Master's Thesis
석사 학위논문

A Delete-Aware Compaction Trigger Method for LSM-Tree Based Key-Value Stores

Ilseop Lee(이 일 섭 李 一 燮)

Department of
Information and Communication Engineering

DGIST

2021

Master's Thesis
석사 학위논문

A Delete-Aware Compaction Trigger Method for LSM-Tree Based Key-Value Stores

Ilseop Lee(이 일 섭 李 一 燮)

Department of
Information and Communication Engineering

DGIST

2021

A Delete-Aware Compaction Trigger Method for LSM-Tree Based Key-Value Stores

Advisor: Professor Jemin Lee
Co-advisor: Professor Min-Soo Kim

by

Ilseop Lee
Department of Information and Communication Engineering
DGIST

A thesis submitted to the faculty of DGIST in partial fulfillment of the requirements for the degree of Master of Science in the Department of Energy Science & Engineering. The study was conducted in accordance with Code of Research Ethics[1]

11. 18. 2020

Approved by

Professor Jemin Lee (Advisor)	(signature)
Professor Min-Soo Kim (Co-Advisor)	(signature)

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of DGIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

A Delete-Aware Compaction Trigger Method for LSM-Tree Based Key-Value Stores

Ilseop Lee

Accepted in partial fulfillment of the requirements for the degree of Master of Science.

11. 18. 2020

MS/IC
201922031

이 일 섭. Ilseop Lee. A Delete-Aware Compaction Trigger Method for LSM-Tree Based Key-Value Stores. Department of Information and Communication Engineering. 2020. 31p. Advisors Prof. Jemin Lee, Co-Advisors Prof. Min-Soo Kim

ABSTRACT

LSM-tree based key-value stores show high performance in write-intensive workloads due to the out-of-place update structure. However, this structure requires additional storage space to store a lot of redundant data in database and results in high space amplification. In order to remove redundant data, most modern LSM-tree stores use a compaction triggered by the capacity in each level. But the size-based compaction trigger often occurs inadequate compaction frequency according to the workload, causing high write amplification and reduces overall performance. To address this, we introduce delete-aware compaction trigger which responds to the current workload's deletion rates and reduces inefficient compactions consisting of most valid records. We implemented Delete-Aware RocksDB on top of RocksDB, and we show that it outperforms by keeping low space amplification without write amplification cost compared to RocksDB. Furthermore, we show our system cooperated with Monkey, one of the state-of-the-art LSM-tree based key-value store, outperforms existing systems in terms of throughput.

Keywords: LSM-tree based key-value store, Compaction trigger, RocksDB, Space amplification

List of Contents

List of Tables

List of Figures

I. INTRODUCTION

A key-value store is a simple data storage paradigm mapping search keys to corresponding data values. Records are stored and retrieved using a unique key to identify and find the value within the key-value store. Due to its simple implementation, many companies have their own key-value store as commercial products or open sources (e.g. Dynamo [20] at Amazon, Cassandra [7] at Apache, RocksDB [3] at facebook, LevelDB [4] and BigTable [21] at Google, and Memcached [10]). To optimize for write-intensive workloads, most of modern key-value stores use Log Structured Merge Tree (LSM-Tree). LSM-tree has a buffer in main memory for inserted/updated/deleted entries and the buffer is flushed as a sorted run into secondary storage when it fills up. LSM-tree sort-merges sorted runs when a level in LSM-tree reaches the capacity. A filter such as Bloom filter [22] is used to skip IO for faster lookups.

LSM-tree based key-value stores are widely used in applications such as graph processing [13, 17], real-time data processing [19], OLTP workloads [18, 8]. However, it's difficult to optimize LSM-tree key-value store because of the various realistic workloads and too many tuning knobs for LSM-tree such as merge policy, size ratio and memory allocation. In addition, there are trade-offs between the read, the update, and the memory with respect to access method [11], so it is important to tune appropriately. However, even though existing systems support tuning guides, there is still inefficiency in terms of compaction triggers that do not respond flexibly to the workload where redundant data occurs in various patterns. Also, many researchers tried to find the optimal system by changing the design of the system such as merge policy and combining various data structures between B+tree, LSM-tree and Hash Table, but it requires a lot of effort to implement and the system tends to be heavy. We address this problem by replacing existing size-based compaction triggers with delete-ratio based compaction triggers that conduct compactions depending on the ratio of redundant data in each level. We implemented Delete-Aware RocksDB on top of RocksDB and show that it outperforms existing systems in terms of overall throughput and space amplification.

In this section, we introduce the motivation, research goals and notations used in the thesis, and summarize the structure of the thesis.

1.1 Motivation

The motivation for the research is to make compactions of LSM-tree more efficient by including a lot of redundant records during compaction itself to keep less space amplification compared to the existing systems. To do this, we explore lifecycle of deletion records and its impact on performance in this section.

1.1.1 Lifecycle of a deletion record and inefficiency in a compaction

Deletion records in LSM-tree based key-value stores are stored as a tombstone, which indicate that corresponding key will be deleted. A deletion record is appended as a normal record due to out-of-place updates structure, but it generates many redundant records by making existing identical data invalid. Redundant records are removed in two cases during compactions: 1) when a record meets the identical key inserted more recently; 2) When a tombstone reaches the largest level. Redundant records occur the larger database than a user actually writes, which is known as space amplification that we will introduce in section 2.3.3.

We focus on when efficient compactions occur. LSM-tree based key-value stores sort-merge, called compaction, in order to 1) delete redundant data; 2) maintain limited number of sorted runs. To keep lower space amplification, it requires more compactions in general, but this entails a lot of resources such as CPU and IOs, which can degrade overall throughput. Frequent compaction reduces overall throughput, and infrequent compaction stores a lot of redundant data, resulting in less read performance and larger storage space of the entire database. Existing systems use size-based triggers as compaction triggers. However, this causes an inappropriate compaction in a specific workload, which can reduce overall performance. Therefore, triggering compactions in an appropriate period is important to maintain lower space amplification and less redundant records. In this thesis, we suggest delete-ratio based compaction trigger in order to properly perform the compaction according to the ratio of the deletion key.

1.1.2 Maintaining low space amplification decreases overall performance

Due to the out-of-place update structure, LSM-tree shows superior write performance with advantage of sequential IOs, but low space utilization. In addition, the major problem of the structure is low read performance since records may be stored in multiple locations. To address this problem, LSM-tree adopted bloom filter [22] to save IO whenever the filter returns "definitely not in the set". However, since it needs to explore from level 0 to the largest level until searching key is retrieved, the number of levels may be able to increase read path. Therefore, the larger number of levels as a result of high space amplification can degrade reads throughput. It is necessary to maintain a low space amplification to obtain high read performance, but this requires other resources. In Table 1, it shows a cost of LSM-tree with two merge policies explained in section 2.2.2. It shows that database size as results of the number of entries and levels is critical for overall performance. However, we can only tune the number of levels as results of size ratio and compaction frequency. The larger space amplification as results of infrequent compaction can increase the number of levels and degrade overall performance.

Many researches have been done to address the problem of inappropriate compaction frequency [16], and it shows that LSM-tree has superfluous compactions at smaller levels. We focus not only on the frequency of compaction, but also when inefficient compaction occurs. Existing systems conduct compactions in order to reduce redundant records and bound the number of sorted runs in a level. We found that superfluous compactions at smaller levels has been done as results of sort-merging between valid records that are triggered by size-based compaction trigger.

	LSM-tree based key-value store		**System keeping lower space amplification**	
	Leveling	**Tiering**	**Leveling**	**Tiering**
Entries in DB	O(N)	O(N)	O(N_δ)	O(N_δ)
Total bytes written to DB	O(N·E·L·T)	O(N·E·L)	O(N_δ ·E· L_δ ·T)	O(N_δ ·E· L_δ)
Write Amplification	O(L·T)	O(L)	O(L_δ ·T)	O(L_δ)
Non-zero result point lookup cost	O(1)	O(1+$e^{-m/N} \cdot T$)	O(1)	O(1+$e^{-m/N_\delta} \cdot T$)
Short range point lookup cost	O(L)	O(L·T)	O(L_δ)	O(L_δ ·T)
Insert/Update cost	O($\frac{L \cdot T}{B}$)	O($\frac{L}{B}$)	O($\frac{L_\delta \cdot T}{B}$)	O($\frac{L_\delta}{B}$)

Table 1. A cost of LSM-tree based key-value store and system keeping lower space amplification in terms of two different compaction policies

1.2 Research Goals and Thesis Structure

1.2.1 Goal 1: To make a compaction efficient according to the workload

Since many deletion records issued by various delete-ratio patterns of the workload generate redundant data, LSM-tree adopted the compaction. However, size-based compaction trigger cannot respond to the ratio of deletion records in a compaction, which often causes inefficient compaction consisting of most valid records that are not to be deleted. Inefficient compactions might be able to reduce overall performance by taking up CPU and IO resources. We aim to provide stable performance regardless of the workload by reducing inefficient compactions with delete-ratio compaction trigger.

1.2.2 Goal 2: To trigger a compaction at the proper time

In addition to the inefficient compaction taking up CPU and IO resources, size-based compaction trigger often occurs superfluous compactions at smaller levels in LSM-tree since it does not monitor the ratio of deletion records. In other words, superfluous compactions are generated by inadequate frequency of compaction. There has been a research that superfluous compactions at smaller levels in LSM-tree reduce the overall performance [16]. We intend to make a compaction to be triggered in an appropriate period using delete-ratio based trigger.

1.2.3 Thesis Structure

The thesis is structured into three main parts. In background section, we introduce main structure and features of RocksDB and factors for performance evaluation. In implementation section, we make the hypothesis and explores

II. Background

2.1 LSM-Tree Basics

A Log Structured Merge-tree [23] is a data structure that maintain data in two separate structures: in memory and disk. LSM-tree provides high write throughput because it is optimized by only performing sequential writes. All records are inserted first into the in-memory data structure. If the in-memory data structure exceeds a certain size threshold, it merges onto secondary storage data structure. Most LSM-trees employ multiple levels. Level 0 represents in-memory data structures and on-disk data at lager levels is organized into sorted runs.

RocksDB

RocksDB [3] is an embeddable persistent LSM-tree based key-value store developed at Facebook, which is built on earlier work on LevelDB [4] by Google. RocksDB is written entirely in C++ and encompasses multi-threaded compactions, bloom filters and transactions. Also, it is suitable for storing multiple terabytes of data in a single database and optimized for storing small to medium size key-values on fast storage: flash devices or in-memory. RocksDB organizes all key-values in sorted order and are written to a *memtable* that is an in-memory data structure and to a *SSTable* that is a secondary storage data structure. Due to the modular data structures, it provides highly flexible configuration settings that may be tuned to run on a variety of production environments.

2.1.1 Memtable and SSTable

The in-memory data structure is called a *memtable* holding data before they are flushed to *SSTable*., which is on-disk data structure. New write operations always insert data to *memtable*, and read operations query *memtable* before reading all *SSTables* at larger levels. There are many various implementations of *memtable* and RocksDB provides skiplist-based *memtable*, which has the same asymptotic expected time bounds as balanced trees.

A *SSTable*, Sorted Strings Table, is a format for storing key-value pairs in sorted order and can be represented on disk as a single file. *SSTables* are immutable once they are written to disk. In RocksDB, each *SSTable* contains a sequence of blocks, meta index and footer. The sequence of key-value pairs is stored in a sequence of data blocks and each block typically is 64KB in RocksDB. The meta block supports various types such as index block, filter meta block and properties meta block. Index blocks are used to locate a data block using the largest key and the smallest key of a block. The filter meta block stores a sequence of filters such as bloom filters to make look up faster, where a filter contains the output of **FilterPolicy::CreateFilter()** located in **filter_policy.cc**. The properties meta block provides various information of the table such as data size, index size, filter size, number of entries and number of data blocks. The meta index is used to locate meta blocks and an index is loaded into memory when the *SSTable* is opened. A lookup needs a single disk seek. It first finds the appropriate block using binary search in the in-memory index and reads the corresponding block from disk. The footer has a fixed size that contains the pointers for meta index, padding and magic number.

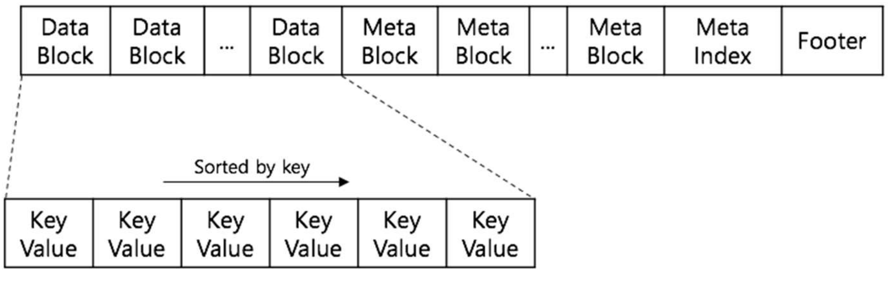


Figure 1. A format of block based SSTable in RocksDB

2.1.2 Compaction

Compactions are needed to remove multiple copies of the same key and to maintain an appropriate number of sorted runs. Compaction selects multiple files and merges them together and this processes only involve sequential reads and sequential writes. This is important to keep a handle on the read performance which degrades as the number of files increases. In most LSM-trees, there are two basic compaction policies as shown in Figure 2: Leveled Compaction and Size-Tiered Compaction, also called Universal Compaction in RocksDB.

Leveled Compaction

Leveled Compaction is the default compaction policy that a level is a sorted run in RocksDB. Leveled Compaction creates *SSTables* that are grouped into levels. Within each level, *SSTables* are guaranteed not to be overlapping. In RocksDB, each level is 10 times as large as the previous. When a compaction is triggered, it chooses a file at a level that the capacity exceeds a certain size threshold, and then sort-merges runs with overlapping range of the next level. Because Leveled Compaction guarantees non-overlapping key ranges between *SSTables*, it needs more IO than Size-Tiered Compaction.

Size-Tiered Compaction

Size-Tiered Compaction is a compaction policy that aims to lower write amplification, trading off read amplification and space amplification. Each level has *N* sorted runs. Size-Tiered Compaction merges all sorted runs in one level to create a new sorted run in the next level. In contrast to LevelDB and RocksDB, Size-Tiered Compaction is the default compaction strategy for Apache Cassandra [5].

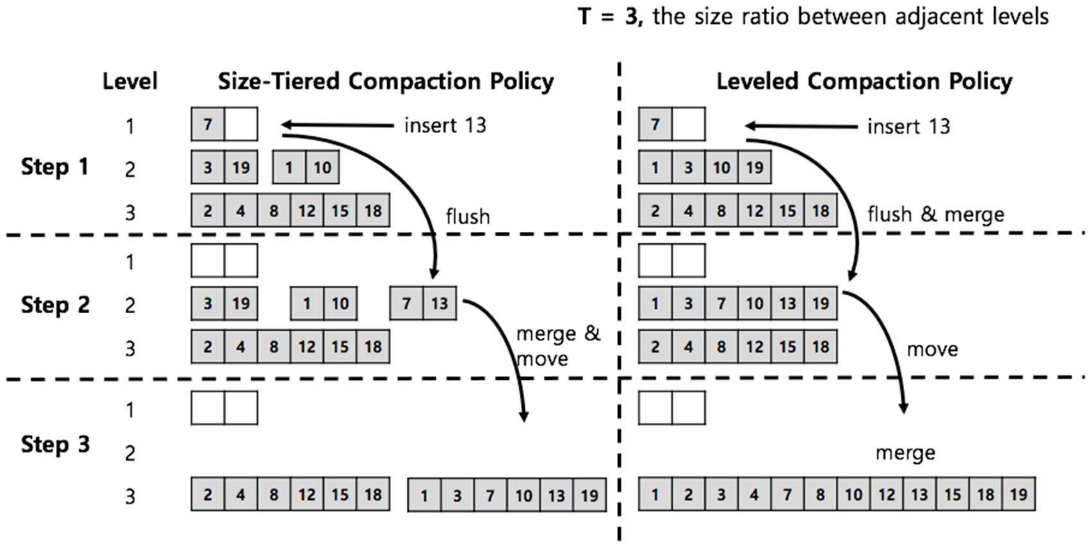

T = 3, the size ratio between adjacent levels
Level
Size-Tiered Compaction Policy
Leveled Compaction Policy
Step 1
Step 2
Step 3
insert 13
flush
flush & merge
merge & move
move
merge

Figure 2. Two different compaction policies of LSM-tree

2.1.3 Bloom Filter

A bloom filter is a space-efficient probabilistic data structure, which might be able to save IO when lookup. In bloom filter, false positive matches are possible, but false negatives are not. In other words, it returns that the data definitely does not exist in the given file, or the data probably exists in the given file. If bloom filter indicates a key is not present, then we can skip reading *SSTables* in given level. Bloom filters are stored in memory and it can provide more accuracy by allowing them to consume more memory. Bloom filter improves the read performance for keys that are missing in the system, but for the keys that are present in the system, the read is still expensive as we have to look into the *memtable*, index and the *SSTable*.

2.2 Amplification Factors

2.2.1 Read Amplification

Read amplification is the ratio of total read IO to user data respectively [14]. If the system needs to 5 IOs to answer a query, the read amplification is 5. Physical reads are handled by the storage device such as flash or disk and expensive compacted to logical reads such as cache or OS filesystem cache. Read amplification can also be defined for point lookups on non-existent keys. Queries for non-existent keys is very common in realistic workloads. LSM-tree may be able to conduct many reads depend on results of bloom filter at each level. For Leveled Compaction, it requires at most 1 IO because it has a sorted run in each level. However, Size-Tiered Compaction may require T IOs due to T sorted runs, where T is size ratio in Size-Tiered Compaction.

2.2.2 Write Amplification

Write amplification is the ratio of bytes written to storage versus bytes written to the database [12]. If a user is writing 10 MB/s to the database and 30 MB/s disk write rate is observed in the system, the write amplification is 3. An SSD or disk with a high write amplification may need to write as much data and cannot complete writing sooner. Also, high write amplification reduces the lifespan of SSDs and Disks. Leveled Compaction of modern key-value stores such as RocksDB still has high write amplification even though it provides better write throughput than B^+-tree [24] by sequential IOs.

2.2.3 Space Amplification

Space amplification of an LSM-tree is defined as the ratio of the size of database files on disk to data size. In other words, it describes how much extra space that a database will use on disk compared to the size of the data stored in the database [1]. If a user put 1 GB in the database and the database uses 2 GB, the space amplification is 2. RocksDB has configuration parameters to control the amount of space amplification. For Leveled Compaction, **hard_rate_limit** option determines its space amplification. But this is enforced by stalling writes and deletes during compactions. For Size-Tiered Compaction, **max_size_amplification_percent** option determines it. But Size-Tiered Compaction with the oldest file can be very large and will be doubled in size when it is compacted. In Figure 3, it shows disk usage for Size-Tiered compaction spikes during compactions [9]. The disk usage spike depends on the size of the SSTables being compacted. Size-Tiered compaction temporarily requires the disk space twice larger than the input SSTables. Therefore, we have to ensure that half the disk is free, but the cost would increase since higher space amplification can occur as data increases.

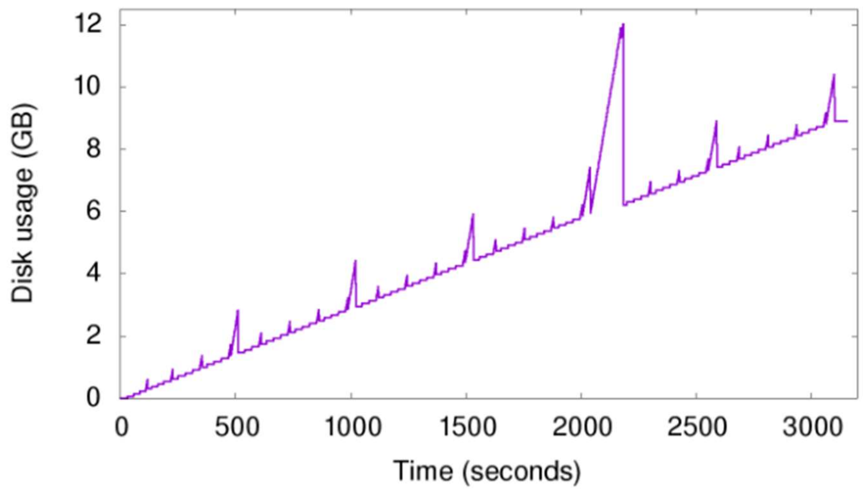


Figure 3. A disk usage of Size-Tiered Compaction during compactions

III. Implementation

In this section, we show the concept and implementation of Delete-Aware RocksDB and extending **db_bench**, benchmark tool of RocksDB, to evaluate our system. Finally, we present configurations for our system and introduce cooperation with Monkey, which is one of the state-of-the-art systems and is expected to create good synergy.

Notations

Table 2 gives a list of notations we use throughout the thesis.

Notation	**Description**
E	average size of a key-value entry
T	size ratio
B	number of entries in a page
m	total main memory allocated to bloom filters
N	total number of entries
N_δ	approximate number of entries as results of lower space amplification
L	number of levels
L_δ	approximate number of levels as results of lower space amplification
M	amount of main memory for *Memtable*

Table 2. Table of the notations used throughout the thesis

3.1 Implementing Delete-Aware RocksDB

Our system is built on top of RocksDB. It contains about four thousand lines of code written in C++. Modular data structures and API of RocksDB allow users to tune and modify it easily. We implemented Delete-Aware RocksDB with three approaches: 1) Delete-ratio based compaction trigger in section 3.2.1; 2) Pre-remove strategy for redundant tombstones in section 3.2.2; 3) Dynamic runtime adaptation in section 3.2.3.

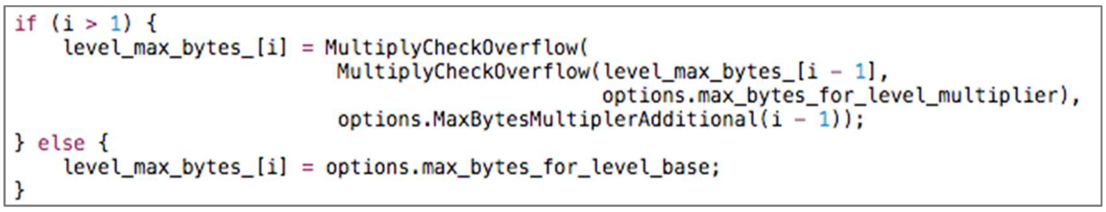
3.1.1 Delete-ratio based compaction trigger

Existing systems use size-based compaction trigger with size ratio *T*, where *T* is capacity multiplier as the level increases. In RocksDB, **max_bytes_for_level_base** indicates max capacity for level 1 and the capacity increases at larger levels by multiplying with **max_bytes_for_level_multiplier**. However, it does not monitor compactions whether the compaction is efficient or not. Inefficient compactions that are compactions between valid records (where it is not significant on space amplification, but significant on write amplification) can result in excessive compactions and lead to degrades performance. By keeping less space amplification without increasing write-amplification, it shows improved performance in terms of worse-case cost in LSM-tree as shown in Table 1.

We introduce a delete-aware compaction trigger, which respond flexibly to the workload's deletion rates. In contrast to existing LSM-tree based key-value stores, it contains the number of deletes in each level *i* as shown in Equation 1. We use the size of *Memtable* to estimate more accurate ratio, since the ratio of the number of deletes to the number of all entries in a level is very coarse-grained. If there are more deletion records in the current workload, it would trigger compaction more frequently and reach the largest level faster, and if there are less deletions, the compaction will be postponed until the deletion rates reach the limit. As a result, we can avoid inefficient compaction that sort-merges between valid data. By reducing the inefficient compaction, we can reduce superfluous compactions.

Deletion rates in level i, $R_i = \frac{E \cdot D_i}{M}$, where D_i is the number of deletes in level i (1)

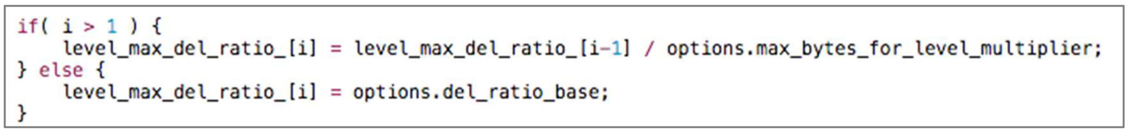
RocksDB calls **CalculateBaseBytes**() [6] to calculate bytes of each levels in LSM-tree, which is located in **version_set.cc**. For Leveled Compaction, max bytes for a level are set with **max_bytes_for_level_base** and **max_bytes_for_level_multiplier** as shown in Figure 4.



```
if (i > 1) {
    level_max_bytes_[i] = MultiplyCheckOverflow(
                            MultiplyCheckOverflow(level_max_bytes_[i - 1],
                                                  options.max_bytes_for_level_multiplier),
                            options.MaxBytesMultiplerAdditional(i - 1));
} else {
    level_max_bytes_[i] = options.max_bytes_for_level_base;
}
```

Figure 4. Calculation of max bytes in a level for Leveled Compaction in RocksDB

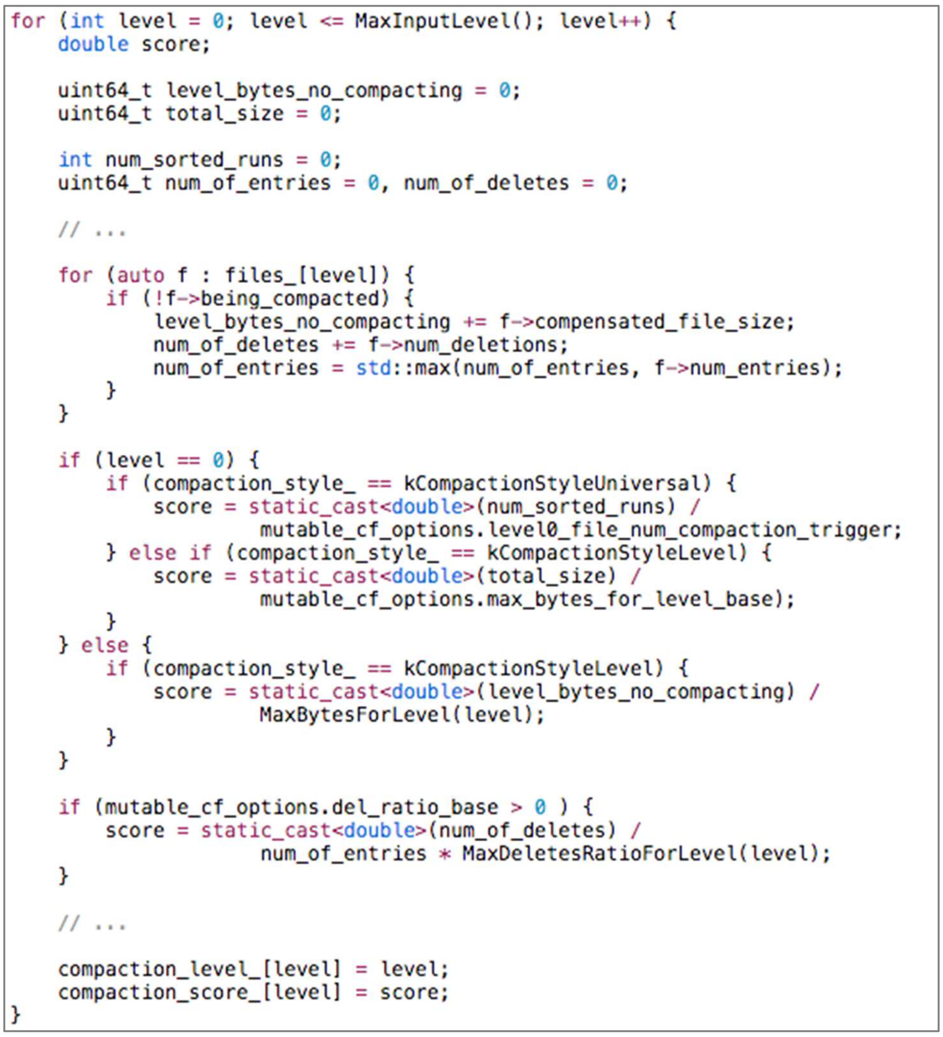
We added new variable called **del_ratio_base**, which is similar to **max_bytes_for_level_base**, in **Mutable-CFOptions** data structure located in **cf_options.h** and **level_max_del_ratio_** vector of double type. If **del_ra-tio_base** is set over 0, our system calls **CalculateDeletesRatio()** to calculate the ratio of deletions for each level as shown in Figure 5. But we use **max_bytes_for_level_multiplier** as divider for variable reuse. The value of **level_max_del_ratio_** decreases as the level increases to store a small number of delete keys at larger levels.



```
if( i > 1 ) {
    level_max_del_ratio_[i] = level_max_del_ratio_[i-1] / options.max_bytes_for_level_multiplier;
} else {
    level_max_del_ratio_[i] = options.del_ratio_base;
}
```

Figure 5. Calculation of max ratio of deletions for Leveled Compaction in Delete-Aware RocksDB

RocksDB computes compaction score to trigger compactions by calling **ComputeCompactionScore()** which is located in **version_set.cc**. if compaction score in a level is set over 1.0 in Leveled Compaction, RocksDB picks files in a level and conduct compaction with files in the next level. In Size-Tiered Compaction, compactions are triggered only by the number of sorted runs using **level0_file_num_compaction_trigger**. We compute compaction score again when **del_ratio_base** is set above 0 as shown in Figure 6. **ComputeCompactionScore()** is called after compactions and flushes to keep consistently between the storage state and compaction score.



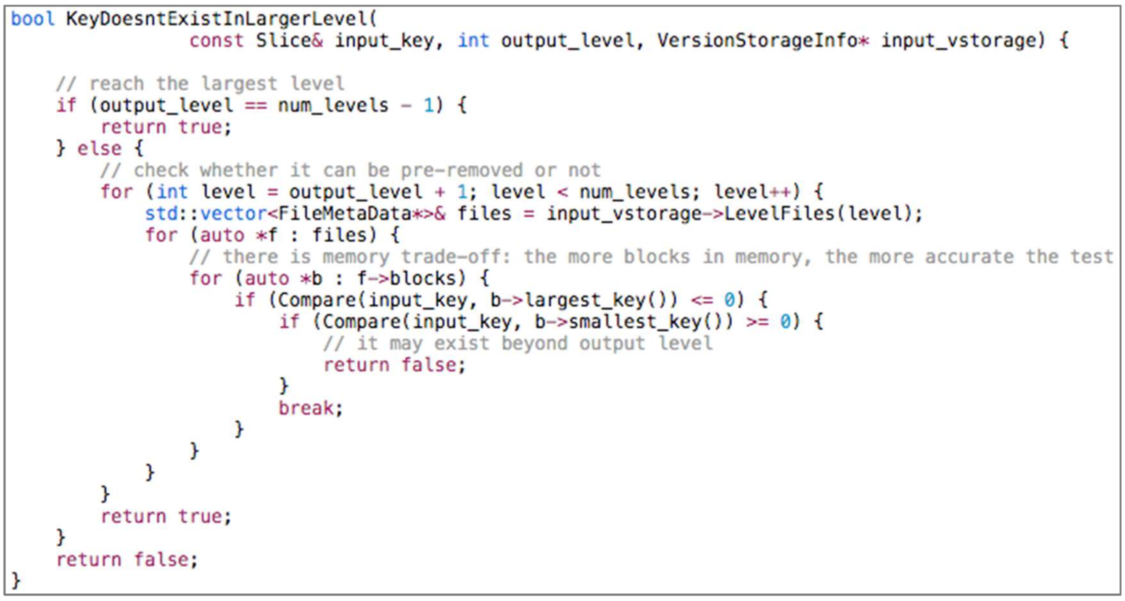
```
for (int level = 0; level <= MaxInputLevel(); level++) {
    double score;

    uint64_t level_bytes_no_compacting = 0;
    uint64_t total_size = 0;

    int num_sorted_runs = 0;
    uint64_t num_of_entries = 0, num_of_deletes = 0;

    // ...

    for (auto f : files_[level]) {
        if (!f->being_compacted) {
            level_bytes_no_compacting += f->compensated_file_size;
            num_of_deletes += f->num_deletions;
            num_of_entries = std::max(num_of_entries, f->num_entries);
        }
    }

    if (level == 0) {
        if (compaction_style_ == kCompactionStyleUniversal) {
            score = static_cast<double>(num_sorted_runs) /
                    mutable_cf_options.level0_file_num_compaction_trigger;
        } else if (compaction_style_ == kCompactionStyleLevel) {
            score = static_cast<double>(total_size) /
                    mutable_cf_options.max_bytes_for_level_base);
        }
    } else {
        if (compaction_style_ == kCompactionStyleLevel) {
            score = static_cast<double>(level_bytes_no_compacting) /
                    MaxBytesForLevel(level);
        }
    }

    if (mutable_cf_options.del_ratio_base > 0 ) {
        score = static_cast<double>(num_of_deletes) /
                    num_of_entries * MaxDeletesRatioForLevel(level);
    }

    // ...

    compaction_level_[level] = level;
    compaction_score_[level] = score;
}
```

Figure 6. ComputeCompactionScore() method for triggering compactions

3.1.2 Pre-remove strategy for redundant tombstones

A deletion record is inserted as a tombstone, which is the main reason for enhancing space amplification due to the characteristic of LSM-tree that performs out-of-place updates. It is important to keep less redundant data created by tombstones with compactions. During the compaction, previously inserted key-value pairs can be removed when they encounter the identical key or corresponding tombstone. But tombstones might not be able to be removed in an intermediate level because redundant key-value pairs with identical key can exist in larger levels. Therefore, tombstones can be removed 1) when they reach the largest level; or 2) when they encounter the latest record with the identical key. For the second case, when they encounter new valid record with the identical key, it can be checked and removed during the compaction without additional cost. However, for the first case, when they reach the largest level, lots of compactions are required in order to reach the largest level. If we can remove redundant tombstones during the compaction, we can not only save little storage space, but also measure a more accurate the deletes ratio between valid tombstones by removing unnecessary tombstones in advance. Although simply erasing some tombstones does not affect the size of the output of compaction, it can prevent the creation of a large amount of redundant data generated by tombstones. As a result, we can make the space amplification lower without a lot of additional resources.

To remove redundant tombstones, we modified the existing **FileMetaData** to store the range of each block of *SSTable* and use these ranges to check whether the input key does not exist in larger levels. In Figure 7, it shows a boolean function **KeyDoesntExistInLargerLevel** to check if the input key does not exist in the larger levels. It returns true when the output level of a compaction is the largest level and input key does not overlap with all blocks in larger levels. In addition, there is memory trade-off: The more memory we use for block ranges, the more accurate we can test. If the block size B is small, more accurate checks are possible as more ranges are explored, but it requires a lot of memory space and CPU cost and may affect the overall system performance. Therefore, it is important to find the optimal block size. We introduce the algorithm to find the optimal block size B in section 3.2.3.



```
bool KeyDoesntExistInLargerLevel(
                const Slice& input_key, int output_level, VersionStorageInfo* input_vstorage) {

    // reach the largest level
    if (output_level == num_levels - 1) {
        return true;
    } else {
        // check whether it can be pre-removed or not
        for (int level = output_level + 1; level < num_levels; level++) {
            std::vector<FileMetaData*>& files = input_vstorage->LevelFiles(level);
            for (auto *f : files) {
                // there is memory trade-off: the more blocks in memory, the more accurate the test
                for (auto *b : f->blocks) {
                    if (Compare(input_key, b->largest_key()) <= 0) {
                        if (Compare(input_key, b->smallest_key()) >= 0) {
                            // it may exist beyond output level
                            return false;
                        }
                        break;
                    }
                }
            }
        }
        return true;
    }
    return false;
}
```

Figure 7. A boolean function for pre-remove of redundant tombstones

3.1.3 Dynamic runtime adaptation

The amount of redundant data varies considerably depending on the workload. Additionally, even in realistic workloads, it is difficult to predict the amount of redundant data generated by deletion records. To address this, many researchers adopt runtime adaptation to make their system respond to the workload dynamically by changing the design with tuning knobs in runtime. We now introduce dynamic runtime adaptation to adjust the trigger value corresponding to the current workload in runtime. Dynamic runtime adaptation consists of three stages:

1) Measurement phase to detect workloads within *X* flushes; 2) Computation phase to find the optimal trigger value by finding local minimum of the average worst-case operation cost; 3) Transition phase to apply the new optimal trigger value. Figure 8. shows pseudo codes of dynamic runtime adaptation and cost functions estimating current worst-case operation cost to find the optimal memory usage and delete-ratio trigger value according to the current workload detected in Measurement phase. Delete-Aware RocksDB optimizes throughput with respect to point lookup cost, range lookup cost and update cost in Computation phase. We multiply different weights between the costs respectively and obtain total cost according to the current workload.

We implemented dynamic runtime adaptation with API in RocksDB. There is an event listener that contains a set of call-back functions that will be called when a specific event happens such as a flush or compaction job. This allows developers to implement custom features such as statistic collector or external algorithm by simply adding a custom **EventListener**. In Measurement phase, we collect statistics of the workload within *X* flushes with **OnFlushCompleted()** override call-back function of event listener class. *X* is a tuning knob that adjust a trade-off between accuracy and speed. Next, Computation phase function iterates over the delete-ratio and the memory usage to find optimal design, and it does not affect overall system performance. After finding optimal value, it moves to Transition phase, replace the current options with the optimal value and update delete-ratio and compaction score based on the latest options. We adopted calculation functions for worst-case cost from earlier researches [15, 16].

```
Algorithm 1: Measurement Phase
procedure OnFlushCompleted( flush_info ) override
    UpdateStatistics( stats _, flush_info )
    flush_count_ ← flush_count_ + 1

    if flush_count_ == X_ then
        ComputationPhase( stats_ );
        init_stats();
```

```
Algorithm 2: Computation Phase
procedure ComputationPhase( stats )
    r ← stats.r          // r : proportion of lookups
    v ← stats.v          // v : proportion of range reads
    w ← stats.w          // w : proportion of writes

    for d ← 0 to max_deletes_ratio do
        for step ← 0 to num_of_tries do
            cost ← 0
            // try to get best memory usage
            buffer ← TryMemoryUsage( step )
            // update memory usage for bloom filter and block
            filter ← UpdateFilter( buffer )
            B ← UpdateBlock( buffer )
            cost ← r * GetPointLookupCost( fitler )
            cost ← cost + v * GetRangeLookupCost( B )
            cost ← cost + w * GetUpdateCost( B )
            if min_cost > cost then
                min_cost ← cost
    optimal_value ← UpdateValuebyCost( min_cost )
    TransitionPhase( optimal_value )
```

```
Algorithm 3: Transition Phase
procedure TransitionPhase( optimal_value )
    // update del_base_ratio with optimal value
    options_.del_base_ratio ← optimal_value.del_base_ratio
    // update deletes ratio and compute
    // compaction score based on the latest option
    vstorage_→CalculateDeletesRatio(options_)
    vstorage_→ComputeCompactionScore(options_)
```

```
Algorithm 4: Evalation of false positive rate of a bloom filter
procedure eval( f )
    return  $e^{-(f\rightarrow bits / f\rightarrow entries)\cdot(\ln 2)^2}$
```

```
Algorithm 5: Worst-Case PointLookupCost
procedure GetPointLookupCost( filter )
    cost ← 0
    for i ← 0 to num_levels do
        cost ← cost + eval( fileter [i] ) * num_of_runs[i]
    end for
    return cost
```

```
Algorithm 6: Worst-Case RangeLookupCost
procedure GetRangeLookupCost()
    cost ← 0
    for i ← 0 to num_levels do
        // B : # of entries in a block, s : target range size
        cost ← cost + s * entries[i] / B + num_of_runs[i]
    end for
    return cost
```

```
Algorithm 7: Worst-Case UpdateCost
procedure GetUpdateCost()
    cost ← 0
    for i ← 0 to num_levels do
        // B : # of entries in a block
        cost ← cost + num_of_runs[i] / B
    end for
    return cost
```

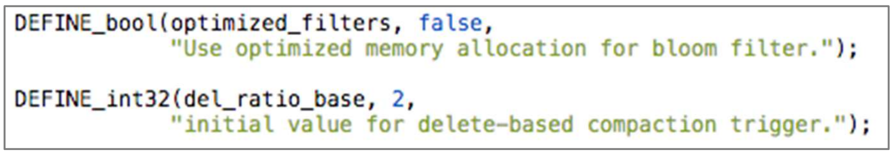
Figure 8. Pseudo codes of dynamic runtime adaptation and cost functions to get optimal memory usage and delete-ratio value according to the workload

3.2 Extending db_bench

RocksDB provides a benchmark tool **db_bench** to evaluate performances. Because the **db_bench** supports many benchmarks to generate different types of workloads, and its various options, it requires a lot of trials to find the optimal configuration.

3.2.1 Flags

To find the optimal configuration of our system, we added some flags to change options easily as shown in Figure 9 and tried many benchmarks to find the optimal settings. RocksDB uses gflags [2], the commandline flags library used within Google, to handle flags automatically by simply adding gflags DEFINE. We defined two flags for our system: **FLAGS_del_base_ratio** to initialize value of delete-based compaction trigger, **FLAGS_optimized_filter** to determine whether to optimize memory allocation for bloom filter or not, which is presented in section 3.5.

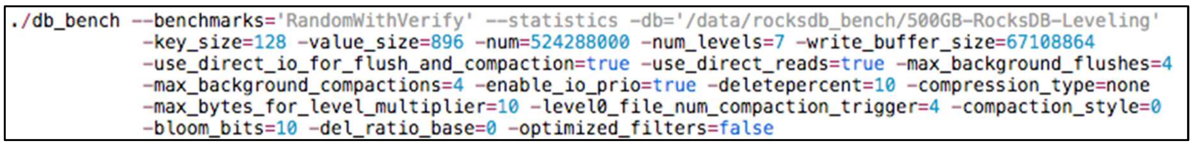


```
DEFINE_bool(optimized_filters, false,
            "Use optimized memory allocation for bloom filter.");

DEFINE_int32(del_ratio_base, 2,
            "initial value for delete-based compaction trigger.");
```

Figure 9. Custom flags for benchmarks of our system

3.2.2 Configuration of Delete-Aware RocksDB

The **db_bench** provides very many tuning options and some of options may have a significant impact on performance such as **num_levels**, **max_bytes_for_level_multiplier** for size ratio and **level0_file_num_compaction_trigger**. We tried various configurations and determined ours as shown in Figure 10, We can configure **del_ratio_base**, **optimized_filters**, **num_levels**, **compaction_style** and **level0_file_num_compaction_trigger** depending on the system: 1) Default RocksDB, 2) Delete-Aware RocksDB by setting **del_ratio_base** over 0, 3) Monkey [15] by setting **optimized_filters** true and 4) Ours+Monkey by using both **del_ratio_base** and **optimized_filters**. The other options that affect performance but are not relevant directly to this research such as the number of sub-compactions, direct IOs and statistics are set the same.



```
./db_bench --benchmarks='RandomWithVerify' --statistics -db='/data/rocksdb_bench/500GB-RocksDB-Leveling'
           -key_size=128 -value_size=896 -num=524288000 -num_levels=7 -write_buffer_size=67108864
           -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -max_background_flushes=4
           -max_background_compactions=4 -enable_io_prio=true -deletepercent=10 -compression_type=none
           -max_bytes_for_level_multiplier=10 -level0_file_num_compaction_trigger=4 -compaction_style=0
           -bloom_bits=10 -del_ratio_base=0 -optimized_filters=false
```

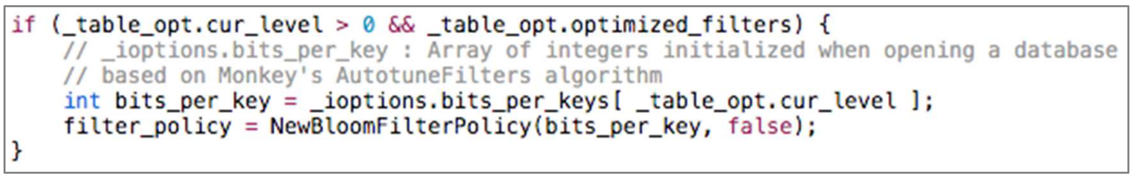
Figure 10. A configuration of default RocksDB with Leveled Compaction

3.3 Cooperation with Monkey Method

The Monkey [15] is a one of the state-of-the-art systems with respect to read performance by optimizing the memory allocation for bloom filter. Existing system used fixed number of bits-per-key to all bloom filter. However, Monkey allocates memory for bloom filter across different levels in order to minimize worst-case lookup IO cost. We found that our system reduces superfluous compactions with delete-based compaction trigger and can store more data at smaller levels compared to existing system. In other words, the actual capacity at smaller levels of our system is larger than existing system. Thus, the main memory of our system occupied by bloom filters should be carefully utilized because the memory may be insufficient as the capacity at smaller levels increases. For that reason, our system adopts Monkey method and has a good synergy with the Monkey method.

Configuration of Monkey

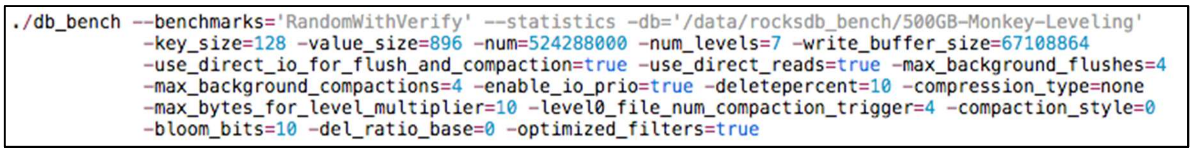
As shown in Figure 11, We simply implemented by assigning different **bits_per_key** values for each level using array of integers rearranged by AutotuneFilters algorithm presented in the Monkey paper. We can assign different **bits_per_key** using **bloom_bits** option in **db_bench**. AutotuneFilters algorithm contains evaluation function that evaluates the false positive rate of a bloom filter for each trial, and each trial allocates a different amount of memory to each level.



```
if (_table_opt.cur_level > 0 && _table_opt.optimized_filters) {
    // _ioptions.bits_per_key : Array of integers initialized when opening a database
    // based on Monkey's AutotuneFilters algorithm
    int bits_per_key = _ioptions.bits_per_keys[ _table_opt.cur_level ];
    filter_policy = NewBloomFilterPolicy(bits_per_key, false);
}
```

Figure 11. Assigning a different number of bits-per-key for bloom filters according the current level

We extend **db_bench** in order to support Monkey method by adding a flag **optimized_filters**. If **optimized_filters** flag is set true as shown in Figure 12, memory allocation for bloom filters will be rearranged when opening a database.



```
./db_bench --benchmarks='RandomWithVerify' --statistics -db='/data/rocksdb_bench/500GB-Monkey-Leveling'
           -key_size=128 -value_size=896 -num=524288000 -num_levels=7 -write_buffer_size=67108864
           -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -max_background_flushes=4
           -max_background_compactions=4 -enable_io_prio=true -deletepercent=10 -compression_type=none
           -max_bytes_for_level_multiplier=10 -level0_file_num_compaction_trigger=4 -compaction_style=0
           -bloom_bits=10 -del_ratio_base=0 -optimized_filters=true
```

Figure 12. A configuration of Monkey with Leveled Compaction

IV. Benchmarks

We now evaluate Delete-Aware RocksDB against existing systems such as RocksDB and Monkey. We introduce experimental setup and results of benchmarks. We compare throughput, space amplification and the number of compactions.

4.1 Experimental Setup

We use a machine with 5TB 7200 RPM disk connected through a SATA 3.1 supported 6.0 GB/s, 64 GB DDR4 main memory, six 3.5 GHz cores with 15MB L3 caches and running 64-bit CentOS 7.7. We allocated half of the threads for compaction and issuing reads/writes/deletes respectively.

Default Workload and Setup

To evaluate worst-case performance, we compare systems under workloads consist of uniformly randomly distributed operations such as reads, writes and deletes. We vary the proportion of deletes from 0 to 10 to test that each system can show stable throughput in an environment where many redundant records are generated. Each default experiment setup is as follows. We insert 500 GB of 1 KB key-value pairs where key size is 128 bytes and corresponding value is 896 bytes, which is common in practice [13, 16]. We repeat each experiment to evaluate throughput three times and measure the average performance. We also compare the systems across workload skews in section 4.4.

4.2 Throughput

We show that Delete-Aware RocksDB improves throughput compared to RocksDB by changing size-based compaction trigger to delete-ratio based compaction trigger. We set up this experiment by repeating multiple tines depending on deletion rate.

In Figure 12, Delete-Aware RocksDB, notated as Ours, shows almost doubled performance in all deletes rate. By replacing with delete-ratio based compaction trigger, our system can reduce superfluous compactions and store more data at smaller levels. Additionally, new compaction trigger responds flexibly to the workloads. However, one of the state-of-the-art system Monkey shows better performance than our system. Monkey assigns exponentially decreasing False Positive Rates to filters at smaller levels. As a result, Monkey shows stable lookup latency and dominates RocksDB's performance. Fortunately, the margins between Monkey and our system are narrow, specifically on 4 % deletes in workload. As we mentioned in section 3.5, we investigated cooperation with Monkey on our system and it will be a good synergy as actual capacity of smaller levels increases. As shown in Figure 12, our system plus Monkey shows better throughput than existing systems. The same algorithm presented in Monkey paper was adopted even though the trigger was replaced, but it shows better performance. With detail of configuration and appropriate strategy, our system plus Monkey can be further improved.

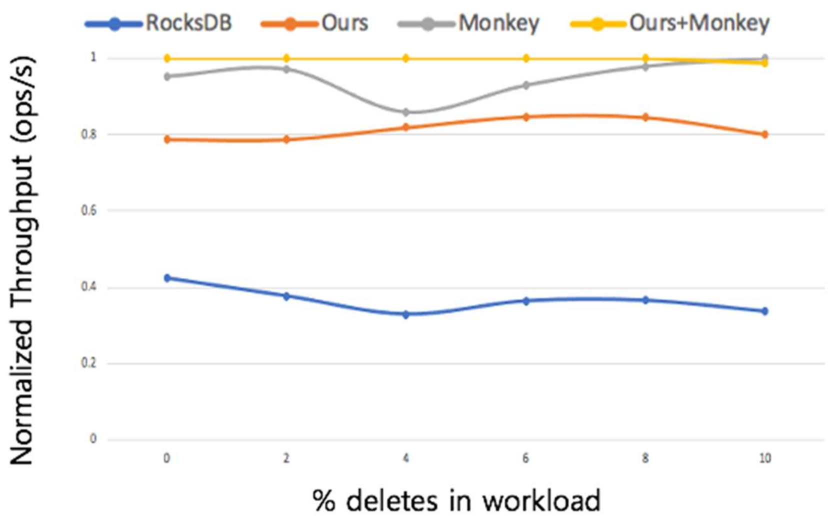


Figure 13. Delete-Aware RocksDB shows doubled performance compared to RocksDB and our system cooperated with Monkey outperforms existing systems in terms of throughput

4.3 The Number of Compactions and Space Amplification

Future 13 shows the comparisons of space amplification and the number of compactions between systems as a result of each experiment. The number of compactions is a knob that can be adjust space amplification and have impact on performance. Generally, the more writes consisting of compactions, the lower space amplification. Since Delete-Aware RocksDB reduces the number of compactions that are superfluous at lower levels, it shows the lower space amplification additional more writes. In this experiment, out system shows fewer compactions compared to RocksDB by up to 13%, to Monkey by up to 8%.

Space amplification, the ratio of the size of database files on disk to data size, is very important to make system improve for overall worst-case cost of LSM-tree as shown in Table 1. As the proportion of the updates operation consisting of inserting existing entries and deletions increases, the system contains more redundant records, as a result, space amplification increases. We evaluate space amplification with two different compaction policies such as Leveled Compaction and Size-Tiered Compaction. We found that our system shows lower space amplification compared to Size-Tiered Compaction. Size-Tiered Compaction has high throughput but poor space amplification and lookup latency because it entails re-writing of the input of a compaction and contains multiple sorted runs in a level. However, Leveled Compaction has better lookup latency and space amplification because one sorted run is stored in a level and few files are used for the compaction, but it has very poor throughput. Delete-Aware RocksDB shows better throughput while maintaining lower space amplification than Size-Tiered Compaction of RocksDB. As a result, our system achieves a lower space amplification by reducing superfluous compaction without degrading performance.

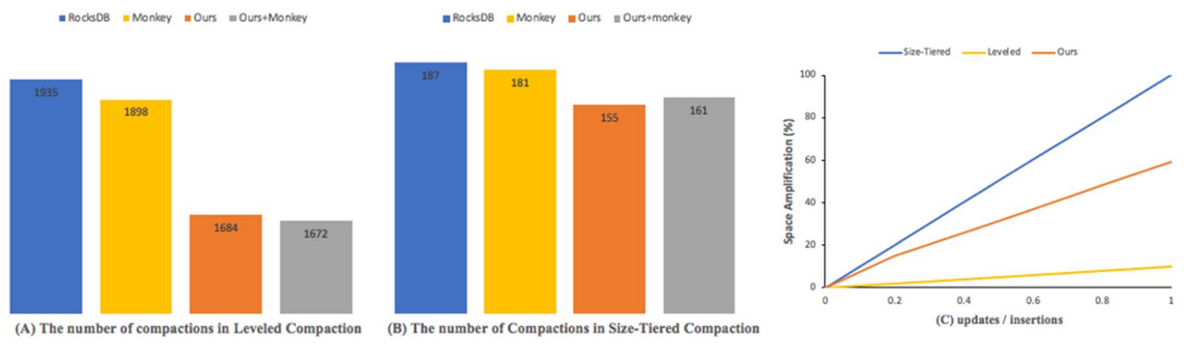


Figure 14. Delete-Aware RocksDB shows fewer compactions and lower space amplification than Size-Tiered Compaction.

V. Discussion

Picking files strategy

Depending on the strategy for picking files, it affects compaction efficiency and performance. Since RocksDB uses level 0 as special level to store just flushed from in-memory *memtable*, they use a strategy that append a level 0 file that overlap with the files in compaction. By doing so, it can reduce additional compactions from level 0 to 1, and it makes redundant records reach to the largest level faster. With memory trade-off for block range in pre-remove strategy of our system, we can use fine-grained test to find level 0 files overlapped. As a result, it enables to make compaction efficient.

Custom optimization algorithm of Monkey's AutotuneFilters to suit new compaction trigger

Even though most of cases shows good synergy with Monkey but, in some cases, we investigated that our system with Monkey method shows degrades performance compared to Monkey itself. We found that AutotuneFilters function of Monkey allocates too much amount of main memory for bloom filter at lower levels since the actual capacity of lower levels increased by delete-ratio compaction trigger. Since it can lead to degrades performance as the latency of reads in larger levels increased, we need custom algorithm that fit our delete-ratio compaction trigger.

VI. Conclusion

We show that LSM-tree based key-value stores conduct compaction in order to remove redundant records and to bound the number of sorted runs in a level. However, compactions that are triggered at smaller levels with size-based compaction trigger can be not only superfluous compaction between most valid data, but also inflexible compaction for workloads. To address this problem, we present Delete-Aware RocksDB, an LSM-tree based key-value store that responds flexibly to workloads and reaches lower space amplification compacted to existing systems. Delete-Aware RocksDB uses delete-ratio based compaction trigger so as to respond flexibly to workloads, pre-remove strategy for redundant tombstones with memory trade-off and dynamic runtime adaptation to maximize performance under the given the amount of memory and workload. Our system shows better throughput, fewer compactions and low space amplification compared to existing systems.

References

[1] Facebook. *Controlling space amplification*. URL: https://www.facebook.com/notes/rocksdb/controlling-space-amplification/351059228369285/ (visited on 03/11/2020).

[2] Google. *gflags*. URL: https://github.com/gflags/gflags (visited on 27/10/2020).

[3] Facebook. *RocksDB*. URL: https://rocksdb.org/ (visited on 09/09/2020).

[4] Google. *LevelDB*. URL: https://github.com/google/leveldb (visited on 08/10/2020).

[5] Facebook. *RocksDB RocksDB-Tuning-Guide Wiki*. URL: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide (visited on 28/10/2020).

[6] Facebook. *RocksDB: CalculateBaseBytes()*. URL: https://github.com/facebook/rocksdb/blob/881e0dcc09f80fa09273c0750ba493d64ada6286/db/version_set.cc (visited on 02/11/2020).

[7] Apache. *Cassandra*. URL: http://cassandra.apache.org (visited on 03/10/2020).

[8] Facebook. *MyRocks*. URL: http://myrocks.io/ (visited on 03/10/2020).

[9] Scylla. *ScyllaDB*. URL: https://www.scylladb.com/2018/01/17/compaction-series-space-amplification/ (visited on 23/11/2020).

[10] Fitzpatrick, B., Vorobey, A. "Memcached: a distributed memory object caching system", 2011.

[11] Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., Callaghan, M. "Designing Access Methods: The RUM Conjecture" *In EDBT*, 2016, pp. 461-466.

[12] Warlo, H.W.K. "Auto-tuning RocksDB", *MS Thesis*, Northwegian University of Science and Technology, Trondheim, Norway, 2018, 86 pages.

[13] T.G. Armstrong et al. "LinkBench: a database benchmark based on the Facebook social graph" *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp.1185-1196.

[14] Mohan, J., Kadekodi, R., Chidambaram, V. "Analyzing IO Amplification in Linux File Systems" *arXiv:1707.08514*, 2017

[15] Dayan, N., Athanassoulis, M., Idreos, S. "Monkey: Optimal navigable key-value store" *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 79-94.

[16] Dayan, N., Idreos, S. "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value

stores via adaptive removal of superfluous merging" *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 505-520.

[17] Bu, Y., Borkar, V., Jia, J., Carey, M.J., Condie, T. "Pregelix: Big(ger) graph analytics on a dataflow engine" *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2014, pp. 161–172

[18] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M. "Optimizing Space Amplification in RocksDB" *In Proceedings of the Biennial Conference on Innovative Data Systems Research CIDR*, 2017 p. 3.

[19] Chen, G.J., Wiener, J.L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Yilmaz, S. "Realtime data processing at Facebook" *In Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1087-1098.

[20] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Vogels, W. "Dynamo: amazon's highly available key-value store" *ACM SIGOPS operating systems review*, 41(6), 2007, pp. 205-220.

[21] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Gruber, R.E. "Bigtable: A distributed storage system for structured data" *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008, pp. 1-26.

[22] Bloom, B.H. "Space/Time Trade-offs in Hash Coding with Allowable Errors" *Communications of the ACM (CACM)*, 13(7), 1970, pp. 422–426

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

[24] Beyer, R., McCreight, E.M. "Organization and maintenance of large ordered indices" *Acta Informatica*, 1(3), 1972, pp. 173-189.

요 약 문

LSM-Tree 기반의 키-값 저장소를 위한 삭제 인지 컴팩션 트리거 방법

LSM 트리 기반 키-값 저장소는 out-of-place 업데이트 구조로 인해 쓰기 집약적인 워크로드에서 높은 성능을 보인다. 그러나 이 구조는 데이터베이스에 많은 중복 데이터의 저장을 필요로 하며, 결과적으로 높은 공간 증폭이 발생한다. 중복 데이터를 제거하기 위해 대부분의 최신 LSM 트리 저장소는 트리 각 레벨의 용량에 따라 트리거되는 컴팩션을 사용하지만, 이는 현재 워크로드에 따른 컴팩션 주기를 결정하기 어렵다. 잦은 컴팩션은 높은 쓰기 증폭을 수반하여 전반적인 성능을 감소 시킬 수 있어 중요하다. 이를 해결하기 위해 현재 워크로드의 삭제 비율을 반영하는 삭제 인식 컴팩션 트리거를 제안한다. 삭제 비율 기반 압축 트리거를 통해, 기존의 크기 기반 압축 트리거로 인해 발생하는 추가적인 컴팩션 및 대부분의 유효한 데이터 간의 비효율적인 컴팩션을 줄인다. 본 논문은 RocksDB 를 활용해 Delete-Aware RocksDB 를 구현했으며, 쓰기 증폭 비용없이 낮은 공간 증폭을 유지하여 RocksDB 에 비해 성능이 우수함을 보인다. 또한, 최신 성능의 LSM 트리 기반 키-값 저장소 중 하나인 Monkey 방법과의 협력을 통한 높은 성능을 보인다.

핵심어: LSM-Tree 기반의 키-값 저장소, 컴팩션 트리거, RocksDB