Optimization of Key - Value Storage System Based on Spatial Locality

Xiang Liu

University of Science and Technology of China, Hefei, Anhui, China

Keywords: KV store, Lsm-Tree, Separation of heat and cold, Prefix tree, Access statistics

Abstract: The advent of the era of big data has put forward new requirements for data management, and writer-intensive applications are appearing more and more frequently, which makes the LSM-Tree based key and value storage system play an increasingly important role in today's information society.In order to optimize the writing-intensive data structure, LSM-Tree takes full advantage of the fact that the sequential read/write performance of the disk is much higher than that of random read/write performance, and accumulates the data written to the disk in batches.However, this design does not take into account the characteristics of the data itself. In this paper, for the workload with spatial locality, the prefix heat statistics method is used to accurately identify the areas with high access frequency, and further the cold and hot separation method is used for targeted management, which effectively improves the system performance.At the same time, in order to avoid excessive memory consumption caused by statistical data heat, this paper adopts an adaptive method to prune and split the statistical tree, effectively reducing the cost of this part.

1. Introduction

With the rapid development of the Internet, data-intensive applications based on big data have begun to widely impact people's lives, and key-value storage is a core data storage model that is widely used in fields such as e-commerce and social networks. As application scenarios have diversified, LSM-Tree has become one of the mainstream storage engines for key-value storage systems.

LSM-Tree [1] is a data structure widely used in modern database systems. It organizes data into multiple levels to achieve efficient data insertion, querying, and deletion operations. Due to its ability to simultaneously meet high performance, scalability, and fault tolerance requirements, LSM-Tree has been widely used in large-scale distributed systems.

In traditional index structures such as B-trees [2] and B+ trees, each node stores a certain number of key-value pairs, which can achieve faster response times for operations such as reading, modifying, and deleting. However, when inserting new data, the entire tree needs to be frequently reorganized and balanced, which leads to lower insertion efficiency. LSM-Tree uses a log-like structure to write data in time order to different levels of disk files, thus avoiding frequent reorganization and balancing operations. Its batch writing feature also makes LSM-Tree more write-efficient than B+ trees. However, because of this design, LSM-Tree may have to traverse multiple data files to find the data when searching, and in the extreme case of searching for non-existent data, it may have to traverse all files in the tree, resulting in poor read performance. Therefore, improving the read performance of the system is particularly important for LSM-Tree.

The main contributions of this paper include three aspects:

We propose a method of separating hot and cold data to improve the performance of LSM-Tree for workloads in production scenarios.

We propose a method of accurately identifying hot and cold data based on prefix heat statistics and dynamically pruning and splitting the statistical tree to ensure that the statistical overhead is controllable.

We use the db_bench tool provided by RocksDB to verify the effectiveness of our proposed optimization method based on hot and cold separation.

  DOI: 10.25236/icmmct.2023.045

2. Related Work

Current optimization work on LSM-Tree mainly focuses on the following four aspects:

Work on read optimization: Russell Sears [3] proposed using Bloom filters to improve index performance by configuring a Bloom filter for each file and first checking whether the file contains the key through the Bloom filter when querying the key. Since Bloom filters occupy a small space and can be placed in memory, they effectively reduce the overhead caused by read misses. ElasticBF [4] developed this idea further and proposed a flexible Bloom filter mechanism tailored to the differences in access frequency of different regions in the data files of LSM-Tree.

Optimization for write amplification: In order to ensure that the system's query performance does not deteriorate excessively, the LSM tree must maintain the order of the data, and therefore, compaction operations need to be performed in the background, leading to duplicate writes of data. Maintaining the validity of the data is actually maintaining the order of the keys, and when key and value are repeatedly written during compaction, write amplification is an important factor. Lanyue Lu [5] noticed this and proposed WiscKey, whose main idea is to store the key and value in different locations, and the LSM tree actually stores the position information of the key and value. During compaction, only the position information of the key and value is processed, avoiding frequent movement of the value.

Optimization for new hardware: With the development of hardware, non-volatile memory (NVM) is gradually being used. Its features are byte-addressable, can be directly operated by CPU instructions, and data is not lost after power failure, and access latency and throughput are in the same order of magnitude as that of DRAM. This reduces the necessity of mechanisms such as WAL to ensure data security. Yao T et al. [6] used NVM hardware to store data in the L0 layer, reducing write pause and write amplification.

Optimization for compaction strategy: Sarkar S et al. [7] divided the compaction process into four components: when to trigger compaction, the physical layout of data, the granularity of data, and the data movement strategy. Based on this, they constructed the design space of compaction. N Dayan [8] analyzed the hierarchical organizational structure of LSM-Tree and modeled the cost of various operations. They proposed the Lazy Leveling strategy, which uses a leveled strategy near memory and a tiered strategy away from memory, achieving balanced system performance.

3. Working Principle of Lsm-Tree

The data in LSM-Tree is organized in layers, with the C0 layer in memory and the C1-Cn layers on disk. In the C0 layer in memory, there are two parts: Memtable and Immutable Memtable. Memtable is used to store incoming data and organizes the data in order by key, but LSM-Tree does not restrict the data structure used by Memtable (See Figure 1).

When the number of keys in Memtable or the space it occupies reaches a certain threshold, Memtable is converted into Immutable Memtable. Immutable Memtable is an intermediate state that is written to disk as an SSTable after Memtable, and once it is converted, it is no longer written to. When the number of Immutable Memtables in memory reaches a limit, they are flushed to disk and enter the C1 layer. This process is called a flush or minor compaction.

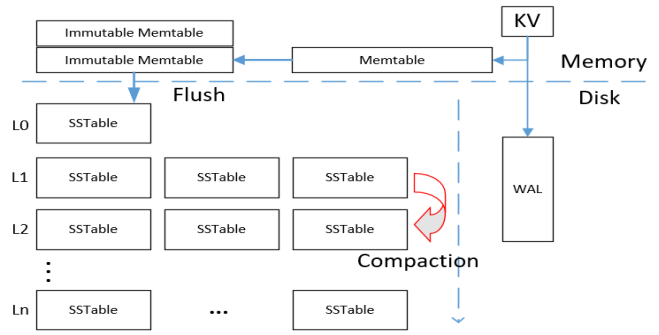

Fig.1 Lsm-Tree Architecture

On disk, LSM-Tree stores KV pairs in the format of Sorted Strings Table (SSTable). As data is written, more and more files are flushed into the C1 layer. When the number of files in the C1 layer reaches a certain threshold, the data in this layer is merged and written into the next layer using merge sorting. This operation is called compaction. When the number of SSTable files in each layer reaches a threshold, the same compaction is performed to maintain the order of the data in that layer.

When reading data, it first searches in the Memtable in memory. If not found, it searches in the Immutable Memtable. If still not found, it searches on disk layer by layer. First, it searches in the L0 layer closest to memory. The data files stored in the L0 layer overlap, so it may need to search all data files. If the data is not found in the L0 layer, it searches in the L1 layer. If the key is not found in the L1 layer, it continues to search in the next layer until the last layer is reached. If the key-value pair is still not found, then it does not exist in the system.

3.1 Compaction Strategies

Leveled Strategy: Except for the L0 layer, there is only one sorted run in each layer on the disk, meaning that there is no overlap in data range between files in each layer. This makes it possible to accurately locate a file that may contain data based on the data range of the file when searching for data, greatly improving the search performance. However, under this requirement, when the i-th layer triggers a compaction, one file F from the upper layer is selected, and all files in the i+1 layer that have an overlapping data range with F are merged and written to the i+1 layer. This results in a large amount of data being written during each compaction operation. As shown in Figure 2, when the data in the L1 layer is compacted into the next layer, it needs to be merged with the three expected data ranges in the L2 layer.

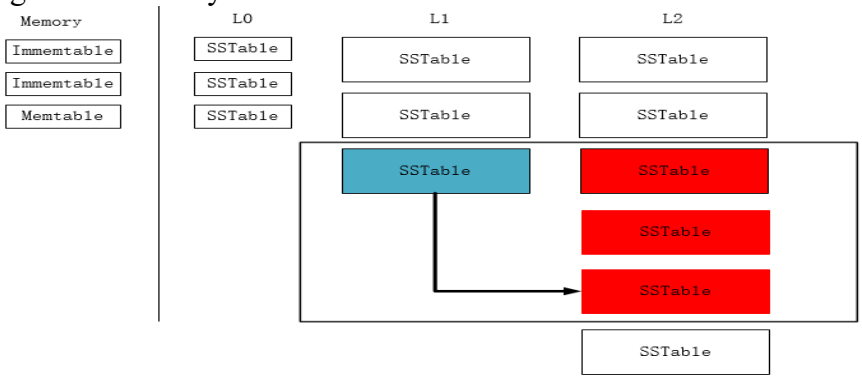

Fig.2 Leveled Strategy

Size-Tiered Strategy: There are multiple sorted runs in each layer on the disk, and the files in each layer do not need to ensure that there is no overlap in data range. This makes it possible to merge all files in the layer during compaction and put the merged file into the next layer for management, resulting in a lower write amplification. However, when reading data, it is necessary to search in multiple data files, resulting in lower read performance. As shown in Figure 3, when the data files in the L1 layer are compacted into the next layer, the three L1 SSTables are written to the L2 layer together.

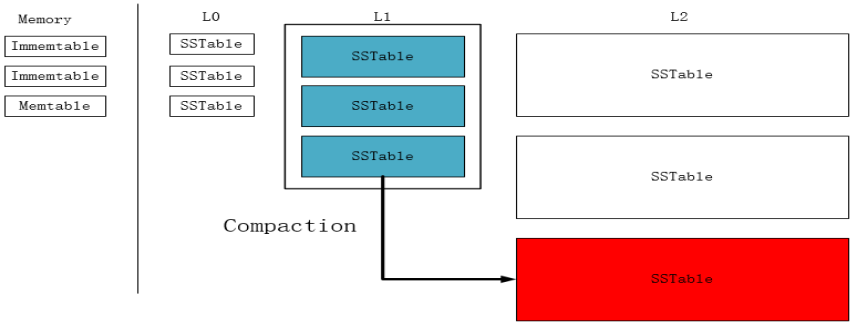

Fig.3 Size-Tiered Strategy

In summary, using the Leveled strategy is more conducive to improving read performance, and the Size-Tiered strategy is more conducive to improving write performance. The choice of strategy should be flexible according to the actual situation.

4. Optimization Based on Cold-Hot Data Separation

Cao Z et al. [9] summarized the workload encountered by Meta in actual production scenarios and found that their workload has strong spatial and temporal locality. That is, for a key that is accessed, the access frequency of keys near its key space is similar, and this key is likely to be accessed again soon. In addition, in this scenario, hot data only accounts for a small part of the overall data, but it occupies the vast majority of the system's access traffic.

To address this workload, the main idea of this paper is to use differentiated methods to manage cold and hot data separately according to the system's different requirements for them: use leveled strategy for frequently accessed data to ensure their read performance, and use size-tiered strategy for rarely accessed data to limit the system's write amplification phenomenon and improve the overall system performance.

Based on this technical route, this paper proposes DP-KV, whose system architecture is shown in the following figure. When the system receives a read/write request for a certain key, it is first judged by the cold/hot determination module, and then different partitions are selected for operation (See Figure 4).

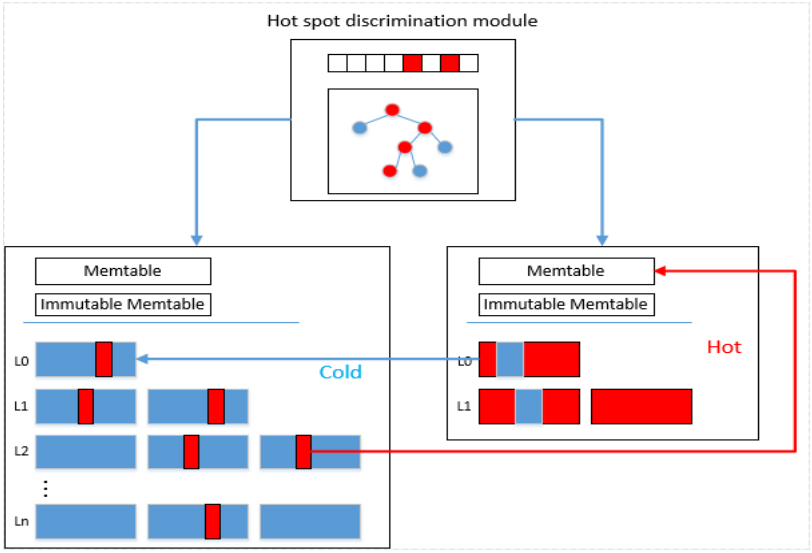

Fig.4 Dp-Kv Architecture

4.1 Prefix Hotness Statistics

This section introduces how to implement prefix frequency statistics. The main idea is based on hierarchical area partitioning, selecting regions with higher access frequency for further partitioning each time, until the hot spot area is accurately found.

The main structure of this module is a trie, where each node represents a character and maintains the access frequency from the root node to the string represented by the node. Whenever the system receives a request to access a certain key, it first updates the access frequency of each node on the trie corresponding to that key. If a node is determined to be a hot node, the key is classified as a hot data, otherwise it is considered a cold data.

However, this design may cause serious memory waste. To address this issue, it is noted that the proportion of hot data to the overall data is small, and it occupies a small key space. Therefore, only this part needs to be identified, and the other nodes in the trie actually have little effect on identifying hot data. Thus, the following strategy is used to limit the memory usage of the trie: a counter is used to count the access frequency of each key, and if the access frequency obtained from the counter is less than a set threshold, the key is considered cold data, otherwise it is further

processed by the prefix trie module, where each node in the trie records the access frequency information of different prefixes.

At the start of the system, there is only the root node in the trie. When the access frequency of a node is much higher than its sibling nodes, the node is considered to be on the hot data path, and it is split and updated with its child nodes. Otherwise, if the access frequency of a node is similar to its sibling nodes, it is considered not on the hot data path and can be merged with its sibling nodes. With this method, only the strings near the hot data are stored in the trie, and because cold data has a lower access frequency, the access frequency of nodes on the path to these strings is too low to trigger the splitting condition, which can effectively limit the memory usage of the trie.

A thread is started within the system every T seconds to update the hot spot detection module and detect whether the range of hot data has changed. If there is a change, the data migration process is triggered. The running process is as follows: first, the access frequency of the counter and each node in the prefix trie is attenuated. When traversing the nodes in the trie, if the access frequencies of the child nodes of a node are similar, it is considered not necessary to continue splitting and the child nodes are recycled. If the access density of the node is greater than a set threshold and the access frequencies of its child nodes are similar, the node is considered a hot prefix. That is, if the module is updated to that node, the key-value pair is considered within the hot spot range and should be further processed in the hot partition. The access density is the ratio of the access frequency of the node to the size of its subspace.

4.2 Data Migration

As the range of hot data changes, data that was once in the hot partition on the disk may start to have lower access frequency, while some data in the cold partition may become hot data. At this point, data migration should be carried out to adapt to the data lookup path. The following will explain from two aspects:

Data becoming cold: The migration direction is from the hot partition to the cold partition, and the migration timing is set to when the hot partition is being compacted for the following reasons:

(1) Migrating during compaction is beneficial in reducing the occupation of system resources. After reading and merging the data, the cold data can be written into the cold partition, which reduces the number of reads of the data.

(2) After the data is judged to be cold, its access frequency is still relatively high. Delaying migration is beneficial to adapt to the read path of this part of the data.

During compaction, the system determines the hotness of each key and creates two files for hot and cold data, respectively. Hot data is written into the hot file, and cold data is written into the cold file. After the compaction task is completed, the cold file is directly placed in the L0 layer of the cold partition without being written into the Memtable of the cold partition.

Data becoming hot: The migration direction is from the cold partition to the hot partition. Read the data in the hot range and write it into the memory of the hot partition. After the migration is completed, the corresponding nodes in the dictionary tree of the hotness judgment module are recycled. The reason for adopting a different plan than for cold data is that hot data should be placed in memory as much as possible to improve access efficiency.

During migration, the system sends a range query request to the cold partition. After the corresponding range of data is queried, it is written directly into the Memtable of the hot partition. Then, the cold partition executes a range delete command to mark the data that has become hot as deleted until the system background performs the compaction task to truly delete it.

It is worth mentioning that, due to the strict partitioning of data in the system before the change of the hot zone, only a small amount of metadata needs to be recorded when data migration is required. The data can be directly placed into the LSM-Tree of another partition without being rewritten into the Memtable. This avoids a large amount of CPU and IO overhead.

4.3 System Security

When the system crashes due to power failure or other reasons, it needs to be recovered and ensure that the written data can be read again. Most modern key-value storage systems use the

Write-Ahead Logging (WAL) mechanism to ensure data security, that is, the data is written to the log before it is written to the storage, and the write result is returned to the user after the write is successful. When the system restarts, data recovery can be done based on the log files. However, it is also necessary to ensure that the data can accurately find the corresponding data partition, so it is necessary to persist the hot spot information. Here, a combination of full and incremental persistence is used: the access data of the hot spot identification module is written to a file every certain period of time, and the changes in the data area are written to the log when each thread scans and updates the module. After the system crashes, the hot spot information can be reconstructed based on this file. The specific information written includes the prefix judged as a hot spot node and its access frequency.

5. Experimental Evaluation

In this paper, we implemented DP-KV based on Rocksdb, mainly including the identification of hot data and the migration of cold and hot data. The experimental environment is Ubuntu20.04, the version of Rocksdb is 6.7, the CPU is Intel Xeon E5-2650 v4 2.20GHZ, the memory is 32GB DDR4 2400 MHz, and the hard disk is Samsung SSD 860 500GB (See Figure 5).

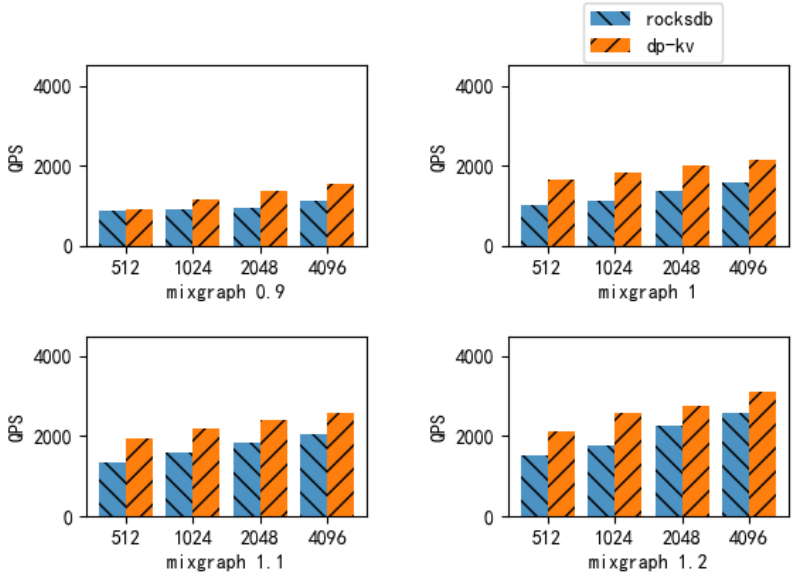

Fig.5 Dp-Kv Performance Test Results

Here, we use the mixgraph workload implemented in Rocksdb's db_bench tool for performance evaluation. This workload divides the key space into several ranges, and the access heat between these ranges conforms to a certain rule (default double exponential distribution). It models the size of key-value pairs and the number of requests per second to simulate various characteristics of space locality workload. The double exponential distribution refers to a distribution that follows the following form: $f(x) = a * \exp(b * x) + c * \exp(d * x)$. Here, for ease of displaying the results, let $a = 0$, $c = 0.5$. In Figure (5), mixgraph 0.9 refers to the data distribution parameter $d = 0.9$.

Here, we set the data size to 100GB, divide it into 100 regions, and change the cache size and the distribution of region access heat to examine the change in the system's QPS (queries per second, which refers to the number of queries the system can respond to per second). The experimental results are shown in the figure: the horizontal axis represents the cache size allocated to the system, and the vertical axis represents the system's QPS. It is easy to find that as the cache size and the data access skew increase, the system's QPS also increases. Moreover, in all scenarios, DP-KV achieved higher QPS results than Rocksdb, especially when the cache size is 4096MB and the distribution parameter is 1.2. In this case, DP-KV's QPS is 23.8% higher than Rocksdb, and when the cache size is 512MB, DP-KV's QPS is 28.2% higher than Rocksdb. When the distribution parameter is 0.9, DP-KV's QPS is improved by 5.7% to 27.7% compared to Rocksdb. This is mainly due to the

separation of hot data and the use of a more suitable reading strategy. When reading hot data, the system only needs to search the data file once to find the data in that layer, thereby reducing the number of I/O operations and improving the system's QPS.

6. Conclusion

This paper proposes a prefix-based hotness detection method and implements cold-hot separation for workloads with spatial locality. A targeted management scheme is applied to hot and cold data, focusing on read performance for hot data and write performance for cold data, effectively improving system QPS. Under this workload, the QPS can be increased by 5.8%-28.2%.

Future work includes two aspects: 1. Provide an interface for upper-layer applications to determine cold and hot data, using richer information to achieve more accurate judgment of cold and hot data. 2. In a distributed environment, data distribution across different nodes must be considered, and the factors affecting migration cost during data cold-hot changes are more diverse, and the migration process is more complex, requiring further study.

References

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

[2] Bayer R, Mccreight E . Organization and maintenance of large ordered indices. Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, 1970, 107–141.

[3] Sears R, Ramakrishnan R . bLSM: A general purpose log structured merge tree. Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012,217–228.

[4] Li Y, Tian C, Guo F, et al. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. Proceedings of the 2019 USENIX Conference on Usenix Annual Technical ConferenceJuly, 2019, 739–752.

[5] Lu L, Pillai T S, Gopalakrishnan H, et al. WiscKey: Separating Keys from Values in SSD-Conscious Storage[J]. ACM Transactions on Storage, 2017, 13(1):1-28.

[6] Yao T, Zhang Y, Wan J, et al. MatrixKV: reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM. Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference, 2020, 17–31.

[7] Sarkar S, Staratzis D, Zhu Z, et al. Constructing and analyzing the LSM compaction design space. Proceedings of the VLDB Endowment, Volume 14, Issue 11, pp. 2216–2229.

[8] Dayan N, Idreos S. 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.

[9] Cao Z, Dong S, Vemuri S, et al. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020, 209–224.