MCC in RocksDB [6] to show its applicability to other LSM-tree implementations.

2 BACKGROUND

This section first provides some background on LSM-trees and then highlights the importance of the compactions. Finally, it introduces the LSM-tree in Apache IoTDB.



Figure 2: System overview of MCC.

2.1 LSM-trees

LSM-trees are commonly employed as storage engines in NoSQL stores. Compared to the original storage engines, such as B⁺-trees, LSM-trees follow an out-of-place update strategy to handle write-intensive scenarios [30] and reduce random I/Os. LSM-trees maintain a memory buffer for latest data ingestion, namely MemTable. When the MemTable is full, LSM-tree sorts the keys in the MemTable and then flushes it into disk. It then forms immutable SSTables in the disk. The SSTables in the disk are organized by levels and the sizes of the levels increase exponentially following a size ratio.

2.2 LSM-tree Compaction

To reorganize data in the disk, LSM-trees leverage compactions, when the size of some level exceeds a predefined threshold. By compactions, LSM-trees sort-merge data from lower levels to higher levels and create new SSTables. Compaction is crucial in LSM-tree designs. It not only reduces the space cost but also accelerates the queries [18]. Leveling and tiering are the most common compaction layouts for LSM-trees [38]. In the leveling layout, the keys in the same level are all sorted. When compaction is triggered, the files that are written to the next level are then sorted with all files that overlap in keys, as utilized in LevelDB [7] and RocksDB [6]. Tiering layout, instead, only generates a new file in the target level without disturbing other files. It treats the files in an LSM-tree level as sorted runs and different runs could overlap in keys. With such design, tiering is more efficient in data insertion and updates but fails in space cost and lookups. Tiering layout is also deployed by NoSQL databases such as Cassandra [1] and HBase [2].

2.3 LSM-tree in Apache IoTDB

Apache IoTDB, an open-source time series database, employs LSMtree as storage engine. Specifically, IoTDB adopts a tiering layout. As introduced in Section 2.2, tiering facilitates significantly fast data ingestion at the expense of high SA. This trade-off is particularly relevant in the application of IoTDB to IoT scenarios, where highfrequency data collection results in vast amounts of data insertions. As a result, while tiering layout enables IoTDB to handle extremely large-scale data inputs and updates, it is also urgent to mitigate SA problem in the LSM-tree of IoTDB.

Table 1: Notations.

Symbol	Description
F	the selected collection of the files
C(f)/C(F)	cost of a file/a set of files
k(f), v(f), b(f)	the space costs of keys, values and bit maps in \boldsymbol{f}
c^k, c^v, c^b	size parameters of keys, values and bitmaps
V_j, f_{v_i}	the j -th column of values and bitmaps
Ť	the collection of all the files in a Level
M	the number of allowed merging files
$Merge(\cdot)$	sort-merging of a collection of the files
G = (V, E)	a DAG with the node set V and edge set E
${\mathcal D}$	the DAG constraint

3 SYSTEM OVERVIEW

This section gives a brief introduction to the proposed MCC in Apache IoTDB. Figure 2 outlines the system.

The common delays of the data insertion and massive updates in IoT workloads could result in high SA for multi-column data. Our MCC is devised based on the tiering layout LSM-tree which is adopted in IoTDB. As shown in Figure 2, MCC mainly works in the Compaction Selector, aiming to select the files with the most space reduction. We formalize the multi-column compaction problem in Section 4.1 and prove its NP-hardness. Next, in Section 5, we devise a file selection engine, equipped with a heuristic approach. The heuristic approach builds a cost model that considers the characteristics of multi-column data to achieve lower space cost.

To accelerate the selection approach, we devise File Prefetcher and Compaction Cache to accelerate both the computation part in Section 6. For the File Prefetcher, the proposed heuristic approach requires scanning files for information including keys and bitmaps. However, the native file readers in existing databases need to read the whole file, i.e., deserializing all the keys, values and bitmaps. To avoid deserializing the values to achieve higher efficiency, we devise File Prefetcher to only consider the keys and the bitmaps during file selection. Compaction Cache is also introduced to avoid unnecessarily repeated computations.

4 MULTI-COLUMN COMPACTION PROBLEM

In this section, we first formalize the *multi-column compaction problem* (Problem 1). Next, we prove the NP-hardness of Problem 1. We then introduce DAG constraint for updates to guarantee the correctness of the LSM-tree semantics. Some frequently used notations are summarized in Table 1.

4.1 Problem

To comprehensively define our multi-column compaction problem, we start from the definition of the multi-column data and its space cost and then formally define the compaction problem. Finally, we discuss the computation of the space cost.

4.1.1 Multi-column Data Organization. We first devise the cost function C, which computes the space cost of a file. Given a file f which stores multi-column data, its space cost is composed of three parts, including keys, values and bitmaps. The bitmaps are



Figure 3: File organization with a bitmap, where $\theta = 3$.

necessary and specially designed to mark the null values and associate the multi-column data with keys. With the indexing of the bitmaps, only a shared key column is required for multiple columns. In particular, the non-null values could be stored continuously to be efficiently encoded and compressed. Since the metadata usually takes fixed but limited space, for simplicity, we do not involve the space cost of the metadata in our problem.

Therefore, the space $\cot C(f)$ of a given file f for multi-column data is defined as follows:

$$C(f) = c^k k(f) + c^v v(f) + c^b b(f)$$
(1)

where $c^k k(f)$, $c^v v(f)$ are the space costs the of keys and values in f, respectively, and $c^b b(f)$ is the space cost of the bitmap. In addition, c^k , c^v and c^b are size parameters that depend on different database settings and data types.

To be specific, k(f) computes the number of the unique keys in f and then multiplies with the space cost c^k of each key. v(f) is the number of the values and c^v denotes the space cost for each value. For b(f), it denotes the total bits in a bitmap, which have the same shape as the values. Analogously, c^b denotes the space cost for one bit. However, we also note that, in the column-oriented databases, if a given number θ of consecutive bits (i.e., a bit vector) in the bitmap are all complete, the vector of this bitmap could be neglected, i.e., it does not require space for storage. Therefore, we have the following definition of b(f):

$$b(f) = \sum_{v \in \{V_1, V_2, \dots, V_m\}} \sum_{y=1}^{\lceil \frac{|f_0|}{\theta} \rceil} \sigma(f_v[y\theta : (y+1)\theta])\theta$$

where $f_v[a:b]$ denotes the bit vector of the column f_v from *a*-th to *b*-th elements. $\sigma(f_v[a:b]) = 0$ if all bits in $f_v[a:b]$ are 1 and otherwise $\sigma(f_v[a:b]) = 1$.

Example 1. Figure 3 presents an example of file organization in IoTDB with bitmap. The space cost C(f), composed of three parts including keys, values and bitmaps, where $\theta = 3$. It contains totally three keys and three columns with a missing value at V_2 of ID = 3. Therefore, we have k(f) = 3, v(f) = 8 in terms of keys and values. For the bitmap, it goes through all values to check the null values. Since $\theta = 3$, the bit vector in columns V_1 and V_3 are complete thus omitted, while the bit vector of (1, 1, 0) in V_2 still remains, i.e., $b(f) = \theta = 3$. Following Formula 1, the space cost C(f) of f could be computed.



Figure 4: An example of multi-column compaction (M = 3). (a), (b) and (c) provide three options with DAGs.

4.1.2 Selecting Files for Multi-column Compaction. In addition, we also define the merge function Merge(F), denoting the sort-merging of a collection of files into a merged file, where $F = \{f_1, f_2, ...\}$. By sort-merging the multiple columns, the same keys are merged or updated according to their bitmaps, different keys are sorted in the meantime. Therefore, C(Merge(F)) denotes the space cost of the merged file *F*. For simplicity, we will use C(F) instead of C(Merge(F)) when the context is clear.

To comprehensively evaluate the space cost reduction of the compaction and minimize the space cost, we further define $\Delta C(F)$, denoting the space reduction of merging *F*:

$$\Delta C(F) = \sum_{f \in F} C(f) - C(F)$$
⁽²⁾

To be specific, the space reduction mainly comes from: (1) the elimination of the redundant keys (including both delay insertions and updates), (2) the updates of the stale data values, and (3) the removed complete bitmap vectors.

We then consider the multi-column compaction problem. The compaction is triggered when the size of a specific level surpasses a given threshold. Next, a specific number of files are chosen to be merged and moved to the next level. While existing methods consider different dimensions of choosing the files (e.g., Roundrobin or choosing the oldest files), we focus on the defined cost model to solve the compaction problem for lower space cost.

PROBLEM 1. (Multi-column compaction problem). Let \mathcal{F} denote the collection of the files in the given level, and M denote the allowed number of merging files. We aim to choose the proper merging files that lead to the lowest space cost, i.e., the maximum space reduction.

$$\max \Delta C(F)$$

s.t.|F| = M,
F $\subseteq \mathcal{F}$.

Example 2. Figure 4 presents an example of selecting compaction files. \mathcal{F} is composed of files from f_1 to f_4 and the color denotes the

arrival time as introduced in Figure 1. To decide M = 3 files for compaction, Figures 4(a) and (b) provide two options of merging f_1, f_2 and f_3 or f_1, f_2 and f_4 . According to the problem, merging f_1, f_2, f_4 eliminates duplicated keys and obsolete values (with *time* = 2, 3, 4, 5) and overlapping bitmap. Compared to merging f_1, f_2 and f_3 , the space reduction, i.e., $\Delta C(F)$ is more significant. Hence, f_1, f_2 and f_4 are selected to be merged.

4.1.3 Computation of the Costs. Finally, we present the methodology for computing the cost. As aforesaid, we define C(F) as the space cost of the merged file F and $\Delta C(F)$ as the space reduction achieved by merging F. That is, our approach mainly focuses on computing the space reduction $\Delta C(F)$ of merging files in F. Recall that the space reduction is related to three aspects: redundant keys, updated data value and complete bitmap vectors. Hence, the keys and bitmaps are essential for this computation. In particular, the bitmaps could provide information about the presence of updated data values by indexing their positions. Consequently, we compute the cost by fetching the keys and bitmaps of two files.

However, there still remain concerns regarding the implementation of this strategy: (1) As the values and bitmaps are stored consecutively in the file and typically deserialized together, is it feasible to fetch only the bitmaps without deserializing the values for improved efficiency? (2) How does such computation affect other crucial metrics like system throughput and resource utilization?

To address the first concern, in Section 6.2, we develop a dedicated file prefetcher, which only accesses keys and bitmaps. This allows for more efficient processing without the need to deserialize the values. For the second concern, we evaluate throughput and resource utilization in Section 7.2.6. While MCC incurs extra disk I/O due to additional bitmap reads during the selection algorithm, this trade-off is considered acceptable when weighed against the remarkable gains of MCC in SA. Moreover, it does not impact the throughput or lead to clearly increased CPU and memory usage. We thus devise MCC to further reduce SA.

4.2 Hardness

Unfortunately, we find the Multi-Column Compaction Problem is generally hard.

THEOREM 1. The Multi-Column Compaction Problem (Problem 1) is NP-hard.

Proof sketch. In summary, we can build a reduction from *set covering* problem [21], one of the Karp's 21 NP-complete problems, to the decision version of Problem 1, thus showing the NP-hardness of Problem 1. Please see full proofs in [8].

4.3 DAG Constraint for Updates

Problem 1 outlines the multi-column compaction problem. Nevertheless, we need to further consider the LSM-tree correctness semantics for updates. To be specific, LSM-tree updates data out of place, thus multiple versions of data may exist in different files. If we select files arbitrarily without considering the versions, it is possible to compact files from a newer file into an older file while skipping files that were created in-between. By compacting them into the higher level, the more recent versions of data could not be accessed by a query, since the in-between files in the lower level are first encountered and returned.

Therefore, we use a directed acyclic graph (DAG) to model the files with updates to ensure the correctness. The DAG is defined as G = (V, E), where V is the set of the nodes and E is the set of the edges. Hence, a node v corresponds to a file f and an edge from v to v' represents that f' (corresponding to v') contains some updated values of f (corresponding to v). We also define the path where each node is connected by a direct edge to its successor [16]. A node v' is reachable from another v if there exists a path from v to v'. $anc_G(v)$ is defined as the set of nodes in G that can reach v, i.e., the ancestors of v. For simplicity, we also use $anc_G(f)$ to denote the files corresponding to $anc_G(v)$. Hence, to ensure the in-between files are not neglected, we propose DAG constraint.

DEFINITION 1. (DAG constraint). Given the file collection in the same level denoted by \mathcal{F} , and the corresponding DAG of \mathcal{F} denoted by G, we say a subset $F \subseteq \mathcal{F}$ satisfying the DAG constraint \mathcal{D} , if $\forall f \in F$, $anc_G(f) \subseteq F$, denoted by $F \Rightarrow \mathcal{D}$.

PROBLEM 2. (Multi-column compaction problem with DAG constraint). Let \mathcal{F} denote the collection of the files in the given level, and let M denote the allowed number of merging files. We aim to choose the proper merging files that lead to the lowest space cost, i.e., the maximum space reduction, and the DAG constraint \mathcal{D} is satisfied.

$$\begin{split} \max \Delta C(F) \\ s.t.|F| &= M, \\ F \subseteq \mathcal{F} \land F \Longrightarrow \mathcal{D}. \end{split}$$

PROPOSITION 2. The DAG constraint ensures the correctness of the file selection strategy in LSM-tree with tiering-layout.

Please see full proofs in [8].

5 METHOD

In this section, we focus on the solution to the multi-column compaction problem. Regarding the NP-hardness in Theorem 1, we develop a heuristic algorithm for the problem.

The heuristic algorithm is based on simulated annealing. Given the set of files \mathcal{F} in a level, we aim to find a subset *F* with |F| = Mthat tries to minimize the space reduction, i.e., $\Delta C(F)$. A greedy initialization algorithm is first employed. Next, a specific number of files are iteratively removed from the current solution and the same number of files will be added, which tries to improve the space reduction.

5.1 Initialization

First, the algorithm starts from choosing M files as an initial solution S. In each step, we select a file that mostly increases the space reduction ($\Delta C(F)$), i.e., in a greedy way. Such strategy locally maximizes the space reduction, and gives an initial solution F_{init} for updating phase. After initialization, we then turn to updating phase to further improve the selection. In the initialization phase, the computation of the space reduction follows the procedure introduced in Section 4.1.3.

5.2 Updating

In the updating phase, the algorithm iteratively updates the current solution to enlarge the space reduction. In short, starting from the F_{init} , the algorithm tries to search for the solution with higher space reduction. In each iteration, a file is selected to be removed from the current solution F with |F| = M, and another file is then selected to be added to F, generating another solution F'. F' is then decided to replace F or not according to its current space reduction.

We will first introduce the removing policies. To remove *d* files from the current solution, where d < M is a given parameter. Although it is a common strategy to randomly remove files from *F*, inspired by [20] which is designed for set packing problem, we adopt two removing policies:

Remove Policy I. The *d* files that have the least overlapping keys and values are removed. This policy tends to remove those files that seem to have less impact to space reduction. Such removal could possibly remove the infeasible files and increase the space reduction with a better choice. However, such a strategy might result in cycling when the solution falls into local optima. Hence a random removing policy is then introduced.

Remove Policy II. The *d* files are randomly chosen from *F*. This policy randomly chooses the file to remove. While it is less possible to generate a better solution, it is still essential to avoid cycling and local optima as aforementioned.

Empirically, Remove Policy I is adopted twice every three iterations, and Remove Policy II is adopted once, to guide the algorithm to find an optimal solution.

Next, we explain how the files are added into F, i.e., how we generate another solution F' based on the F by removing and adding files. Two heuristics are considered for adding files, starting from the dimensions of files and keys, respectively.

Add Policy I. Add Policy I starts from the files. For each file, we compute how it could increase the space reduction by inserting it into F, which is similar to the initialization strategy. The files are then sorted according to the increases they could bring to the solution in decreasing order, and the first d files are selected.

Add Policy II. Add Policy II starts from the keys. In each iteration, it randomly finds a key in existing files in *F*, and then searches for the files that cover the key. Among these files, the one that increases the space reduction (i.e., with the largest $\Delta C(F)$) mostly will be added to *F*.

While Add Policy II might not result in a local optimum immediately, it is meaningful to consider the key dimension to increase the space reduction, which might lead to a global optimum. Add Policy I and Add Policy II are adopted alternatively in the updating phase. Both policies will generate a feasible solution F' with C(F'). Following the idea of simulated annealing, F' will replace F either C(F') > C(F) or with a probability that is related to both iteration number and their cost differences. Formally, we have the following update rule:

$$F = \begin{cases} F', & \Delta C(F') > \Delta C(F) \lor p < \exp\left(-\frac{\Delta C(F) - \Delta C(F')}{t_k}\right) \\ F, & \text{otherwise} \end{cases}$$

Algorithm 1: MCC (\mathcal{F} , M, d, k, t)

where p = random(0, 1) is a random variable, t_k denotes the temperature in the *k*-th iteration, having $t_{k+1} = \lambda t_k$, i.e., shrinking with a parameter $\lambda \in (0, 1)$.

Example 3. (Example 2 continued). Figure 4 presents an example of the proposal algorithm with the corresponding DAGs of the data in Figure 1. The DAGs are constructed using the four files in \mathcal{F} from Figure 1(b). Figure 4(a) illustrates the files selected at the initialization stage. By applying Remove Policy I and Add Policy I, f_3 is removed from the list and f_4 is added, as illustrated in Figures 4(b). The result correctly follows the DAG constraint and shows a lower space cost than that in Figures 4(a). However, as shown in Figures 4(c), applying Remove Policy II and Add Policy II results in f_2 being removed and f_4 being added. This conflicts the DAG constraint since f_2 is the ancestor of f_4 , which is invalid. The algorithm would avoid such conflicting selection following the DAG constraint.

Algorithm 1 outlines the complete procedure of the proposed Multi-Column Compaction method. In some cases, the number of files in a level is not large (e.g., with size ratio 10). While a brute force strategy also works by comparing all the possible options, it is still time-consuming considering it invokes more computations of the overlaps than our proposal, i.e., the reduction in compaction time compensates for the extra selection time.

6 SYSTEM DEPLOYMENT

In this section, we introduce how the proposed MCC is integrated into Apache IoTDB [4]. We first describe the system as well as the compaction procedure. Next, to reduce the time cost of the algorithm, we devise File Prefetcher and Compaction Cache respectively in Sections 6.2 and 6.3 to improve the efficiency.

6.1 Deployment in Apache IoTDB

As illustrated in Figure 2, the Compaction Scheduler triggers the Compaction Selector with a predefined interval and size capacity. After the files are selected, they will be packaged and submitted as a compaction task. The compaction task is then submitted to Compaction Task Manager and waits to be conducted. All the submitted tasks are stored in a task queue, and will be extracted to execute by the compaction thread pool.

The proposal has been open-sourced as part of IoTDB in its official GitHub repository [5]. Specifically, our contributed codes are mainly as the new SizeTieredCompactionSelector [9]. To use MCC, one may build from source following the instructions of IoTDB [5] and set compactionSelectFileMethod=mcc in the configuration file iotdb-engine.properties.

6.2 File Prefetcher

While the proposed MCC selects files for less space cost, it requires reading files while performing the proposed algorithm. Such strategy might incur higher time cost when reading files. While reading the files and obtaining their keys is viable, we further consider devising our dedicated file reader, namely File Prefetcher, for more efficient file processing.

We start from the problem introduced in Section 4.1. Recall that, our problem focuses on space reduction. While the space cost of a file is composed of keys, values and bitmaps, the computation of the space reduction is indeed not related to the values themselves, but related to the positions of the values, i.e., the bitmaps. In other words, we can check the value overlaps (i.e., updates and delays) with bitmaps and keys, since they already mark the positions of all the values regardless of what the values are.

Unfortunately, the native File Reader in IoTDB deserializes all the keys and values together, and cannot separately read keys and bitmaps without deserializing values. We thus implement the File Prefetcher, which only reads keys and bitmaps to avoid the time-consuming value deserialization. The File Prefetcher follows similar steps of File Reader. It first loads the file into memory, then deserializes the keys and then the bitmaps. The value buffer is neglected thus saving the time of deserializing the values.

6.3 Compaction Cache

To avoid repeated computations, we also provide the Compaction Cache, which records the computed overlaps between files. It is maintained in each level in LSM-tree. When the files are selected and submitted as a compaction task, it will remove the related costs of them. When new costs of files are computed in the specific level, the cache of this level is also updated. By Compaction Cache, MCC could (1) prune files when they do not overlap with each other, and (2) check the cache for computed costs.

7 EXPERIMENTS

In this section, we implement our proposal in IoTDB with tieringlayout LSM-tree, and experimentally evaluate the methods over real-world datasets. The experiment-related codes and data are available at [8].

Table 2: Dataset summary.

	#point	#row	#column	null rate
Campus	8.4M	1,000,000	10	15.81%
CSSC	28.8M	639,770	48	6.22%
WC	64.3M	10,000,000	8	19.58%
WH	12.2M	1,600,000	8	4.43%
SAMSUNG	5.6M	300,000	19	1.25%
CitiBike	8.9M	600,000	15	0.1%
CNNC	46.2B	300,000,000	154	0.5%

7.1 Settings

7.1.1 Datasets. In the experiments, we use seven real-world time series datasets with different rows, columns and null rates. SAM-SUNG [42] and CitiBike [10] are real-world datasets without artificial delays. Table 2 provides a summary of the datasets.

7.1.2 *Metrics.* While the main focus of the paper is space size, i.e., how much space is for storing the multi-column data on disk, we also analyze the potential benefits of MCC to other metrics.

Space size. Space size is defined as the disk space used for storing the data. Since the compaction is continuously triggered, without further explanations, the space size is computed after all the compaction tasks finish, i.e., none of the level exceeds its capacity.

Space amplification (SA). SA is the key metric of the evaluation. SA is defined as the ratio of the total cells divided by the unique cells [25, 30]. ¹ Actually, SA computes the space cost for the obsolete/invalid data to evaluate the usage of the disk space.

Throughput and resource utilizations. We also report the write and read throughput as well as the resource utilizations including CPU, memory and disk I/O to show the impact of MCC on system performance.

7.1.3 Workloads and Database Parameters. It is important to evaluate how different methods handle different workloads. We thus introduce how we generate different workloads for evaluating the proposal. Taking both delays and updates into consideration, given the input files, first, we set *delay rate* and *update rate* to control the ratio of the data that are delayed or updated. Based on them, we randomly select the data to be delayed or updated, simulating IoT data input. By default, update rate and delay rate are set to 25%.

7.1.4 Methods. We compare our proposed Multi-Column Compaction (MCC) with the following competitors:

- Round-Robin: Selecting the files following a round-robin strategy [22].
- (2) Oldest: Selecting files according to their flushing time, i.e., selecting the oldest files for compaction [4, 38], which is also the default strategy of IoTDB.
- (3) Update: The file selection strategy selects files based on the highest number of updates in single-column scenarios [6].

¹To apply SA to the multi-column data, we slightly modify the definition of SA in [30], where SA is the ratio of the total entries divided by the unique entries.



Figure 5: Scalability on CNNC dataset.



Figure 6: Varying delay rate and update rate on WH dataset.

7.2 Comparison over Different Workloads

In this section, we evaluate the influences of different workloads to the compactions.

7.2.1 Scalability. To evaluate the scalability of MCC, we consider an extremely large-scale IoT environment of CNNC, customer of IoTDB, comprising 154 columns, with row counts varying from 1×10^8 to 3×10^8 . It amounts to 205GB of data. The space cost and SA are provided in Figure 5. The main impact of larger data volume is the increasing number of levels in the LSM-tree. As the data size grows, the LSM-tree becomes deeper, making it more challenging to manage SA due to a large number of delays and updates. By our selection algorithm for proper files to compact, MCC consistently demonstrates high performance with hundreds of gigabytes data. This again highlights its robustness and scalability.

7.2.2 Varying Delay Rate. Delay rate is another essential workload parameter, as it controls the proportion of delayed data. In general, a high delay rate poses challenges for all methods, since the keys are delayed more severely and incur higher SA. Figure 6(a) reports the results of the methods by varying delay rate from 10% to 50%. Overall, as the delay rates increase, the SA of all methods tends to grow. Nevertheless, MCC consistently shows the lowest SA among all the competitors, validating the applicability of MCC when data are delayed to different degrees.

7.2.3 Varying Update Rate. Update rate affects the space size more evidently as shown in Figure 6(b). As the update rate increases, the baselines demonstrate significant growth in SA, whereas MCC maintains a lower SA and experiences comparatively less growth.



Figure 7: Varying null rate.



Figure 8: Evaluation on datasets with real-world delays.

This validates the effectiveness of MCC in managing large workloads with high update rates.

7.2.4 Varying Null Rate. The gain of the proposed method mainly comes from sparse updates and data delays, as illustrated in Figure 6. Especially, with the increase of update rates in Figure 6(b), the improvement by our MCC becomes more significant. Null rates of the datasets do not largely affect the improvement, but only the corresponding space costs. Nevertheless, we vary the null rate of the WH dataset from 5% to 25% by randomly removing some data and evaluating the space size and SA of different methods. The results are presented in Figure 7. It can be observed that different null rates result in different methods is not affected by the null rates according to Figure 7(b).

7.2.5 Datasets with Real-world Delays. In this section, we evaluate MCC over two real-world datasets without artificial delays. SAMSUNG [42] is recorded with out-of-order (delayed) events with network issues over mobile devices in IoT applications. CitiBike [10] is composed of Citi Bikers ride records with out-of-order recorded time (i.e., with different degrees of data delays).

The results in Figure 8 show that MCC achieves the lowest SA across varying data sizes in both datasets. Since these datasets mainly suffer from real-world delay issues, the performance gains of MCC are primarily attributed to its ability in handling time delays more effectively. Other non-time-aware techniques such as Round-Robin have worse performance. The results are consistent with Figure 6(a) on various delay rates, i.e., MCC is more effective in handling severe delays.



Figure 9: The performance of MCC on throughput (pts/s) and resource utilization metrics across different workloads.

7.2.6 Throughput and Resource Utilization under Various Workload Patterns. In this section, we evaluate MCC and baseline algorithms under various workload patterns, including mixed read/write scenarios (with a read-to-write ratio of 1:1) and read-intensive scenarios (with a read-to-write ratio of 10:1). These workloads are generated using IoT-benchmark [11], a specialized tool for benchmarking time-series databases. The experiments utilize the data from the WH dataset to generate the workloads. Writes within these workloads are interleaved with reads. The queries are mainly range lookups, which are more common for time series databases. We separately record throughput and resource utilization (including CPU usage, memory usage and disk I/O). All metrics are normalized by dividing by the largest value among the three methods for better visualization.

Figures 9(a) and (b) display the results for mixed workload and read-intensive workload, respectively. As for the throughput, while it is not the primary target of our MCC, it still outperforms the baselines in both workload types, especially in the read-intensive workload. This improvement is attributed to the reduced lookup costs of MCC due to the reduced file sizes of the merged files.

In addition to the system throughput, we expand our experiments to involve additional metrics, including CPU usage, memory usage and disk I/O. During the evaluation, the CPU and memory usage metrics are the average values recorded throughout the execution of the workloads. Additionally, we also record the frequency of disk I/O operations for various methods.

The results are also presented in Figure 9. It can be observed that the performance of MCC in terms of throughput, CPU usage, and memory usage is comparable to that of the baseline methods. While MCC incurs extra disk I/O due to additional bitmap reads during the selection algorithm, this trade-off is considered acceptable when weighed against the remarkable gains of MCC in SA. Moreover, it does not impact the throughput as also evidenced by Figure 9. In addition, MCC does not lead to clearly increased CPU and memory usage.

7.3 Effectiveness and Limitations of MCC

Table 3: Ablation study of MCC on SA results.

	WH	Campus	WC	CSSC
MCC	1.659	1.646	1.713	1.703
MCC without Add Policy I	+0.061	+0.091	+0.058	+0.072
MCC without Add Policy II	+0.013	+0.022	+0.017	+0.015
MCC without Remove Policy I	+0.065	+0.105	+0.042	+0.081
MCC without Remove Policy II	+0.003	+0.001	-0.001	+0.002
Round-Robin	1.973	1.940	2.005	1.989
Oldest	1.936	1.902	1.966	1.938

Table 4: Comparison of MCC and the optimal solution.

Metric	Dataset	MCC	Optimal	Difference	Avg
SA	WH	1.652	1.648	-0.3%	
	Campus	1.429	1.409	-1.4%	-2.7%
	WC	1.458	1.413	-3.1%	
	CSSC	1.669	1.568	-6.0%	
Select Time (s)	WH	18.5	147.7	+698%	
	Campus	12.4	73.2	+490%	+828%
	WC	76.2	1209.3	+1487%	
	CSSC	17.1	125.9	+636%	

7.3.1 Ablation Study. In this section, we conduct an ablation study to validate the impact of the add/remove strategies on MCC. The results of SA are summarized in Table 3.

Add Policies. Both Add Policies are essential for MCC, as they consider overlaps in different dimensions. Note that MCC without Add Policy I leads to a significant increase in the SA, highlighting its contribution to the overall performance. This is not surprising since Add Policy I specially focuses on space reduction through a locally optimal strategy. Add Policy II considers the key dimension to reduce space cost. To avoid local optima, it prevents adding repeatedly the same files by Add Policy I.

Remove Policies. According to the results, Remove Policy I is beneficial for the performance of MCC by removing the files that minimally affect space reduction. Remove Policy II appears less impactful as it randomly removes files from the list. Nevertheless, as introduced in Section 5.2, it still plays an important role in preventing cycling and local optima, thereby appearing to be effective in three out of four datasets.

7.3.2 Comparison to the Optimal Solution. In this section, we compare MCC with the optimal solution that utilizes a brute-force approach to traverse all file combinations, to better understand the efficiency and potential limitations of the proposal.

As shown in Table 4, MCC demonstrates comparable space cost to the optimal solution, which only shows an average improvement of 2.7%. However, the optimal solution results in a significantly higher time cost to select proper files, with an average increase of 828%. Note that the compaction execution time of both solutions is almost the same and thus not included in the table. For instance, both methods take about 115s to execute compaction for WH dataset. However, the optimal solution incurs an additional 147.7s of latency, which is 1.28× the compaction execution time,



Figure 10: Varying column number.

while bringing only limited performance gains. Additionally, the number of selected files is set to 3 in this evaluation. Given that the latency of the brute-force approach for the optimal solution increases exponentially with the number of selected files, it becomes increasingly untenable as the number of selected files grows. To conclude, the heuristic approach MCC may have some effectiveness limitations, but the performance gain compared to the optimal solution is slight. The efficiency of MCC is significantly better, with only 11% of the time cost of the optimal solution in average.

7.3.3 Comparison to Single-column Compaction. Note that singlecolumn compaction strategies solely focus on overlapping keys (updates) while neglecting the bitmap for representing multi-column data. The extra space of bitmap is however crucial for reducing SA. Moreover, the delay of single-column data will not introduce overlapping keys thus has no SA to consider as well. For instance, one of the file selection strategies used in RocksDB [6, 38] selects files based on the highest number of updates in single-column scenarios, without considering delay and bitmap issues. Thereby, simply using single-column compaction is insufficient for multi-column data. This issue is more severe with a larger number of columns, where the space cost of the bitmap is significant. Unfortunately, this is common in IoT time series applications, suggesting the necessity of multi-column compaction.

We compare the proposed MCC with the single-column updatebased selection strategy (i.e., Update [6, 38] in Figure 10). In the context of single-column data without bitmap, SA mainly comes from data updates. Therefore, as shown in Figure 10, for singlecolumn data (i.e., when the column number is 1), MCC achieves similar performance as the update-based strategy, since both of them choose files based on updates. It verifies the above analysis. However, as the column number increases, the performance of the update-based strategy declines, whereas MCC continues to deliver robust performance with the consideration of bitmap costs

Table 5: Combining MCC with Lazy Leveling (LL).

Metric	Dataset	MCC	MCC+LL	Difference	Avg
SA	WH Campus WC CSSC	1.652 1.429 1.458 1.669	1.537 1.350 1.407 1.565	-7.0% -5.5% -3.5% -6.2%	-5.6%
WA	WH Campus WC CSSC	3.45 3.34 3.43 3.36	5.50 4.77 4.31 5.13	+59.4% +42.8% +25.7% +52.7%	+45.2%

for multi-column data. The reason is that simply relying on the number of updates neglects the incomplete bitmaps resulting from data delays, which significantly impacts SA.

7.3.4 Combination with Other Methods. Dostoevsky [23] introduces the Lazy Leveling LSM-tree design, known as L-leveling. This design utilizes leveling layout for the last level of the LSMtree while employing tiering in the other levels. The Lazy Leveling strategy can reduce SA associated with tiering layout through our design. We thus implement the Lazy Leveling strategy in Apache IoTDB and integrate it with the proposed MCC.

Table 5 presents the results. It can be observed that Lazy Leveling (LL) can be combined with MCC and reduce the space amplification (SA). On average, Lazy Leveling achieves a reduction of 5.6% in space cost compared to MCC alone. This improvement seems not substantial, because MCC already addresses space issues with updates and delays in lower levels effectively. However, this integration results in about 45.2% increase in write amplification (WA), owing to the need for sorting time series in the leveling layout which incurs additional disk I/O operations.

7.4 Applicability of MCC

MCC can also be generalized to other LSM-tree implementations without similar support, such as tiering layout. In addition to the tiering LSM policy with columnar storage, this section discusses the applicability of the proposal, especially on implementing MCC in row-based storage and LSM-trees with leveling and partitions. We further implement MCC in RocksDB [6], a leveling LSM-tree based storage, to show the applicability of MCC to other systems as well as the use cases.

7.4.1 LSM-trees with Leveling and Partitioning. LSM-trees with leveling, such as LevelDB [7] and RocksDB [6], also incorporate the file selection strategies during compaction. Unlike tiering, LSM-trees with leveling partition a run into non-overlapping files. The file selection problem in leveling is slightly different from that in tiering. As described in Section 2.2, in a leveling layout, the files in the same level are all sorted by keys, i.e., the updates and delays data in the same level will be merged directly. When compaction is triggered from Level *i* to Level *i* + 1, the selected files should be merged with the files in the next level. Despite these differences, we can still apply the proposed MCC to LSM-trees with leveling. Following the idea of optimizing SA, the goal of MCC in leveling layout is to choose the proper merging files in Level *i* that lead to the



Figure 11: Implementation of MCC in RocksDB.

lowest space cost after merging them into Level i+1, i.e., maximizing space reduction. Since the files in Level i do not overlap, we can optimize space reduction by individually assessing the impact of each file when moved to the next level.

Therefore, we implement the proposed MCC in RocksDB [6], an open-source key-value store adopting leveling layout. The implementation of MCC in RocksDB is similar to that in IoTDB by modifying the file picker module. We also reuse the index reading module in RocksDB to prefetch the keys for reducing the overhead of reading files. The computed costs of merging the files are also cached, to avoid redundant computations. Figure 11 illustrates the results of each dataset in RocksDB by varying data size. Owing to the data updates and delays, MCC consistently outperforms baselines across all datasets. Nevertheless, the performance enhancements offered by MCC in a leveling layout are not as substantial as those observed with tiering layouts. The reason is that leveling layouts ensure the files within the same level are all sorted without any overlaps, in contrast to tiering layouts.

7.4.2 Row-based Storage Using LSM-trees. MCC can be generalized to row-based storage based on LSM-tree architecture. For instance, TiDB [27] is an open-source HTAP database with a row-based storage engine TiKV [12]. In its data model, each row is represented as a key-value pair, analogous to the multi-column time series storage in IoTDB. When updates and delays are introduced to a row in TiDB, it leads to NULL values as well as extra space overhead. This presents an opportunity for MCC to optimize space utilization during compaction by selecting files to optimize space reduction. To apply MCC, the cost function needs to be slightly adjusted as TiKV does not use bitmap. However, the core part of the cost function and the selection framework of MCC are still applicable, as they already consider the data updates that are frequent in OLTP workload. Hence, implementing MCC in row-based storage systems is not only feasible but can also enhance their efficiency. Note that TiKV also employs RocksDB for persistent storage. In this sense, we can



Figure 12: Varying deletion rate.

implement MCC in TiKV by adapting the implementation with RocksDB as well, which has been validated in Figure 11. We omit the evaluation results on this similar implementation.

7.4.3 Use Cases in Addition to Time Series. Note that RocksDB [6] is a general-purpose system not specialized for time series. LinkedIn utilizes RocksDB to manage its vast and complex data workloads composed of user interactions, connections and posts, leading to a large number of real-time updates [13]. Given its global user base, the data management task of LinkedIn faces the challenges from frequent updates by user activities (e.g., profile modifications and interaction results). To mitigate the high space amplification resulting from these workloads, integrating RocksDB with MCC could provide a solution for managing the large number of updates. The overall storage efficiency is thus expected to be improved.

7.4.4 Applicability of MCC to Other Time Series Databases. While our findings are mainly discussed in the context of Apache IoTDB, the considered features such as multi-column storage and bitmap management are indeed prevalent in other time series databases using LSM-trees. For instance, InfluxDB IOx [14] leverages LSM-trees with Apache Parquet file format [15], which inherently supports multi-column storage and bitmap management. It suggests that our proposed MCC approach could also be effectively implemented in similar structures and demonstrates its adaptability.

7.5 Variation of File Selection Goals

Different file selection algorithms have different goals for optimization. For instance, selecting "oldest" files in a level [6, 22] tries to improve the read performance. Selecting files with the highest number of updates [6] tries to reduce the space cost and in the meantime improve the read performance. Round-robin strategy [19, 22] tries to maintain a balance between different metrics, whereas MCC takes both delays and updates into consideration. While SA is the primary focus, we show that MCC can be extended and integrated with other goals through the cost function.

Specifically, in addition to delays and updates, reducing tombstones resulting from data deletes could also be an interesting goal. Due to the out-of-place update principle of LSM-trees, a delete operation inserts a tombstone to invalidate the stored data [37]. While this facilitates fast data deletion, it comes at the cost of high SA. To explore the potential of enhancing MCC by incorporating various objectives, we integrate tombstone density [1] within the MCC framework to further reduce space costs and enhance read performance. Following Formula 2, we can define $\Delta C(F)$ as:

$$\Delta C(F) = \sum_{f \in F} C(f) - C(F) + D(F), \tag{3}$$

where D(F) denotes the size of deleted data by tombstones. This adaptation enables MCC to select files with higher D(F), leading to lower SA after compaction.

We extend our experiments to validate the proposed approach by incorporating data deletion into the workloads at varying rates. The proposed MCC-D (i.e., MCC with deletion optimization) modifies the cost function of MCC to combine the goal of selecting tombstones. Figure 12 presents the results. It can be observed from Figure 12(a) that the space cost of all methods decreases as the deletion rate increases. However, as shown in Figure 12(b), SA of the baseline methods increases with varying deletion rates. This rise is owing to the fact that not all the deleted data are immediately removed but are marked with tombstones, thereby increasing SA. The original MCC primarily considers data delays and updates, thus leading to a slight increase in its SA as well. MCC-D, instead, shows consistently lower SA with various deletion rates.

8 RELATED WORK

In this section, we review previous works on LSM-tree optimization and LSM-tree compactions.

8.1 LSM-tree Stores and Optimization

LSM-trees [32] have been widely used in modern stores [36], such as LevelDB [7], RocksDB [6], Apache IoTDB [4] and AsterixDB [17]. LSM-trees pioneer the storage engine for write-intensive scenarios, employing immutable files for storage and proposing compaction for better utilization of disk space. This makes LSM-trees adaptive and advanced in supporting different kinds of workloads and databases, not only NoSQL [1, 2], but also relational [28, 31] and time series databases [3, 40].

The increasing usage of LSM-trees also brings opportunities for optimization. Large number of techniques have been studied to optimize the performance of LSM-trees, ranging from improving WA [33, 34], merge operations [39, 46], special workloads [35, 43] and so on, according to the taxonomy in [30].

Among all the optimization techniques for special workloads, LSM-trie [43] considers ultra-large key-value stores as well as high WA, so that the storage of the metadata is also challenging. SlimDB [35] focuses on semi-sorted data and proposes optimization for such data to reduce WA. In this paper, we mainly focus on the workloads of Apache IoTDB, i.e., the typical workloads in IoT applications. The frequent out-of-order insertions and updates are the main concerns. Our previous works aim to mitigate the impact of the delayed issues in IoTDB, regarding data sorting [45] and WA [29], while MCC focuses on reducing SA and also obtains performance gains in both WA and lookup costs.

8.2 LSM-tree Compaction in Apache IoTDB

To manage disk space, LSM-trees employ compaction operations to reorganize data. Compactions are fundamentally significant to the performance of LSM-trees. Overall, the task of MCC belongs to the Data Movement Policy in the LSM-tree compaction design space discussed in [36]. Compared to the existing design space, this is the first work to extend the background of the files into multicolumn storage, a structure supported by Apache IoTDB to reduce space cost of multiple time series. IoTDB also leverages bitmaps to manage null values in multi-column data. Regarding bitmaps, our previous study [26] has improved the storage efficiency during flushing stage. Nevertheless, the issue of the bitmap in compaction stage motivates us to further devise MCC for IoTDB.

Existing studies optimize compactions in several aspects. Dostoevsky [23] proposes hybrid layouts to combine the advantages of leveling and tiering. The skip-tree [44] proposes a skipping merging idea for choosing files in higher levels. Dynamic Capacity Adaptation [25] dynamically adjusts LSM-tree level size to reduce spaceamplification due to the existence of obsolete entries. However, compaction of multi-column data is not researched. To this end, MCC addresses the scenarios when multi-column data are separately inserted or updated in Apache IoTDB. To the best of our knowledge, this is the first work concerning compaction for multicolumn data. In addition, Dostoevsky [23] could be combined with our proposal for file selection in the lower levels with tiering layout in IoTDB. Spooky [24] and Dynamic Capacity Adaptation [25] also accept tiering design and can be combined with MCC. PebblesDB [34] leverages guards to split the keys, which might result in large partitions in the worst cases.

9 CONCLUSION

LSM-trees are commonly employed in time series databases, and compactions are fundamental operations of LSM-trees to reorganize the files for high performance. Space amplification (SA) issues are prevalent in LSM-trees. Even worse, the IoT applications make the SA issues more severe in Apache IoTDB. In particular, the challenges of SA in IoTDB are unique owing to (1) multi-column storage devised with bitmaps and (2) frequent out-of-order insertions and updates in IoT applications. Regarding the duplicated keys, obsolete values and overlapping bitmaps of multi-column data, we devise Multi-Column Compaction (MCC) to reduce SA in Apache IoTDB. We formalize the problem of multi-column compaction and prove the hardness of the problem in Theorem 1. Selection strategy tailored to multi-column storage is also proposed to solve the problem. We then devise dedicated File Prefetcher and Compaction Cache to reduce the computation cost and accelerate the decisions. Analysis and experimental results in IoTDB validate the effectiveness of the proposed MCC in reducing SA in IoTDB. Notably, we implement the proposed MCC in RocksDB to show its applicability to other LSM-tree implementations.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (NSFC 62232005, 62072265, 62021002, 92267203), the National Key Research and Development Plan (2021YFB3300500), Beijing Key Laboratory of Industrial Bigdata System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

- [1] Apache Cassandra. https://cassandra.apache.org.
- [2] Apache HBase. http://hbase.apache.org/.
- [3] Influxdata. In-memory indexing and the Time-Structured Merge Tree. https: //docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/.
- [4] Apache IoTDB. https://iotdb.apache.org.
- [5] Code. https://github.com/apache/iotdb/tree/research/column-compaction.
- [6] RocksDB. https://github.com/facebook/rocksdb.
- [7] LevelDB. https://github.com/google/leveldb/.
- [8] Experimental code, data and full version technique report with proofs. https: //github.com/column-compaction/column-compaction.
- MCC core component. https://github.com/apache/iotdb/blob/research/columncompaction/server/src/main/java/org/apache/iotdb/db/engine/compaction/ inner/sizetiered/SizeTieredCompactionSelector.java.
- [10] Citi Bike Trip Histories. https://ride.citibikenyc.com/system-data.
- [11] IoT-benchmark. https://github.com/thulab/iot-benchmark.
- [12] TiKV. https://github.com/tikv/tikv.
- RocksDB Users and Their Use Cases. https://github.com/facebook/rocksdb/wiki/ RocksDB-Users-and-Use-Cases.
- [14] InfluxDB IOx. https://www.influxdata.com/blog/intro-influxdb-iox/.
- [15] Apache Parquet. https://parquet.apache.org/.
- [16] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In J. Clifford, B. G. Lindsay, and D. Maier, editors, Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989, pages 253–262. ACM Press, 1989.
- [17] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *Proc. VLDB Endow.*, 7(10):841–852, 2014.
- [18] M. Athanassoulis and S. Idreos. Design tradeoffs of data access methods. In F. Özcan, G. Koutrika, and S. Madden, editors, Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 2195–2200. ACM, 2016.
- [19] 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 Annual Technical Conference, pages 753–766. USENIX Association, 2019.
- [20] M. J. Brusco, L. W. Jacobs, and G. M. Thompson. A morphing procedure to supplement a simulated annealing heuristic for cost-andcoverage-correlated set-covering problems. *Annals of Operations Research*, 86:611–627, 1999.
- [21] A. Caprara, P. Toth, and M. Fischetti. Algorithms for the set covering problem. Annals of Operations Research, 98(1):353–371, 2000.
- [22] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal bloom filters and adaptive merging for lsm-trees. ACM Trans. Database Syst., 43(4):16:1–16:48, 2018.
- [23] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, pages 505–520. ACM, 2018.
- [24] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto. Spooky: Granulating lsm-tree compactions correctly. *Proc. VLDB Endow.*, 15(11):3071– 3084, 2022.
- [25] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in rocksdb. In 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org, 2017.
- [26] C. Fang, S. Song, H. Guan, X. Huang, C. Wang, and J. Wang. Grouping time series for efficient columnar storage. *Proc. ACM Manag. Data*, 1(1):23:1–23:26, 2023.
- [27] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao,

N. Cameron, L. Pei, and X. Tang. Tidb: A raft-based HTAP database. *Proc. VLDB Endow*, 13(12):3072–3084, 2020.

- [28] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30* - *July 5, 2019*, pages 651–665. ACM, 2019.
- [29] Y. Kang, X. Huang, S. Song, L. Zhang, J. Qiao, C. Wang, J. Wang, and J. Feinauer. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. In 38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022, pages 3340–3352. IEEE, 2022.
- [30] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. VLDB J., 29(1):393-418, 2020.
- [31] Y. Matsunobu, S. Dong, and H. Lee. Myrocks: Lsm-tree database storage engine
- serving facebook's social graph. Proc. VLDB Endow., 13(12):3217–3230, 2020.
 P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.
- [33] F.-F. Pan, Y.-L. Yue, and J. Xiong. dcompaction: Speeding up compaction of the lsm-tree via delayed compaction. *Journal of Computer Science and Technology*, 32:41–54, 2017.
- [34] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, pages 497–514. ACM, 2017.
- [35] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. Proc. VLDB Endow, 10(13):2037–2048, 2017.
- [36] S. Sarkar and M. Athanassoulis. Dissecting, designing, and optimizing lsmbased data stores. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 -17, 2022, pages 2489–2497. ACM, 2022.
- [37] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A tunable delete-aware LSM engine. In SIGMOD Conference, pages 893–908. ACM, 2020.
- [38] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis. Constructing and analyzing the LSM compaction design space. *Proc. VLDB Endow*, 14(11):2216–2229, 2021.
 [39] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok.
- [39] F. J. Sherty, R. F. Spinale, R. R. Mapali, D. Fildrews, J. Seyster, and E. Zadok. Building workload-independent storage with vt-trees. In Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13), pages 17–30, 2013.
- [40] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. Mcgrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun. Apache iotdb: Timeseries database for internet of things. *Proc. VLDB Endow.*, 13(12):2901–2904, 2020.
- [41] C. Wang, J. Qiao, X. Huang, S. Song, H. Hou, T. Jiang, L. Rui, J. Wang, and J. Sun. Apache iotdb: A time series database for iot applications. *Proc. ACM Manag. Data*, 1(2):195:1–195:27, 2023.
- [42] W. Weiss, V. J. E. Jiménez, and H. Zeiner. A dataset and a comparison of out-oforder event compensation algorithms. In *International Conference on Internet of Things, Big Data and Security*, volume 2, pages 36–46. SciTePress, 2017.
- [43] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In S. Lu and E. Riedel, editors, 2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA, pages 71–82. USENIX Association, 2015.
- [44] Y. Yue, B. He, Y. Li, and W. Wang. Building an efficient put-intensive keyvalue store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2016.
- [45] X. Zhang, H. Zhang, S. Song, X. Huang, C. Wang, and J. Wang. Backward-sort for time series in apache iotdb. In 39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023, pages 3196–3208. IEEE, 2023.
- [46] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun. Pipelined compaction for the lsm-tree. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 777–786. IEEE, 2014.