2021 20th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)

Improve the Performance of LSM-Tree based on Key-Value via Multithreading

Yuan Gao[1,2,3,4] , Ping Xie[1,2,3,4,*], Wendi Hua[1,2,3,4], Meng Lv[1,2,3,4], Jiating Lu[5]
1. College of Computer of Qinghai Normal University, Xining 810016, P. R. China
2. The State Key Laboratory of Tibetan Intelligent Information Processing and Application, Xining 810016, P. R. China
3. Key Laboratory of Internet of Things of Qinghai Province, Xining 810016, P. R. China
4. Academy of Plateau Science and Sustainability, Xining 810016, P. R. China
5. School of Physical and Electrical Engineering，Qinghai Normal University，Xining 810016, P. R. China
*Corresponding author: xieping@qhnu.edu.cn

***Abstract*—In the era of big data, key-value storage systems based on Log-Structure Merge tree (LSM-tree) are widely used in numerous industries. LSM-Tree is divided into two parts, one part is in the memory and the other part is on the hard disk. When writing data, first write to the memory, when the Immutable Memtable in the memory is full of data, then write to the disk to output a Sorted String Table (SSTable) file. However, the use of writing data from memory to disk is single-threaded. When the Immutable Memtable writes data to the disk, it will block other threads. This article uses multiple Immutable Memtables and multi-threaded writing to solve the single-thread blocking problem. By running the benchmarks of LevelDB to analyze the experiment, the experimental results indicate a higher improvement.**

Keywords—Key-Value store, Multi-Threading, Write performance, Read performance , LSM-Tree, Immutable Memtable

I. INTRODUCTION

With the advent of big data, data is the most important asset of data centers and even enterprises. In the data society, data has the dual role of basic strategic resources and key production factors. As the core part and underlying base of the information system, the construction and use of storage systems are directly related to the storage, use, and mining of data, which is the core asset of the enterprise. Currently, the challenges faced by storage systems are as follows. First, the amount of data has exploded. With the continuous development of the mobile Internet, the scale of enterprise data has shown explosive growth. Research shows that in 2018, China's new data will be 7.6ZB; in 2025, China's new data will reach 48.6ZB, with an average annual growth rate of 30%. Second, the business is responsible for presenting dynamic changes. A load of modern service platforms is non-linear and dynamically changing, especially for Internet-based services, and sudden changes in service load may occur at any time. Taking Double Eleven in 2020 as an example, the real-time transaction volume of online shopping malls exceeded 372.3 billion yuan. The peak order creation is 583,000 transactions per second, which is 1457 times that of the first Double Eleven event in 2009.

Currently, key-value storage has become an important part of various data-intensive storage applications. Compared with traditional storage structures, non-traditional storage has many advantages, including scalability, ease of use, and higher throughput. Non-relational databases are broadly divided into four categories based on data size: key-value, wide-column, document, and graph. Typical applications of key-value storage include graph databases, task queues, stream processing engines, NoSQL storage, and distributed databases. Increasing demand is higher reading and writing performance. According to the current situation, workloads increasingly tend to be key-value storage. Therefore, it is necessary to optimize key-value storage.

The current mainstream key-value storage structure is LSM-Tree[1], which is a hard disk-based data structure. Compared with B-Tree, it can significantly reduce the overhead of the hard disk and provide a longer time for high-speed insertion. LSM-Tree divides the storage structure into two parts, one part in the memory and the other part in the hard disk, which can make good use of the high-speed performance of the memory and the large capacity of the disk. Using such a model can trade high-speed writing performance at the expense of small read performance. The data in memory is stored in the Memtable and immutable Memtable in order. In the disk, the structure for storing data is SSTable. The memory and data in SSTable are ordered and unique. The one exception is data in disk tier 0. Because all memory data needs to be received at a high speed, the data at level 0 is repeated. The data storage process of LSM-Tree is generally written to the Memtable first, then to the immutable Memtable when it is full, and then flush to level 0 of the disk when it is full. After that, after level 0 is full, it is compressed and merged to level 1, and so on.

Many articles improve performance by optimizing the data structure of LSM-Tree. Optimize the SkipList memory data structure and optimize the disk storage structure SSTable to improve writing and reading performance. This paper analyzes the LSM-Tree writing process and analyzes each module one by one to find the optimization points. Through analysis, it is found that when the immutable Memtable is flushing data to the disk, LSM-Tree uses a single thread for flushing. When the immutable Memtable is inputting data, it will block the Memtable from writing data to the Immutable Memtable, and by reading the paper SIKL[7], it is found that LSM-Tree will trigger the long tail delay in three places, of which the immutable Memtable flush to the disk is a very important trigger point. And get inspiration from the article Multiple Immutable Memtables, which increases efficiency by increasing the number of immutable Memtables and writing multiple immutable Memtables to disk at the same time through multithreading.

II. BACKGROUD

A. Concept of LSM-Tree

Fig 1 shows the basic structure of the original LSM-Tree. LSM-Tree is divided into two parts, one part in the memory and the other part in the hard disk. The part on the hard disk

2473-3636/21/$31.00 ©2021 IEEE
DOI 10.1109/DCABES52998.2021.00046

is divided into multiple parts, each part is called a level, from top to bottom is level 0 to level n. When writing data, first write it to C_0 of the memory. When C_0 is full, C_0 will flush the data to the disk, and C_0 will flush the data to c1. When the data of c1 is full, it will be compressed and merged with C_2, And so on, until the C_n layer. As the number of layers increases, the capacity of each layer will increase exponentially. Because the data is continuously compressed downwards, the new data is in the lower layer and the old data is in the higher layer.

Fig 2 shows the basic structure of LevelDB[2]. LevelDB optimizes the structure of LSM-Tree. The memory c0 is divided into two parts, one part is Memtable, and the other part is immutable Memtable. Created a new data structure named SSTable, each layer contains many SSTables. For some auxiliary functions, log files, manifest files, and current files are introduced. The following briefly introduces the function of each part.

Memtable: Memory data structure, SkipList implementation, new data will be written here first.

Log file: Before writing to Memtable, the log file will be written first, and the log will be written sequentially by append. Another function of logs is that they can be used to restore data after the machine is down.

Immutable Memtable: After reaching the upper limit of the capacity set by the Memtable file, the Memtable will be converted to an immutable Memtable. This is to prepare for the conversion to an SSTable file. The immutable Memtable is not allowed to be modified. After the immutable Memtable is written, a new Memtable will be generated.

SSTable file: Disk data storage file. From level 0 to level n of the disk, each layer contains multiple SSTable files. As the number of layers increases, the total amount of SSTable contained in each layer increases exponentially, and the data in the SSTable is ordered . Among them, the SSTable file in level 0 is generated by directly dumping the immutable Memtable of the memory, and the other level SSTable files are generated by merging the files of the upper layer and the files of this layer. The SSTable files are written and generated sequentially during the merging process. Later, it can only be deleted in subsequent merges without any modification operations.

Manifest file: The manifest file records the distribution of SSTable files at different levels, the maximum and minimum keys of a single SSTable file, and other meta-information required by LevelDB.

Current file: When LevelDB starts, the first task is to find the current manifest file, and there will be multiple manifest files, the current file will record the file name of the current manifest, making it easy to find.

After introducing each component of LevelDB, let's briefly introduce the reading process and writing process. Writing process: The written operation of LevelDB includes setting key-value and deleting key. It should be pointed out that these two cases are the same in the processing of LevelDB. The deleted operation is actually to insert a piece of data marked as deleted into LevelDB. The external write interfaces provided by LevelDB include put, delete and write. Among them, writing needs a WriteBatch as a parameter, and put and delete first encapsulate the current operation into a WriteBatch object and call the write interface. A WriteBatch is a collection of a batch of write operations, and its significance is to improve writing efficiency and provide atomicity of all writes in the batch. In the write function, a writer is first encapsulated with the current WriteBatch, which represents a complete writing request. The LevelDB lock ensures that only one writer can work at the same time. Other writers suspend and wait until the previous writer finishes executing and wakes up.

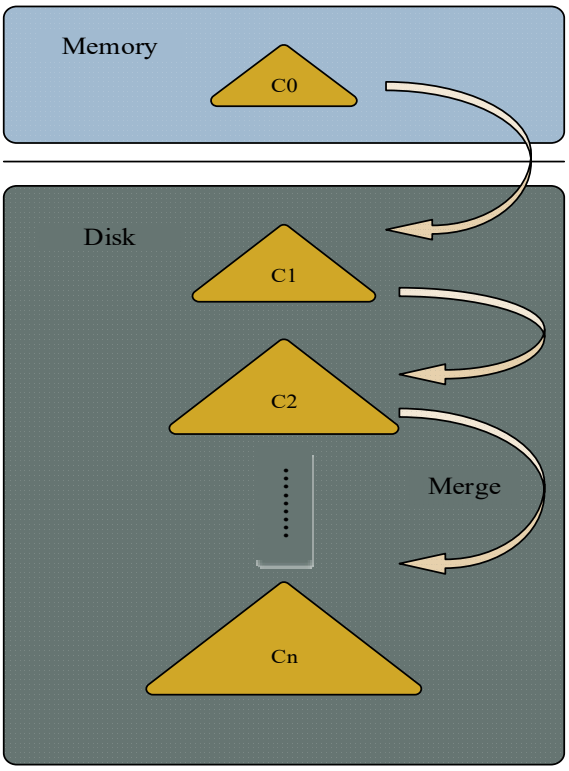

Figure 1: The original structure of LSM-Tree

The SSTable file on the disk is also very important. The keys in the SSTable file are ordered, and the key value in the file is the smallest. This is the case for each level of SSTable. However, the SSTable at level 0 is different from the SSTable at other levels and the key ranges of the two SSTable files at level 0 overlap. A certain file in SSTable belongs to a specific level, and the keys are in order, so it is very important to record the smallest key and the largest key. The manifest file is used to store this information. It records the management information of each file in SSTable, Such as which level it belongs to, what is the file name, and what is the minimum and maximum key.

Reading process: First, generate the key used for the internal query, which is generated by splicing the sequence with the UserKey requested by the user. The sequence can be provided by the user or use the current latest sequence, and LevelDB can ensure that only the writes before this sequence are queried. Using the generated key, try to read from the Memtable, immutable Memtable, and SSTable files in turn until it is found. Finding from the SSTable file needs to try to read in each layer in turn, because the key

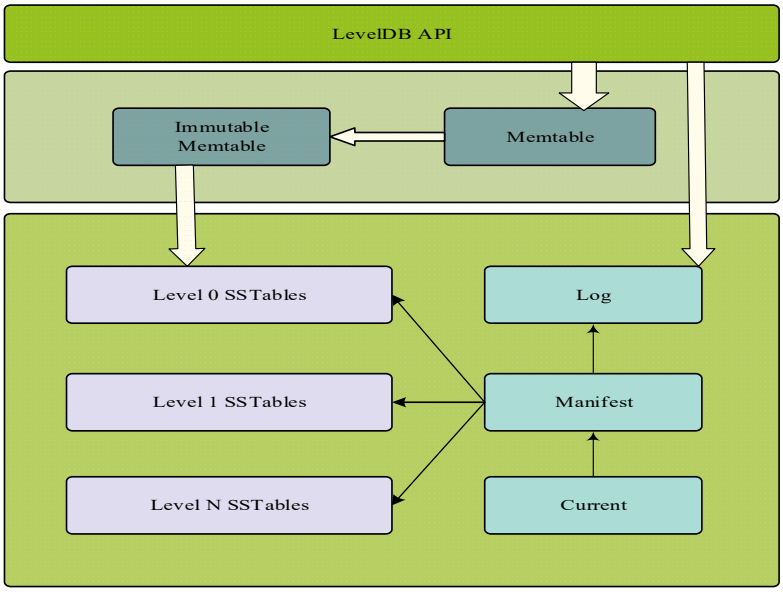

Figure 2:The structure of LevelDB

interval of each file recorded in the manifest can quickly know whether a key is in the file. Since level0 files are directly dumped from immutable Memtable, they will inevitably overlap with each other, so each file needs to be searched once. For other levels, because the merging process ensures that they do not overlap and are in order, the binary search method provides better quality efficiency.

B. Related Work

By studying the overall structure and read-write process of LSM-Tree, the current article is working on the following six aspects, such as: write amplification, merge operation, hardware, special workloads, Auto Turning, and Secondary Indexing. The write amplification problem is the root problem of LSM-Tree. There are many articles on optimizing this problem, such as: LWC-tree[8], PebblesDB[3], dCompaction[4], SifrDB[5], TRIAD[6], etc. Changing the compression method of each layer of SSTable on the disk can also improve the efficiency of LSM-Tree. There are the following research results: bLSM[10]. At present, the emergence of new storage media also provides a good opportunity for the application of LSM-Tree. The following articles are optimized under the new storage media: WiscKey[11], etc. In the new application environment, its special load has also changed. Like some extreme loads, one hundred thousand or one million visits will be generated within a few milliseconds. In response to these extreme loads, the following results have been produced: SlimDB[12], etc. The remaining two aspects are relatively novel, so we won't repeat them here.

III. METHODS

Here we introduce Multithreading. This idea comes from an article about Multiple Immutable Memtables[9]. The article mentioned that since there is only one immutable Memtable structure in memory when the immutable Memtable flushes data to the disk, it will block the thread that the Memtable writes to it. If multiple immutable Memtables are created, this kind of problem will not occur, and the blockage caused by a single module is well solved here. Through our research, we found that when the immutable Memtable is filled and then flushed to the disk, it is a thread that is responsible for it. This results in the fact that even if there are more immutable Memtables, it can only queue up for flush data. We will provide a new one here. The idea is to open multiple threads to concurrently flush data so that it will be written to the disk at a high speed.

Fig 3 shows the most primitive state of LevelDB. There is only one Memtable and immutable Memtable component in memory, and the immutable Memtable is single-threaded when flushing data to the disk. Although this method can ensure data consistency, it also sacrifices More performance, and it is not friendly to the current multi-core CPU multi-threading, cannot play its full performance.

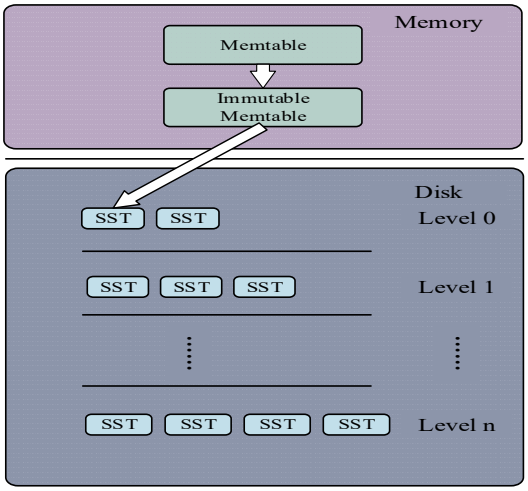

Figure 3:LevelDB Single Thread Mode

Fig 4 shows the optimization scheme of multiple immutable Memtable components and multithreading. It can be seen that multiple immutable Memtables can better optimize thread blocking, and multi-threaded flushing to disk can also optimize the long-tail delay problem.

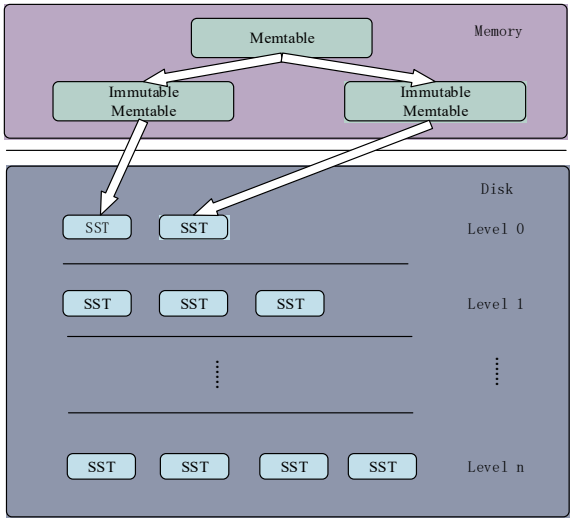

Figure 4:LevelDB Multithread mode

IV. EVALUATION

Experiment Setup. We conduct experiments on one machine running the Ubuntu 20.04 final with kernel 5.11.0. The machine includes two-sockets *Intel® Xeon®* Silver 4210(40 cores, 2.20GHz, 20MB L2 cache, 27.5MB L3 cache). The machine has 126GB of DRAM and nineteen 600GB disks. The file system used is ext4.

We tested fillseq, fillrandom, readseq, readreverse, overwrite these indicators respectively. The key size we use is 16 bytes, the value size is 100 bytes, and the workloads used are 1,000 and 10,000 respectively.

Results with benchmarks

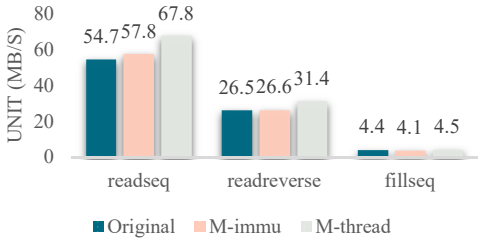

Figure 5:The comparative results of original, M-immu and M-thread under 1,000 Key/Value Workloads

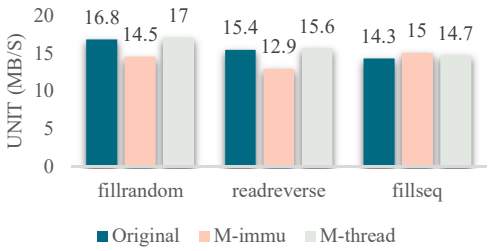

Figure 6:The comparative results of original, M-immu and M-thread under 10,000 Key/Value Workloads

Fig 5 shows the comparison of the three methods under 1000 key-value workloads. It clearly shows that the multi-threading method proposed in the article is better than the original method and the multi-immutable method. In readseq, readreverse, and fillseq, the improvement is 24%, 18%, and 2% respectively compared with the original method.

Fig 6 shows a comparison of the three methods under 10000 key-value workloads. It clearly shows that the multi-threading method proposed in the article is better than the original method and the multi-immutable method. In fillrandom, readreverse and fillseq, the improvement is 7%, 4%, and 3% respectively compared with the original method.

V. CONCLUSION

In this article, we introduced Multithreading, which is a new solution to improve reading and writing performance. It aims to increase the number of threads that interact with data from the memory to the disk to increase the rate of writing and reading data. And we have shown good results in the comprehensive workload experiment, which proves that this scheme is feasible.

ACKNOWLEDGMENT

This work is supported by The National Natural Science Foundation of China under Grant No.61762075.It is also supported by The Provincial Nature Science Foundation of Qinghai under Grant No.2020-ZJ-926. Ping Xie is the corresponding author of this paper.

REFERENCES

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

[2] LevelDB. 2005. A fast and lightweight key/value database library by Google. https://github.com/google/LevelDB.

[3] Raju, P., Kadekodi, R., Chidambaram, V., & Abraham, I. (2017, October). Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In Proceedings of the 26th Symposium on Operating Systems Principles (pp. 497-514).

[4] Pan, F., Yue, Y., & Xiong, J. (2017). dCompaction: Delayed compaction for the LSM-tree. International Journal of Parallel Programming, 45(6), 1310-1325.

[5] Mei, F., Cao, Q., Jiang, H., & Li, J. (2018, October). SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In Proceedings of the ACM Symposium on Cloud Computing (pp. 477-489)

[6] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., ... & Konka, P. (2017). {TRIAD}: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17) (pp. 363-375)

[7] Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., & Didona, D. (2019). {SILK}: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19) (pp. 753-766).

[8] Yao, T., Wan, J., Huang, P., He, X., Wu, F., & Xie, C. (2017). Building efficient key-value stores via a lightweight compaction tree. ACM Transactions on Storage (TOS), 13(4), 1-28.

[9] Gao, Y., Xie, P., Hua, W., Lv, M., & Li, P. (2021, August). Improve the Performance of LSM-Tree Based Key-Value via Multiple Immutable MemTables. In 2021 IEEE 12th International Conference on Software Engineering and Service Science (ICSESS) (pp. 223-227). IEEE.

[10] Sears, R., & Ramakrishnan, R. (2012, May). bLSM: a general purpose log structured merge tree. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (pp. 217-228).

[11] Lu, L., Pillai, T. S., Gopalakrishnan, H., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2017). Wisckey: Separating keys from values in ssd-conscious storage. ACM Transactions on Storage (TOS), 13(1), 1-28.

[12] Ren, K., Zheng, Q., Arulraj, J., & Gibson, G. (2017). SlimDB: A space-efficient key-value storage engine for semi-sorted data. Proceedings of the VLDB Endowment, 10(13), 2037-2048.