LTG-LSM: The Optimal Structure in LSM-tree Combined with Reading Hotness

JiaPing Yu[1], HuaHui Chen[1], JiangBo Qian[1], YiHong Dong[1]
1. Department of Information Science and Technology, Ningbo University
Ningbo, China
{1811082061, chenhuahui, qianjiangbo, dongyihong}@nbu.edu.cn

2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-7281-9074-7/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICPADS51040.2020.00011

***Abstract*—A growing number of KV storage systems have adopted the Log-Structured-Merge-tree (LSM-tree) due to its excellent write performance. However, the high write amplification in the LSM-tree has always been a difficult problem to solve. The reason is that the design of traditional LSM-tree under-utilizes the data distribution of query, and the design space does not take into account the read and write performance concurrently. As a result, we may sacrifice one to improve another performance. When advancing the writing performance of the LSM-tree, we can only conservatively select the design pattern in the design space to reduce the impact on reading throughputs, resulting in limited improvement. Aiming at the shortcomings of existing methods, a new LSM-tree structure (Leveling-Tiering-Grouped-LSM-tree, LTG-LSM) is proposed by us that combined with reading hotness. The LTG-LSM structure maintains hotness prediction models at each level of the LSM-tree. The structure of the newly generated disk components is determined by the predicted hotness. Finally, a specific compaction algorithm is carried out to handle the compaction between the different structural components and processing workflow hotness changes. Experiments show that the scheme proposed by this paper significantly reduces the write amplification (up to about 71%) of the original LSM-tree with almost no sacrificing reading performance and improves the write throughputs (up to about 24%) in workflows with different configurations.**

Keywords-LSM-tree; KV Store; Storage Management; Hotness Prediction; Big Data

I. INTRODUCTION

The demand for big data storage and query has exposed the deficiencies of traditional relational database systems, which has led to a new type of complementary non-relational data storage, NoSQL. The NoSQL technology complements the relational database, which has the advantages of fast update and query speed, large-scale storage support in distributed systems, natural expansion, and low disk cost [1]. These benefits are suited to the storage and management of big data. The Log-Structured-Merge-tree (LSM-tree) is the most widely used structure among many NoSQL database storage layers. It caches the updates in memory, then uses sequential I/O to persist the updates to disk and compact them based on a particular strategy. This storage structure has the advantages of better write performance, higher space utilization, more straightforward concurrency control, and more accessible data recovery. Consequently, it is in line with the era of big data with various and heavy workflow. Google's BigTable and LevelDB, Apache's HBase and AsterixDB, Facebook's RocksDB and Cassandra, etc., these NoSQL databases all use the LSM-tree as their bottom storage structure.

The high write amplification (WA) in the LSM-tree has always been a difficult problem to solve, and the value of it affects the overall write performance. The write amplification in the LSM-tree can be defined as follows:

$$WA = \frac{Total\ data\ size\ to\ read\ and\ write}{Size\ of\ record} \quad (1)$$

where the denominator indicates the size of the record written by the user, and the numerator indicates the amount of disk reading and writing caused by writing the data to the database. Obviously, the write performance of the system deteriorates with the increase of write amplification, and the service life of the hard disk is also greatly affected.

The current research work mostly reduces the write amplification by changing the structure of the LSM-tree and the compaction strategy [2], without utilizing the read hotness information of the data. Since the skewness of data access is quite common in reality [3], failure to make use of this information causes these works to have relatively limited write performance enhancement and is usually accompanied by sacrifices in read performance. Even in some cases, the system is not as stable as it could be.

Aiming at handling the intractable contradiction between write and read performance optimization in the LSM-tree, this paper proposes a novel heterogeneous structure of LSM-tree (Leveling-Tiering-Grouped-LSM-tree, LTG-LSM) combined with data reading hotness. In the design of the LTG-LSM structure, hotness prediction models are maintained at each level. The fresh generated disk components determine the structure through the predicted hotness by these models. Then a specific elaborate compaction algorithm is used to handle the merge operations between different structural components and to deal with the situation where the workflow hotness changes, making the update and query abilities of the LSM-tree approach the optimal design.

The rest of the paper is organized as follows: Section II briefly introduces the structure of the LSM-tree and our motivation for improvement. Section III introduces the implemental details of the LTG-LSM structure. Section IV implements this scheme on LevelDB for evaluation. Section

2690-5965/20/$31.00 ©2020 IEEE
DOI 10.1109/ICPADS51040.2020.00011

V summarizes the current improvement works on the LSM-tree. Finally, Section VI summarizes the work of this paper.

II. Background and Motivation

A. Background

The LSM-tree [4] was proposed in 1996, and its purpose is to solve the problem that the write performance, read performance, and space utilization in the log-storage-structure are challenging to balance. In modern LSM-based storage architectures, SSTable (sorted-string-table) is mostly used as the storage unit of the disk, which was initially proposed in Google's BigTable. This paper takes LevelDB as an example to show the design pattern of modern LSM-tree (Figure 1). The components residing in memory include a MemTable and an ImmutableMemtable; the files in external storage include some SSTable files, a Log file, a Current file, and a Manifest file. When the user writes a KV record, LevelDB will first save the data in the Log file and insert it into the MemTable after success. The Manifest file is used to record the meta information of the SSTable, and the Current file is used to point to the current Manifest file.

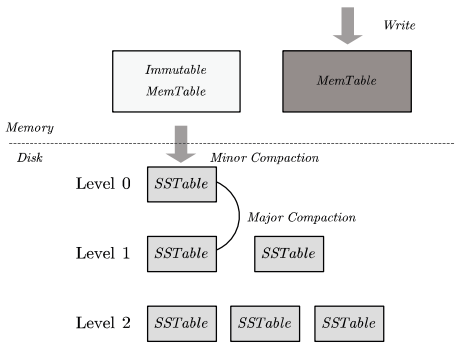

Fig 1. The architecture of a modern LSM-tree using LevelDB as an example

When the MemTable in memory is too large, LevelDB will generate a static snapshot ImmutableMemTable. At the same time, a new MemTable is created to receive the latest updates. The background scheduling thread is responsible for saving this snapshot in level 0 of the disk as SSTable for persistent storage, which is called minor compaction. When the file size of an individual level on the disk reaches its capacity, the system will compress and merge it with the SSTable of the next level, and the newly generated SSTable resides in the next level. This process is called major compaction. The capacity of level $i+1$ is usually T times larger than the level i, where T is called the capacity increasing ratio.

Reading takes more time than writing because the various components (i.e., SSTable) in the system need to be checked in order according to the oldness of the data. The most recent data is held in memory, so it is retrieved first. And then, the components in the disk will be checked in order from low levels to high levels in order of hierarchy.

The structure of the LSM-tree can be generally divided into leveling and tiering. The former is more conducive to reading, and the latter is more beneficial for writing. A leveling structure represented by LevelDB maintains one component at each level and operates compaction between adjacent levels. A tiering structure represented by FlameDB maintains T components as a common group at each level. The components in a group are not strictly ordered, and compaction is operated at the same level.

Modern LSM-based databases use the leveling strategy to maintain the balance of update and query delays mostly. Dayan et al. [5] believe that frequent compaction at low levels increases a lot of update cost but does not improve query performance significantly, so the Lazy-Leveling strategy is used in Dostoevsky, which only uses leveling at the last layer. In contrast, tiering structures are used for other levels. It combines two structures to achieve a compromise effect while preferring to improve writing performance. Above all, the heterogeneous strategy adopted by Dostoevsky is coarse-grained and data-independent.

Additionally, FlameDB [6, 7] is an implementation of the tiering structure. It does not even use the partitioning strategy itself, which will cause the allocation of I/O cost to be uneven among compactions. In experiments, we have found that the naive tiering strategy may cause the system to lose response for a long time because of too many components involved in a compaction operation, which is not allowed in practice applications.

B. Motivation

We believe that the reason why the tiering structure will seriously affect the read performance and make it challenging to apply to the actual database is that the structure does not consider the access hotness information of the records. For example, if a tiering group is hot and accessed frequently, then the tiering strategy is inappropriate. Because the key ranges of each tiering group overlap, this will cause all the components in a tiering group to be accessed frequently, which seriously exacerbates the read performance. Therefore, the tiering structure should be applied to components with lower access hotness in the ideal case, while other components are still leveling. In this way, one can reduce the write amplification and minimize the negative effect on the read performance.

Motivated by the above, we propose a new structure of LSM-tree which has a heterogeneous organization strategy (LTG-LSM). In consideration of the data hotness information, the organization strategy of the original LSM-tree is divided into finer-grained. Specifically, the hot components are organized with leveling structure, and the rest is tiering. The main goal of the framework is to improve write performance as much as possible without losing read performance.

Next, we analyze the I/O consumption in some typical cases. We assume that the capacity increasing ratio of the LSM-tree is T, and the number of levels is $L+1$. To simplify the analysis, level 0 is used as a memory buffer, and the number of buffer pages is P. Each page can hold an average of B data items. Besides, there are N data items that need to be stored, and the size of each data item is E. Since the capacity of each level is T times larger than the previous level, the number of data items stored in the last level is about $N \cdot (T-1)/T$. If the size of the memory buffer allocated by the system is M_{buf}, then L can be expressed as

(2):

$$L = \log_T \left(\frac{N \cdot E}{M_{buf}} \cdot \frac{T-1}{T} \right) \quad (2)$$

For write operations, the write cost for a data item usually refers to the total number of disk I/O from insertion to move to the highest level. In the leveling structure, the components of each level need to be compacted $T-1$ times to reach the capacity threshold of this level. While for the tiering type, only one compaction occurs when the capacity of the layer reaches the threshold. Thus for the leveling structure, the cost of each write operation is $O(T \cdot L/B)$. And for the tiering structure, the cost of each write operation is $O(L/B)$.

For read operations, each component in the system needs to be checked in sequence, and the I/O cost is the number of components in the system. So for the leveling structure, the cost of each point query is $O(L)$. And for the tiering structure, the cost of point query is $O(T \cdot L)$. In modern LSM-based storage architectures, Bloom Filter is often used to reduce the I/O cost of reading operations. When the point query prepares to read a disk page to determine whether a specified data item exists, it will first access the Bloom Filter in the memory buffer. One disk access can be saved if the Bloom Filter returns negative. Obviously, the false positive rate (FPR) of the Bloom Filter determines the average cost of the query. When a false positive occurs, the time of disk access will be wasted. The FPR of Bloom Filter can be calculated by (3):

$$FPR = e^{-bits/entries \cdot \ln(2)^2} \quad (3)$$

where $bits/entries$ means that the average number of filter bits assigned to each record. If the value is 10, it can achieve the FPR of about 1%. It should be noted here that when analyzing the query cost, the zero-result lookup is usually used. This is because that zero-result query is common in practical applications [8] and will cause the largest I/O cost. If the total space allocated to the Bloom Filter by the system is M_f, and it has the same false positive rate at each level, then the I/O cost for the point query of the leveling structure and the tiering structure can be expressed as $O\left(L \cdot e^{-M_f/N}\right)$ and $O\left(T \cdot L \cdot e^{-M_f/N}\right)$ respectively.

Assuming that at a certain point in time, the LTG-LSM architecture predicts that the proportion of hot components is $\alpha(\alpha \in (0,1))$, and the proportion of hot components involved in the query is $\beta(\beta \in (0,1))$, then the I/O cost of each update operation is $O\left(\left(\alpha \cdot T + (1-\alpha)\right) \cdot L/B\right)$. With the decrease of α, the update performance is close to the leveling strategy. And the I/O cost of each query operation is $O\left((\beta + (1-\beta) \cdot T) \cdot L \cdot e^{-M_f/N}\right)$, the read performance approaches the tiering strategy with increasing of β. From the above analysis, we can find that α and β are mutually restrictive. When α is small enough and β is large enough, i.e., the hotness prediction is accurate enough, the system as much as possible ensures that most of the queries fall into the predicted hot components. Still, the total number of hot components cannot be too much. Consequently, the loss of read performance due to the tiering structure will be negligible, and the advantage of write performance will be exerted.

The I/O cost depends on the compaction strategy, capacity increasing ratio T, space allocated to the memory buffer M_{buf}, and space allocated to Bloom Filter M_f. Compared with the tiering structure, the leveling structure has better query efficiency but worse update efficiency. The LTG-LSM structure uses the hotness information of data access to make that the query performance of the LSM-tree structure as close as possible to the leveling type and the update performance as close to the tiering type as possible. In other words, if the hotness prediction is accurate enough, the structure is close to optimal design.

III. IMPLEMENTATION

A. Overview of Framework

Figure 2 shows the LTG-LSM framework proposed in this paper. The organization of components is related to the hotness of data access, rather than merely dividing them by levels. Note that in LTG-LSM, we use Group to manage all components. Assuming that there are $L+1$ levels in the structure, the *Model Manager* is responsible for maintaining the hotness prediction model $M_l(l = 0,2,\ldots,L)$ of each level. The models predict the hotness of a newly-built component according to the data distribution. At the same time, the models maintained by the *Model Manager* will be retrained at intervals to retain their accuracy. The *Structure Recorder* is used to record the metadata of the Group organization information. When the hotness exceeds the threshold θ, the newly generated components will be organized in a leveling manner. Otherwise, they will be leveling. Finally, due to changes in the organizational structure of the components, the original compaction method of LSM-tree needs substantial adjustments. The *Compaction Controller* is used to implement the compaction strategy of the components in the LTG-LSM structure to prevent space waste and query deterioration caused by data accumulation.

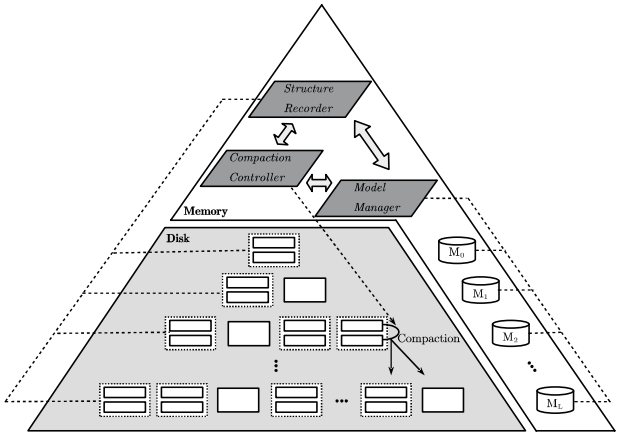

Fig 2. Overview of LTG-LSM framework

The advantages of the LTG-LSM framework are as follows:

1) Data access in real life always follows a specific pattern, such as Zipfian, Hotspot, etc. These patterns will cause the access hotness of different components to be extremely uneven. This phenomenon is apparent even in uniformly distributed reading workflows. Rather than waste this information like other data-independent strategies, the organizational structure of the components in LTG-LSM is determined by the data distribution and query distribution.

2) The hotness threshold θ can be used as an adjustable parameter for users to adjust according to actual needs. The larger it is, the smaller the number of leveling organized components in the system, and the overall organization is closer to the tiering type, i.e., the system is more focused on write performance. At this time, users can speed up the reading process of the database by adding Bloom Filters.

B. Model Manager

We regard a series of Get requests as a logical timeline. The most recent time window τ represents several requests closest to the current Get request, and the length of τ is W. The advantage of this design is that the concept of "recent time window" itself reflects the temporal locality of a stable workflow, with the number of accesses to components that are monotonous with the increase in Get requests. Still, only the amount of access in the most recent time window can more accurately reflect the hotness of a component.

First, we need to define component hotness based on component access. The access of the k-th component of the level l at time point i is defined as:

$$access_{l,k}^{i} = \begin{cases} 1 & \text{,if the component is accessed} \\ 0 & \text{,else} \end{cases} \tag{4}$$

Assuming that the current time point is c, the current hotness $f_{l,k}$ of the k-th component at the level l is defined as:

$$f_{l,k} = \left(\sum_{i=c-W+1}^{c} access_{l,k}^{i} \right) / W \tag{5}$$

Then we implement the maintenance of the model. As mentioned above, the access of components in the LSM-tree behaves much unevenly among layers, so we build multiple models according to the granularity of the levels to prevent large errors caused by using the only individual model. In the representation of the data distribution, the space of key range R is divided into consecutive intervals $\{r_1, r_2, \dots, r_e\}$ which have the same size, then the data distribution of a particular component can be expressed as a vector $\varphi = (d_1, d_2, \dots, d_e)$, which d_i represents the number of data in the range of the key r_i. Then the problem can be transformed into using φ to predict the hotness value f'. Considering the high real-time requirements in the LSM-tree structure, we use linear regression here with faster training speed to build the model $M_l (l = 0,1, \dots, L)$ of each level.

Our goal is to build a model M_l at each level of the LSM-tree, so that the model can predict the access hotness $f'_{l,k}$ of the component in the future through the data distribution of the newly generated component at the level l, i.e., the goal is to obtain $M_l(\varphi_{l,k}) = \omega_l \cdot \varphi_{l,k} + b_l$, so that $M_l(\varphi_{l,k}) \approx f_{l,k}(k = 1,2, \dots, N(l))$, where ω_l is a vector of length e and b_l is a scalar. The cost function of each level J_l can be expressed as:

$$J_l(\omega_l, b_l) = \min_{\omega_l, b_l} \sum_{l=0}^{L} \sum_{k=1}^{N(l)} \left(M_l(\varphi_{l,k}) - f_{l,k} \right)^2 \tag{6}$$

The advantage of linear regression is that the optimal parameter solution can be obtained in constant time. This feature is particularly crucial for LSM-tree with high write and read performance requirements.

Besides, when the hotness condition of the workflow changes, it is necessary to retrain the model periodically to avoid the difference between the predicted hotness and the actual hotness, affecting the read and write performance. The adjustment of the model is just the accumulation of $f_{l,k}$ and $\varphi_{l,k}$, then the recalculation of the model parameters b_l and ω_l. Therefore, the model parameters are updated efficiently when using regression models.

C. Structure Recorder

For functional reasons, we divide the component data structure in LTG-LSM into two categories, Leveling-Group and Tiering-Group, which can be seen from Figure 3. The gray boxes in the figure represent a component, the dotted boxes represent the logical grouping of components, and the numbers in components represent the key ranges of data stored in. The Leveling-Group maintenance key ranges do not overlap each other so that these components can be regarded as a component with a more extensive key range logically.

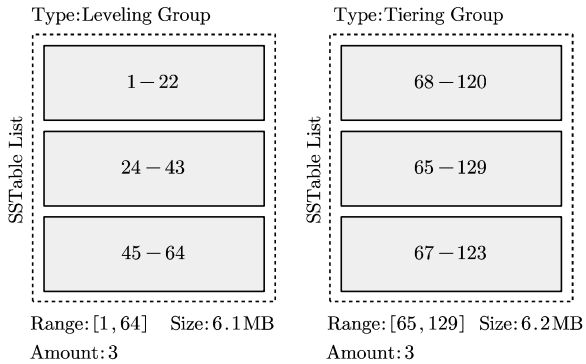

Fig 3. Two data structures in the LTG-LSM framework

Leveling-Groups are more conducive to data reading due to the non-overlapping key ranges, but the maintenance of non-overlap will affect write performance absolutely. The Tiering-Groups are the opposite. These components are generated in the compaction process of the previous layer. In

addition, the metadata to be maintained for the two data structures is:

- *Type*, used to indicate whether the Group belongs to Leveling-Group or Tiering-Group.
- *SSTable List*, used to save the component information contained in the group, components Metadata is consistent with LevelDB.
- *Range*, save the overall key range of all components in the group; this information is critical during the merge process.
- *Size*, save the size of all components in the group, used to estimate the degree of filling of a specific level.
- *Amount*, save the number of components contained in the group, used to trigger the merge of a Tiering-Group.

D. Compaction Controller

We implement the compaction algorithm of LTG-LSM in this section. Many LSM-based databases used to optimize compaction using partitioning strategies, which refer to splitting components into small parts that are not overlapping in key range. We emphasize the purpose of the partitioning strategy is to spread the I/O overhead of each compaction, which will not have an impact on the total I/O cost. We also use the partitioning strategy in the LTG-LSM structure.

To begin with, we determine the conversion method between the two organization methods. As shown in Figure 4, the Leveling-Group containing n components can be split into n corresponding ranges of Tiering-Group, without disk I/O and compaction operations. However, it is necessary to read all components from secondary storage in the group when a Tiering-Group is converted into a Leveling-Group. After compaction, the newly generated components are combined into a Leveling-Group.

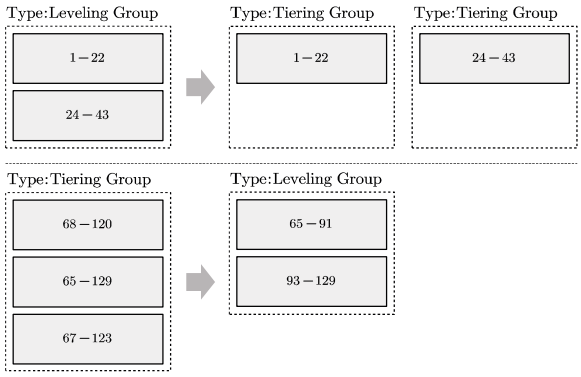

Fig 4. Conversion between two types of Group

When a Tiering-Group g_t at the level l operates compaction, the new components are generated from g_t according to the key range of Group at the level $l+1$. Then these new components are added to the corresponding Group. Finally, the Group may adjust its structure according to the hotness of the new components that attended in, i.e., transform Tiering-Group into Leveling-Group or the opposite. If a new component is added to a Leveling-Group, all the overlapping components need to be compacted, and they will still join the same Group after the compaction.

When a Leveling-Group g_l at the level l operates compaction, all Groups having overlapping keys with g_l will be selected at the level $l+1$. Then components included by these Groups will be merged and sorted to generate new components. The new components are added to the Groups at the level $l+1$ with the same range and adjust the structure according to the hotness.

Algorithm 1 Compaction in TLG-LSM

```
1:  function PickCompaction(g)
2:      result ← ∅
3:      if g.Type = Tiering then
4:          result.append(g.SSTableList)
5:      else
6:          og ← GetOverlapGroup(g)
7:          for t ∈ og do
8:              result.append(t.SSTableList)
9:          end for
10:     end if
11:     return result
12: end function
13:
14: function AddFiles(files, level, CurrentVersion)
15:     for f ∈ files do
16:         for g ∈ CurrentVerion.LevelInfo[level].Group do
17:             if f.Range ⊂ g.Range then
18:                 g.add(f)
19:                 if ifg.Type = Leveling then
20:                     new ← Compaction(g.SSTableList)
21:                     g ← NewGroup(new)
22:                 end if
23:                 MayExchangeType(g, f.Type)
24:             end if
25:         end for
26:     end for
27: end function
28:
29: function DoCompactionWork(g, level, CurrentVersion)
30:     set ← PickCompaction(g)
31:     new ← Compaction(set)
32:     AddFiles(new, level, CurrentVersion)
33: end function
```

The description of the LTG-LSM's compaction process is shown in Algorithm 1. The method DoCompactionWork completes the compaction of Group g. When a Group operates compaction, first call the PickCompaction method to select the components to be merged (line 30), and then merge and sort to construct the new components (line 31), finally add the new component to the next level (line32).

IV. Experiment and Evaluation

We implemented LTG-LSM on the open-source database LevelDB and conducted a series of experiments to check its performance.

A. Experimental configuration

The configuration of the experimental platform is shown in Table 1. The version of LevelDB used in the experiment is V1.21. In the experiment, the length of the recent time window τ is $W = 100$, the threshold of hot component is $\theta = 0.7$.

Like other studies, we use the Java version (V0.1.2) of YCSB [9] as a performance measurement tool to evaluate the performance with different workflow configurations. The data items in the database are composed of a series of independent keys and the corresponding randomly generated values. The size of the key-value pair is set to 1KB, and different numbers of records are created to compare the write and read performance of databases at different data scales.

Table 1. The configuration of the experimental platform

OS	Ubuntu 18.04 LTS
Kernel	Linux 4.15.0-74-generic
CPU	Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz
DRAM	DDR4 SDRAM / 16G
HDD	SATA2 / 500GB 7200RPM

B. Benchmark experiment

First, we conduct benchmark experiments to check the performance of the framework. For the reading workflow, we use a Zipfian distribution with a zero-results query ratio of 0.5. Under this distribution, some key-value pairs are more popular than others, which is quite common in real-world scenarios. Figures 6-8 are the results obtained.

Figure 5 and Figure 6 show the results in the write capability between LTG-LSM and LevelDB under different increasing ratios T. As can be seen, the LTG-LSM structure dramatically reduces the I/O cost of the LSM-tree structure during compaction (reduced by about 40%, 56%, 67%, 71%, respectively). Correspondingly, the write throughputs have also been improved to a certain extent (about 9%, 15%, 21%, 24%, respectively). This result is consistent with the I/O cost analysis in Section III. There is a phenomenon that we should note, the increase in write performance gradually slows down with the T increase. The reason for this phenomenon is that with the increase of T, the capacity of the highest level in the LSM-tree becomes large, while this level does not participate in the compaction (the compaction score is always 0). The actual compaction I/O occurs at the levels with smaller capacity, resulting in the performance bottleneck of the LTG-LSM structure.

Figure 7 shows the changes in the read performance of LTG-LSM and LevelDB under different increasing ratios T. As analyzed above, the LTG-LSM structure will slightly reduce the read performance of the LSM-tree. For instance, when querying a *value* corresponding to a specific *key*, you need to select candidate components from level 0 to the highest level. The key range of these candidate components includes *key*. For LTG-LSM, the components in the Tiering-Group have overlapping keys, so if the key range in a Tiering-Group contains *key*, all runs in this Group will be selected as candidate components. There are many optimization skills for read performance in LevelDB fortunately, such as Bloom Filter (we have also optimized the Bloom Filter slightly according to component hotness), block cache, etc. And the LTG-LSM structure itself also makes most of the hot queries fall into the Leveling-Group as much as possible, so the impact on read performance is almost negligible (reduced by about 2.4%, 2.1%, 1.3%, 1.9%, respectively).

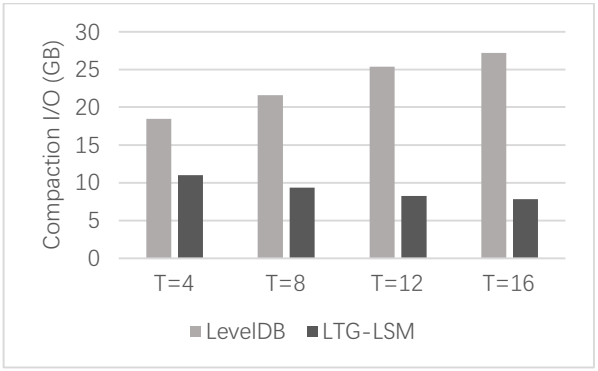

Fig 5. I/O cost of compaction at different increasing ratios

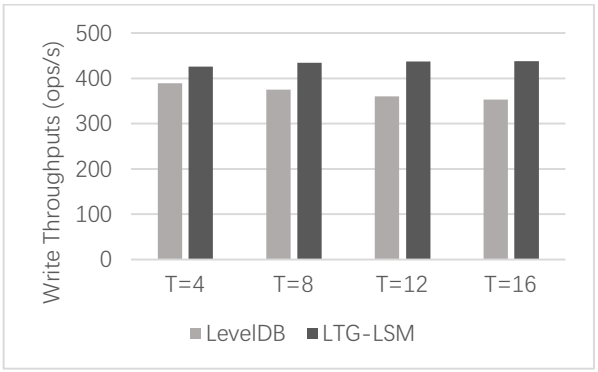

Fig 6. Write throughputs at different increasing ratios

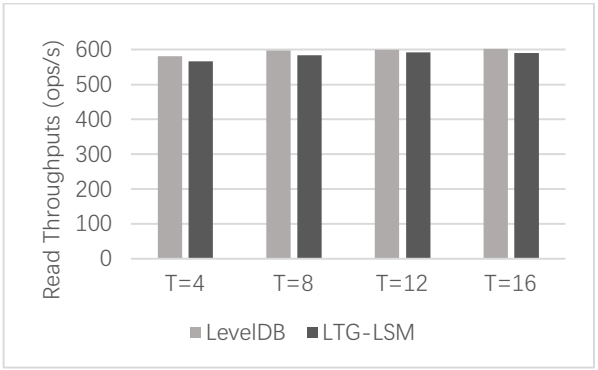

Fig 7. Read throughputs at different increasing ratios

C. Extended experiment

This section conducts some extended experiments to test the scalability of the LTG-LSM structure. The default settings

of the experiment are the same as those in above, among which $T = 12$.

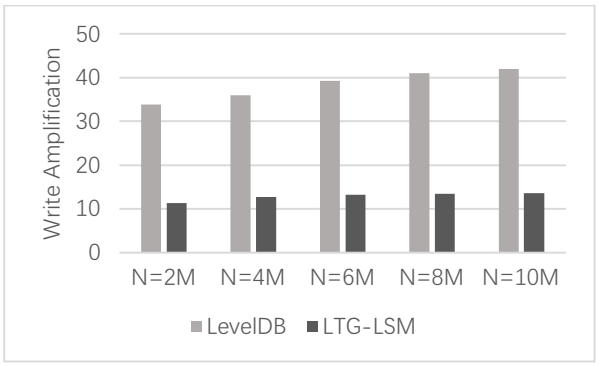

Fig 8. Write amplification at different data scales

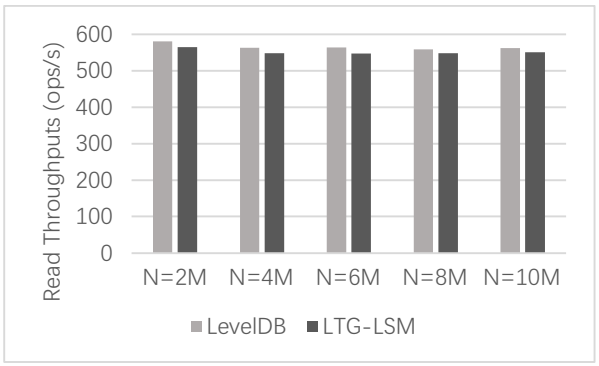

Fig 9. Read throughputs at different data scales

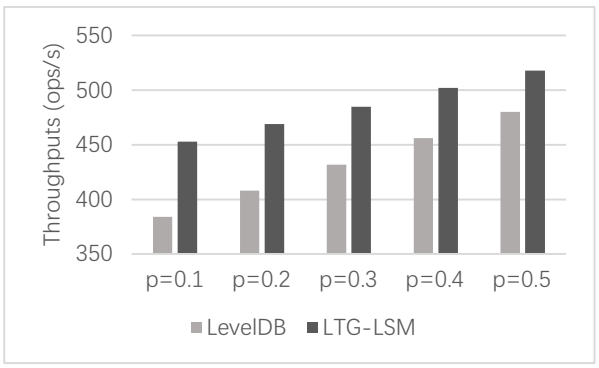

Fig 10. Overall throughputs at different r/w rates

1) Experiments at different data scales. We adjusted the values of N to detect the performance of the LTG-LSM structure in datasets of different scales. The experimental results are shown in Figures 8-9. Due to the vast difference in I/O caused by constructing datasets of different scales, this experiment compares the write amplification of the system defined by (1). As can be seen, LTG-LSM reduces the write amplification at different scales (a reduction of 64.7%-76.3%), while the read performance is hardly affected (the decrease is between 1.97%-3.01%). The write amplification of LevelDB will be more evident at larger datasets because of the large amount of data at the highest level. In contrast, the write performance of the LTG-LSM structure is more independent of the amount of data at the highest level.

2) Experiments with different read-write ratios. We adjusted the value of p to detect the performance of the LTG-LSM structure under workflows with different read-write ratios. Figure 10 shows the results we obtained, which the ordinate is the overall throughputs, including update and query. It can be found from Figure 10 that since LTG-LSM optimizes the write performance, the performance gradually approaches LevelDB itself as the read ratio increases in the workflow. However, the overall throughputs are still significantly improved (by about 18%, 15%, 12%, 10%, 8%, respectively). According to the previous experiments, the promotion effect will be more evident in a larger database.

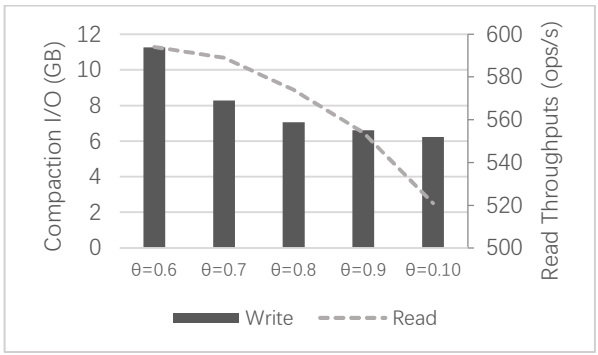

Fig 11. W/A performance at different hotness thresholds

3) Experiments under different hotness thresholds. We adjusted the value of θ to detect the ability of the LTG-LSM structure to balance update and query performance. The experimental results are shown in Figure 11. The left/right ordinates in Fig 11 reflect the write/read performance of the system, respectively. It can be found that the closer it is to 1, the closer the LTG-LSM structure is to the naive tiering structure. At this time, the write amplification is reduced to a minimum, while the loss of read performance is also maximized accordingly. It has the opposite effect when θ becomes smaller.

D. Compare with Lazy-Leveling

We also implemented the Lazy-Leveling strategy [5] in LevelDB to perform performance comparison, and the results are shown in Figure 12. It can be found that the compaction I/O of LTG-LSM under different capacity increasing ratios is less than Lazy-Leveling (reduced by about 4%, 33%, 43%, 47%, respectively). Lazy-Leveling uses a leveling structure at the highest level. Its write amplification is also closely related to the data filling degree of that level, so there will be a large fluctuation in the performance gap with different T in Figure 12.

The performance of Lazy-Leveling is superior, and it does not depend on the reading workflow. This method of not relying on data has its advantages and disadvantages. On the one hand, it is still applicable to the workflow of strictly separating write and read. On the other hand, the distribution information of data cannot optimize the write performance without affecting the query experience. Under the premise of

performance, the write amplification of the system is failed to reduce to a minimum.

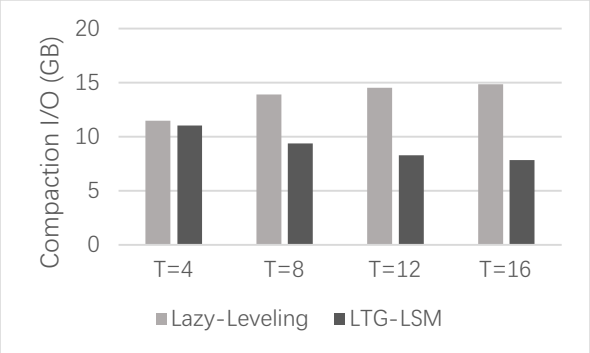

Fig 12. Compaction I/O compared with Lazy-Leveling

V. Related Work

According to the systematical arrangement of the research work, the improvement of the LSM-tree can be mainly divided into two categories: optimization of update and query. The research work on write performance improvement can be divided into three detailed categories: memory component optimization, such as S3 [10], FloDB [11], etc.; compaction optimization, such as dCompaction [12], PebblesDB [13], etc.; structure optimization, such as Dostoevsky [5], FlameDB [6, 7], etc. And the research work of improving read performance can be divided into three detailed categories: cache optimization, such as Kreon [14], Dual Grained Caches [15], etc.; filter optimization, such as Monkey [16, 17], ElasticBF [3], etc.; index structure optimization, such as LSM-trie [18], FASTER [19], etc.

VI. Conclusion

This paper designs and implements a heterogeneous LSM-tree structure based on reading hotness, which we called LTG-LSM. We achieved the structure on LevelDB and compared the performance with the original design. The experiment shows that the LTG-LSM structure can greatly reduce the write amplification of the database system based on the LSM-based structure and improve the write performance without almost sacrificing read performance.

Acknowledgement

This work was supported in part by National Natural Science Foundation of China (61572266) and Zhejiang NSF Grant (LZ20F020001).

References

[1] Davoudian, Ali, Liu Chen, and Mengchi Liu. A survey on NoSQL stores[J]. ACM Computing Surveys (CSUR), 2018, 51(2): 1-43.

[2] Luo, Chen and Michael J. Carey. LSM-based storage techniques: a survey[J]. The VLDB Journal, 2020, 29(1): 393-418.

[3] Li, Yongkun, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores[C]//2019 {USENIX} Annual Technical Conference. 2019: 739-752.

[4] O'Neil, Patrick, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[5] Dayan, Niv and Stratos Idreos. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging[C]//Proceedings of the 2018 International Conference on Management of Data. 2018: 505-520.

[6] Zhang, Weitao, Yinlong Xu, Yongkun Li, Yueming Zhang, and Dinglong Li. FlameDB: A key-value store with grouped level structure and heterogeneous bloom filter[J]. IEEE Access, 2018, 6: 24962-24972.

[7] Zhang, Weitao, Yinlong Xu, Yongkun Li, and Dinglong Li. Improving write performance of LSMT-based key-value store[C]//2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2016: 553-560.

[8] Bronson, Nathan, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, and Harry Li. TAO: Facebook's Distributed Data Store for the Social Graph[C]//2013 {USENIX} Annual Technical Conference. 2013: 49-60.

[9] Cooper, Brian F, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears.Cooper, Brian F, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM symposium on Cloud computing. 2010: 143-154.

[10] Zhang, Jingtian, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. S3: a scalable in-memory skip-list index for key-value store[J]. Proceedings of the VLDB Endowment, 2019, 12(12): 2183-2194.

[11] Balmau, Oana, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. FloDB: Unlocking memory in persistent key-value stores[C]//Proceedings of the Twelfth European Conference on Computer Systems. 2017: 80-94.

[12] Pan, Feng-Feng, Yin-Liang Yue, and Jin Xiong. dcompaction: Speeding up compaction of the lsm-tree via delayed compaction[J]. Journal of Computer Science and Technology, 2017, 32(1): 41-54.

[13] Raju, Pandian, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 497-514.

[14] Papagiannis, Anastasios, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage[C]//Proceedings of the ACM Symposium on Cloud Computing. 2018: 490-502.

[15] Li, Xiang, Guangping Xu, Hao Fan, Hongli Lu, Bo Tang, Yanbing Xue, and Ziliang Zong. Improving Read Performance of LSM-Tree Based KV Stores via Dual Grained Caches[C]//2019 IEEE 21st International Conference on High Performance Computing and Communications. IEEE, 2019: 841-848.

[16] Dayan, Niv, Manos Athanassoulis, and Stratos Idreos. Optimal Bloom filters and adaptive merging for LSM-trees[J]. ACM Transactions on Database Systems, 2018, 43(4): 1-48.

[17] Dayan, Niv, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store[C]//Proceedings of the 2017 ACM International Conference on Management of Data. 2017: 79-94.

[18] Wu, Xingbo, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items[C]//2015 {USENIX} Annual Technical Conference. 2015: 71-82.

[19] Chandramouli, Badrish, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates[C]//Proceedings of the 2018 International Conference on Management of Data. 2018: 275-290.