HotKey-LSM: A Hotness-Aware LSM-Tree for Big Data Storage

Yi Wang[1], Peiquan Jin[1,2*], Shouhong Wan[1,2]

[1] *School of Computer Science and Technology, University of Science and Technology of China*, Hefei, China

[2] *Key Laboratory of Electromagnetic Space Information, China Academy of Science*, Hefei, China

*jpq@ustc.edu.cn

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9377736

***Abstract*— In this paper, to improve the read performance of LSM-tree, we propose an enhanced LSM-tree called HotKey-LSM. The key idea of HotKey-LSM is to put hot keys and cold keys in two separated column families. Thus, when a hot key is not in the block cache, LSM-tree only needs to access a small hot-key LSM-tree to read the key. With this mechanism, most hot-key requests will be answered with low latency: if the hot key is in the block cache, we can return the memory address of the key; if it is not in the cache, we search the small hot-key LSM-tree. This differs from the traditional LSM-tree in that a hot-key request may traverse a large LSM-tree, which causes a high read latency. We implement HotKey-LSM on RocksDB and compare HotKey-LSM with the original RocksDB. The result in terms of QPS suggests the efficiency of our proposal.**

***Keywords*—Key-value store, LSM-tree, Read optimization, RocksDB**

I. INTRODUCTION

The Log-Structure Merge tree (LSM-tree)[1] has been a fundamental structure for big data storage. It has been implemented in many NoSQL database systems like BigTable [2] and LevelDB[3] at google ,RocksDB[4] and Cassandra[5] at Facebook, and HBase[6] at apache. LSM-tree maintains a multi-level data structure, and all data in each level are stored using *Sorted String Tables* (SSTables), in which all key-values are sorted in order. Data are flushed from memory to the SSTables in the first level through sequential writes. The top levels will be compacted to bottom levels using necessary compaction operations, which are also sequential writing operations. LSM-tree is a write-friendly storage structure because it can always transform random writes into sequential ones. However, such a feature also makes LSM-tree read-unfriendly, i.e., it has to traverse multiple levels to read data. Although a block cache is used in LSM-tree to maintain recently-read data, this cache will be polluted by the compaction operations because each compaction will make the data in the block cache invalid.

In this paper, to improve the read performance of LSM-tree, we propose an improved LSM-tree called *HotKey-LSM*. The high-level idea of HotKey-LSM is to put hot keys and cold keys in two separated column families. Thus, when a hot key is not in the block cache, LSM-tree only needs to access a small hot-key LSM-tree to read the key. With this mechanism, most hot-key requests will be answered with low latency: if the hot key is in the block cache, we can return the memory address of the key; if it is not in the cache, we search the small hot-key LSM-tree. This differs from the traditional LSM-tree in that a hot-key request may traverse a large LSM-tree, which causes a high read latency.

Briefly, we make the following contributions in this paper:

(1) We propose an improvement of LSM-tree called HotKey-LSM, which utilizes the column families to partition keys into a hot and cold group. By putting hot and cold keys in separated column families, we can reduce the reading cost on LSM-tree.

(2) To evaluate the effectiveness of HotKey-LSM, we conduct a series of YCSB experiments and compare HotKey-LSM with RocksDB. Under YCSB-Zipfian read-only workload, its data hotspots are constant, and its slope is 0.99. When the block cache ratio to the data set is less than 1%, the read performance of HotKey-LSM can reach 1.6x~2x more than that of RocksDB. When the block cache size gradually increases, its performance is equivalent to RocksDB.

II. BACKGROUND AND RELATED WORK

LSM-tree [1] has been implemented in many key-value stores [2-6]. In this paper, we mainly focus on RocksDB. The LSM-tree storage engine in RocksDB consists of two parts, Memtable and Immutable Memtables in memory and SSTables in the disk. There is a Memtable and one or more Immutable Memtables in memory. Memtable uses Skiplists as its structures. Skiplists are data structures that use probabilistic balancing. The algorithms for insertion and deletion are much simpler and significantly faster than equivalent algorithms for balanced trees. We can consider Memtable as an in-memory buffer for inserting, deleting, and updating. When Memtable reaches its threshold, it will transform into Immutable Memtable, which cannot be modified by any operations, and a new Memtable will be created for writing new key-value pairs. As we can know that memory is not an unlimited resource, so when a new Immutable Memtable appears, a background thread would be called and schedule flush to reduce memory usage. When the number of Immutable Memtables reaches the threshold, the background thread will flush all related Immutable Memtables (if the number of Immutable Memtables to be flushed is more than one, these Immutable Memtable will merge sort firstly) to disk. All these related key-value pairs are stored in one SSTable.

In previous work, some techniques were proposed to reduce read amplification by group more popular keys into higher LSM levels, decreasing the average read path. Splaying[7] copy frequently accessed keys to the top levels of the LSM-tree to make future reads cheaper. Tidal-tree[8] modifies the existing LSM-tree structure, considering the access frequency of each file. PrismDB[9] keeps the hot keys in the upper layer and flushes the cold keys to the lower layer, when performing compaction. However, these techniques consider IO read path and do not consider optimizing memory usage efficiency, which is the key factory for reading optimization.

Other techniques were proposed to improve the read performance by improving memory usage efficiency. AC-Key[10] manages three different caching components and adjust their sizes according to the workload. Monkey[11, 12] allocates memory to filters across different levels to

minimize the sum of their false-positive rates. ElasticBF[13] constructs many small filters for each SSTable and dynamically loads them into memory as needed based on access frequency. It realizes a fine-grained and elastic adjustment in running time with the same memory usage. These techniques show that reasonable use of memory can significantly improve read performance, but they have not tried to change the distribution of data in the disk.

III. Design of HotKey-LSM

HotKey-LSM utilizes an essential feature of the real LSM-tree implementation in RocksDB, namely the multi-column-family feature. We can partition all keys into multiple column-families, and each column family maintains its own LSM-tree structure. In HotKey-LSM, we use two column-families, namely a hot one and a cold one. Its reading count determines the hotness of a key. Thus, we can ensure that all keys in the hot column-family (Hot Key LSM) are read-intensive. HotKey-LSM is particularly efficient for read-intensive workloads, but it can also suit other kinds of workloads. For example, if the workload is write-intensive, we can infer that most keys will go to the cold LSM-tree (Cold Key LSM). In this situation, HotKey-LSM is similar to the traditional LSM-tree, in which all keys are stored in one LSM-tree.

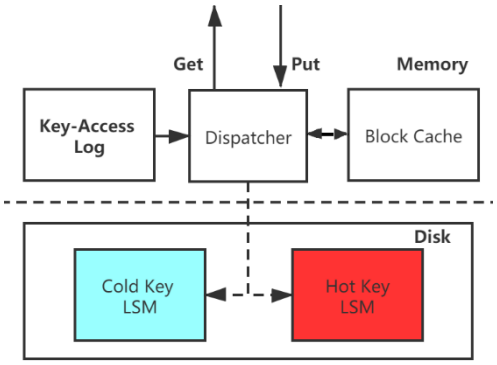

Fig .1. High-Level Architecture of HotKey-LSM.

Figure 1 shows the high-level architecture of HotKey-LSM. The *Key-Access Log* stores the access statistics of keys, which is used to identify the hotness of keys. If a key is identified as hot, the *Dispatcher* will send the key to the *Hot Key LSM*, which corresponds to the column family for hot keys; otherwise, the key will be directed to the Cold Key LSM, which maintains all cold keys. The *Dispatcher* will also be in charge of the interchange between *Cold Key LSM* and *Hot Key LSM*. If a hot key becomes cold or a cold key becomes hot, the *Dispatcher* will write the corresponding column family's key, and the next compaction operation will remove the original key. When searching for a key, HotKey-LSM first checks the block cache, then the Hot Key LSM, and finally the Cold Key LSM. For read-intensive skewed workloads, we can assume most queries will be evaluated in the block cache and the Hot Key LSM, yielding higher searching performance than the traditional single LSM-tree.

Algorithm 1 shows the detailed process of searching a key in HotKey-LSM. The searching process is straightforward. First, we check whether the search key is within the block cache. If yes, we return the cached key-value pair. Otherwise, we search for the hot key LSM-tree. If the key is not found in the hot key LSM-tree, we check the cold key LSM-tree. For skewed access patterns, we can assume that most requests focus on a couple of keys, which will become hot keys and be maintained in the hot key LSM-tree. Thus, we can infer that most search requests will be satisfied either in the block cache or in the hot key LSM-tree. As the hot key LSM-tree is much smaller than the cold key LSM-tree in skewed access patterns, we can evaluate a search request more efficiently than the original RocksDB.

Algorithm 1 *Get*

Input: userkey: the search key

Output: value: the result of query.

```
1: if(value=findInCache(userkey is true)
2:     return value;
3: type = dispatcher(userkey)
4: if(type is hot)
5:     value = findInHotkeyLSM(userkey)
6: else
7:     value = findInColdKeyLSM(userkey)
8: endif
9: return value
```

Algorithm 2 shows the process of inserting a key into HotKey-LSM. We first identify the type of the key to be inserted, which is performed by the *Dispatcher*. Then, we forward the hot key to the hot key LSM-tree and direct the cold key to the cold key LSM-tree.

Algorithm 2 *Put*

Input: <key, value>

Output: none.

```
1: type = dispatcher(key)
2: if(type is hot)
3:     PutInHotkeyLSM(<key,value>)
4: else
5:     PutInColdKeyLSM(<key,value>)
6: endif
7: return
```

One key issue in HotKey-LSM is how to deal with the change of hotness of keys, which means that some hot keys may become cold with time, or vice versa. It is not appropriate to move keys between the cold key LSM-tree and the hot key LSM-tree frequently because such movements require nearly reconstruction of both LSM-trees, which is time-consuming. We do not move keys between the two LSM-trees during the normal process in our implementation but delay the key movement to compaction. More specifically, when a compaction operation is triggered, we will read the involved keys in the cold key LSM-trees and the hot key LSM-trees, merge and write hot keys into the hot key LSM-tree, and write cold keys into the cold key LSM-tree. If one key is in the cold key LSM-tree but now it becomes hot, it will be removed from the cold key LSM-tree and inserted into the hot key LSM-tree after compaction. Before a compaction operation is executed, one hot key may simultaneously exist in the cold key LSM-tree and in the hot key LSM-tree. However, as the *Dispatcher* will direct key requests to the hot key LSM-tree, such a situation will not affect the searching performance of HotKey-LSM.

In our design, HotKey-LSM's block cache is more efficient than RocksDB. Considering a data block in Hot-Key LSM, each key in the data block has a high access frequency. Assuming that the skewness of workload is 1%, there is only one hot key in every one hundred keys. In general, the size of data block is 4KB. When the length of Key-Value is 1KB, the average memory usage efficiency is about 25%. When the length of Key-Value is further reduced, the memory usage efficiency will be further reduced. Theoretically, HotKey-LSM is more friendly to the short key.

By default, the block caches of different LSMs are shared, which will lead to the problem of accessing the cold

data block in Cold Key LSM evict hot data block in hot Hot Key LSM. Therefore, in the case of separation of hot and cold data, the caching strategy should be redesigned. In general, block cache is managed by some LRU-like algorithms [7, 8], which will discard the least recently used items first. However, the number of cold keys is much larger than that of hot keys; therefore, the least recently used item may also be a hot key. If we replace a hot key data block instead of a cold key data block, it is obviously not wise.

The replace algorithm can keep more likely hot keys data block in memory. However, accessing cold keys just likes a scan operation, which will cause hot data blocks to be evicted out of memory. The MRU(Most Recently Used) algorithm may have better performance. When a file is being repeatedly scanned, MRU will achieve high performance. In our implementation, we have separated hot keys from cold ones, so we can adopt a more effective cache strategy.

IV. Performance Evaluation

We conduct experiments using workload from Yahoo! Cloud Serving Benchmark. We use a Zipfian [8] read-only workload to benchmark in YCSB. We implement HotKey-LSM on RocksDB and compare HotKey-LSM with the original RocksDB. We use a workload consisting of 100 million keys, and the dataset size is 110 GB, resulting in an LSM-tree with three levels. Then, we issue one million operations to the LSM-tree and calculate the QPS (queries per second). In the experiment, we set the ratio of hot keys to 1% and vary the block cache's size to see the QPS change of HotKey-LSM and RocksDB.

In the experiment, we verified the impact of memory size and memory allocation strategy on performance. Firstly, the memory size is our main experimental parameter, it will directly affect the hit rate, thereby affecting the total number of I/Os and the overall performance. Secondly, we also compared the impact of different memory allocation strategies on performance. Two memory allocation strategies were used in the experiment: the sharing strategy and an exclusive strategy. The former does not make use of the existing data to separate information.

Figure 2 shows the QPS of HotKey-LSM and RocksDB under different block-cache sizes. HotKey-LSM outperforms RocksDB in all settings. The overall QPS improvement of HotKey-LSM to RocksDB, as shown in the right Y-axis of Fig. 2, is 1.2x on average and up to 2.05x, suggesting our proposal's efficiency. When we use separated block caches for the cold-key LSM-tree and the hot-key LSM-tree (80% of the total cache for the hot-key block cache, and 20% for the cold-key block cache), which corresponds to the "1024MB_mono" result in Fig. 2, HotKey-LSM yields much higher performance than using a shared block cache.

V. Conclusions and Future Work

This paper presents an improvement to LSM-tree called HotKey-LSM. HotKey-LSM sperate keys into two LSM according to the history access frequency of each key. HotKey-LSM targets read-intensive workloads and aims to improve the read performance of key-value stores. Experimental results show that the proposed HotKey-LSM can significantly improve read performance under a small block cache size compared to the RocksDB.

In the future, we plan to investigate efficient key-hotness calculating strategy as well as data migration strategy. Merely counting the number of reads and write operations of keys is not suitable for flash-memory based SSDs [16], because of the asymmetric read/write speed of flash memory. It is also worth studying how to move data between the cold and hot LSM-trees efficiently.

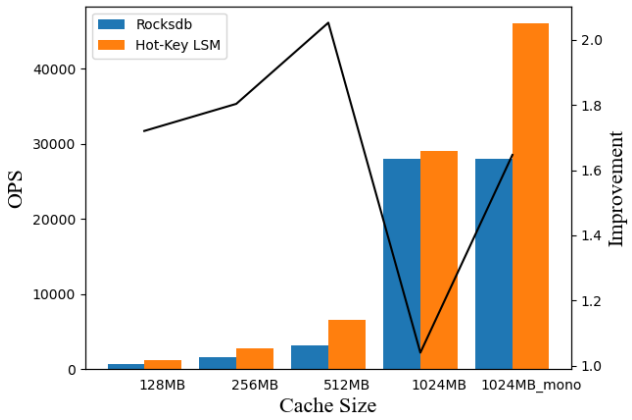

Fig. 2. QPS comparison between HotKey-LSM and RocksDB.

Acknowledgments

This work is partially supported by the National Science Foundation of China (No. 61672479 and No. 62072419).

References

[1] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, "The log-structured merge-tree (LSM-Tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, 2008

[3] LevelDB, http://code.google.com/p/leveldb.

[4] RocksDB, http://rocksdb.org/.

[5] Cassandra, http://cassandra.apache.org/.

[6] HBase, http://hbase.apache.org/.

[7] T. Lively, L. Schroeder, C. Mendizábal. "Splaying log-structured merge-trees". *SIGMOD*. 2018, pp. 1839-1841

[8] Y. Wang, S. Wu, R. Mao. "Towards read-intensive key-value stores with tidal structure based on LSM-tree". *ASP-DAC*, 2020, pp. 307-312

[9] A. Raina, A. Cidon, K. Jamieson, M. Freedman. "PrismDB: Read-aware log-structured merge trees for heterogeneous storage". *CoRR abs/2008.02352*, 2020

[10] F. Wu, M. Yang, B. Zhang, D. Du. "AC-key: Adaptive caching for LSM-based key-value stores". *ATC*, 2020. pp. 603-615

[11] N. Dayan, M. Athanassoulis, S. Idreos. "Optimal bloom filters and adaptive merging for LSM-trees". *ACM Transactions on Database Systems*, vol.43, no.4), pp. 16:1-16:48, 2018

[12] N. Dayan, M. Athanassoulis, S. Idreos. "Monkey: Optimal navigable key-value store". *SIGMOD*, pp. 79-94. 2017

[13] Y. Li, C. Tian, F. Guo, C. Li, Y. Xu. "ElastiCBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores". *ATC*, pp. 739-752, 2019.

[14] Z. Li, P. Jin, X. Su, K. Cui, L. Yue. "CCF-LRU: A new buffer replacement algorithm for flash memory". *IEEE Transactions on Consumer Electronics*. vol.55, no.3, pp. 1351-1359, 2009

[15] P. Jin, Y. Ou, T. Härder, Z. Li. "AD-LRU: An efficient buffer replacement algorithm for flash-based databases". *Data & Knowledge Engineering*. vol.72, pp. 83-102, 2012

[16] H. Zhao, P. Jin. P. Yang, L. Yue. "BPCLC: an efficient write buffer management scheme for flash-based solid state disks", *International Journal of Digital Content Technology and its Applications*, vol.4, no.6, pp.123-133, 2010