2019 IEEE 19th International Conference on Communication Technology

An Optimization of Key-Value Store Based on Segmented LSM-Tree

Kai Zhang, Yongsheng Xia, Yang Xia
School of Computer Science and Technology
China University of Mining and Technology
Xuzhou, China
e-mail: 15605205992@163.com, crossnote@163.com
yxia@cumt.edu.cn

Feng Ye
Beijing AutoNavi Yuntu Technology Co., Ltd.
Beijing, China
e-mail: 285429004@qq.com

***Abstract*—Storage Engine is the core of the storage system, R/W performance (read and write performance) of the storage system depends on the performance of the storage engine. sLSM-Tree structure (LSM-Tree structure based on the segmented index) is proposed, which is based on the structure of LevelDB. Segmented index structure is introduced to solve the collisions brought by adding hash storage RAM index structure to the index structure parts of LSM-Tree, i.e. trie index and hash index segmentally. By this way, index speed is improved and the pressure of updating index terms by compacting is reduced. The contrast experiment was conducted about the novel segmented index method presented in this paper. From the analysis of experimental results, sLSM-Tree has a significant performance in the RAM index and R/W operation on the hard disk compared with LevelDB which uses conventional LSM-Tree storage engine.**

Keywords-key-value; LSM-Tree; hash storage; levelDB

I. INTRODUCTION

Key-Value database is an important branch among NoSQL databases, and Key-Value Store is the core of Key-Value database. As the engine of a storage system, the performance of the storage engine directly determines the R/W speed of storage system [1]. There are three main storage engine currently such as hash storage engine, B-Tree storage engine and LSM-Tree (Log-Structured Merge Tree) [2] storage engine. The B-Tree storage engine is mostly used for the index of traditional relational database. Key-Value storage engine is based on the hash storage and LSM-Tree storage. Databases using hash storage engine includes Redis [3] and Memcached [4] database, while LevelDB [5] and HBase [6] database are using LSM-Tree storage engine.

As the most widely used Key-Value storage structure, the LSM-Tree is inspired from the log- structured merge-tree (LSM-Tree) from the literature. The presence of LSM-Tree storage engine is due to the serious disk IO problems in the B-Tree storage engine.

The principle is to store data as a certain data structure in RAM. When the amount of data in RAM gradually increases up to a certain threshold, the system will merge data in RAM onto the disk. When reading and querying the data, it queries in RAM firstly. If there is no target data in RAM, then it will query in disk. Meanwhile, there will be a large number of IO operations, which is the expense of read performance of LSM-Tree. In the view of the drawback of the LSM-Tree, researchers have already done a lot of work to improve the performance of LSM-Tree. bLSM-Tree [7] is a structure which combines the advantages of B-Tree and LSM-Tree. Firstly, bLSM-Tree which has near-optimal read and query performance differs from traditional LSM-Tree. Secondly, a 'spring and gear' compaction operation is provided. The operation ensures the stability of the merge process in each level of the tree, which avoids re-store operations in order to reduce the sacrifice of the traditional LSM-Tree's read performance. cLSM-Tree [8] is an optimization algorithm for extensible parallel storage. The algorithm adopts a multi-process friendly data structure and non-data block synchronous manner to maximize the utilization of multi-core CPU to improve the R/W performance. The study focuses on the background compaction process in the literature [9]. PCP (Pipelined Compaction Procedure) is proposed after analyzing the performance and bottlenecks of the compaction process, which makes full use of parallel performance of CPU and IO devices aimed to wider the bandwidth. PCP process is different from the conventional continuous compaction process. PCP utilizes the computing resources and IO resources fully in the system. It splits the original sequential scheduling into different hardware units to complete the operations independently, which is aimed to realize the parallel use of computing resources and IO resources.

Research based on LSM-Tree storage engine focus on the extension of the LSM-Tree structure to take advantage of LSM-Tree sequential write and avoid or optimize the short board of read strategy.

In a hash storage engine, hash table which consists of Key-Value and index item is usually stored in RAM and read and written by a hash map. Therefore, the average RAM space occupied by each Key is a very important performance indicator under the premise of limited RAM space. The RAM space occupied by each Key is smaller, which means more data can be indexed and stored under the condition of limited system RAM. Related research on hash storage structure mainly concentrate on the optimization of RAM index and efficient use of RAM space.

FlashStore [10] is a high-throughput Key-Value storage system. SSD is adopted as the stable cache between RAM and hard disk in the system to reduce RAM pressure. FlashStore maintains a log structure storing Key-Value in the SSD with the aspect of the realization of the specific data structure. Hash table is used to index in RAM. Only the identity of the Key is stored in hash table rather than the full

978-1-7281-0535-2/19/$31.00 ©2019 IEEE

key value to reduce RAM footprint and extra SSD operation. SkimpyStash [11] further reduces RAM footprint on the basis of FlashStore. SkimpyStash still indexes the Key-Value pairs stored in the SSD by hash table in RAM. However, the pointer of the Key is transferred from RAM to the SSD. It resolves hash table collisions using linear chaining and stores the linked lists in the SSD. Only the head pointer of the linked lists is stored in RAM to further reduce RAM footprint. SILT (Small Index Large Table) [12] is a memory-efficient, high-performance Key-Value storage system, which differs from SkimpyStash and FlashStore. SILT determines the index record using Hash Filter to reduce unnecessary SSD operations. SILT consists of three Key-Value storage system with different emphases in the storage structure, which are LogStore System focused on sequential R/W, memory-efficient HashStore System and SortedStore System which stores sorted dataset in SSD. Research on Key-Value based on SSD have also some breakthroughs domestically. A novel low latency Key-Value LLStore (Low Latency Store) System is proposed based on SkimpyStash and FlashStore in the literature [13]. The storage system adopts memory mapping technology to reduce IO requests in SSD, and in term of RAM index, introduces a new compaction strategy to largely improve the performance and reduce the query time at the expense of certain RAM space.

The focus of research on hash storage engine is to innovate and optimize the structure of RAM index in order to improve the R/W performance of the system. RAM index structure is the advantage of hash storage engine. However, there is not much mention about the optimization of hard disk storage in the research on hash storage engine.

II. sLSM-TREE SOLUTIONS

sLSM-Tree (Segmented-Index based LSM-Tree) is presented in this paper, which is combined with the optimization of the index RAM and hard disk storage. sLSM-Tree is a storage structure with hash storage engine characteristics. This structure not only retains the excellent characteristics of traditional LSM-Tree Sequential Write, but introduces RAM index method of hash storage structure to improve the index performance of system.

A. sLSM-Tree Structure

The design of sLSM-Tree structure divides into RAM structure and hard disk storage. Overall structure is shown in Fig. 1. The RAM structure consists of five index tables, while hard disk storage section is composed of an 8*8 storage matrix. The hard disk storage structure is split into 8 storage levels from Level 0 to Level 7. sLSM-Tree focuses on the optimization of RAM index based on LSM-Tree level structure.

Write Buffer is used to store Key-Value pairs which are latest written into system as a RAM buffer table. When the size of Write Buffer reaches a certain threshold, the data will be compacted into SSD. The data in buffer table will be stored into the hard disk. Then a new write buffer which stores new written data will be created.

sBlock address list is used to store the address of sBlock in the hard disk system. Each item in the address list consists of two variables, AIB (Array index bit) and sBlock ptr. sBlock is a persistent data file of the hard disk structure in the sLSM-Tree storage system, similar to the SSTable file of the LevelDB.

A sBlock address list is created and maintained in RAM to improve index speed. The 64 persistent data files are indexed by AIB using 1 byte of space. sBlock ptr is used to show the physical address of sBlock in the hard disk. An AIB is also included in the Key segmented index in RAM. Thus, we can quickly locate the Key Value pair in which data block is stored among 64 data blocks.

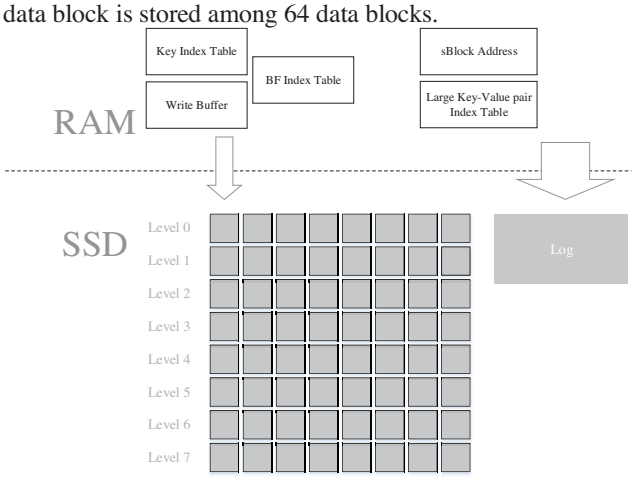

Figure 1. sLSM-Tree structure

Each item of large-sized Key-Value pair index table is composed of two variables. They are the Key value of the Key-Value pair and the physical address of the Key-Value pair stored in the hard disk respectively. As the statistics of the RocksDB is shown, the average length of Key is 24 Bytes, and the average length of Value is 100 Bytes [14]. While the Key length ranges 1B-2KB and the Value length ranges 1B-400KB in the DynamoDB [15]. Larger than 8 KB data items are individually indexed and stored in the large-sized Key-Value pairs index table because the single page size of the SSD in this paper is 8 KB. It is better to make data aligned.

Skiplist [16] is a data structure as a substitute for the balanced tree, which adopts the probabilistic approach rather than strictly mandatory adjustments to maintain the balance. Therefore, the insertion and deletion algorithms of the Skiplist are easier and faster than the equivalent balanced tree algorithm. As a simple and efficient data structure, Skiplist is adopted to index Key Index Table which is the core index table of sLSM-Tree storage system in this paper. Each item of the index table is composed of two variables. The one is the Key stored in the hard disk system and the other one is Segmented-index. Key Index Table is organized as Skiplist cluster in RAM. They are indexed according to the Key. Each Skiplist is used to index the Key-Value pairs of the sTable which makes up the sBlock. Each sTable is designed to include 21 SSD Blocks. The size of each SSD block is 1536 KB. Each Skiplist indexes at most 63*2^18<2^24 Key-Value pairs assuming that each Key-Value pair occupies 2 Bytes. So we design that the

probability factor is 1 / 2 and the MAXLEVEL is 24. That is, the upper limit of the index ability of a single Skiplist is set to 2 ^ 24.

B. Solution

sLSM-Tree introduces the RAM index structure with hash storage characteristics which solves the issues caused by the frequent changes of physical address in the compaction operation when writing the data. Hierarchical data storage is adopted in the part of persistent data storage referring to the storage structure of LevelDB. There are 8 levels from Level 0 to Level 7. Each level is composed of 8 sBlocks. Thus, the persistent data is arranged into a storage matrix of 8 * 8. The sBlock size is also different in the different levels.

AIB is in the Key suffix index item stored in RAM. AIB occupied one byte represents the position of sBlocks which stores the Key-Value pair of the Key in the storage matrix. The first 4 bits of the byte represent the level, while the left indicate the column of the sBlock. Other index items of the Key value suffix are used to represent the relative position of the Key-Value in the sBlock corresponding to the Key. Only need to modify the value of the AIB after each compaction operation. Other index values change little, because data is compacted as a block. As a result, the problem of the frequent modification of the hash RAM index large data brought by the compaction operation has been alleviated. Meanwhile, segmented index is introduced to the LSM-Tree storage engine.

Without introducing segmented index, if the target Key-Value pair is not in the hot data cached in RAM, we need to lookup in the hard disk. To lookup the level 0 firstly, we should search the level 1 if we fail finding, and so on. It wastes time and increases IO visits when searching. After the introduction of segmented index, we are able to directly locate the level and column of the sBlock which stores the Key-Value pair of the Key according to the value of the AIB in the suffix index item of the Key. Then, we can locate the exact storage position where the Key-Value pair stores in the hard disk using segmented index item. It not only simplifies lookup steps, but also reduces the IO visits when searching.

III. SEGMENTED INDEX OPTIMIZATION

A. RAM Structure

The relationship of RAM index table is shown in Fig. 2. RAM data structure consists of five major components. They are Write Buffer, sBlock address list, Key Index Table, BF Index Table and large- sized Key-Value index table. Write Buffer, sBlock address lists, and large-sized Key-Value pair index table have been introduced in detail in Section 2. While Key Index Table and BF Index Table will be described as the core optimization structure in this section.

BF Index Table (Bloom Filter index table) is an assistant index table, which is built in RAM to speed up queries in this paper. The table consists of the BF bit vector and the Skiplist RAM address. The BF bit vector built on the Skiplist in RAM is to determine whether the target Key is in the Key set indexed by Skiplist. The Skiplist RAM address is used to point to the head pointer of the corresponding Skiplist which is in the Skiplist cluster created by Key Index Table.

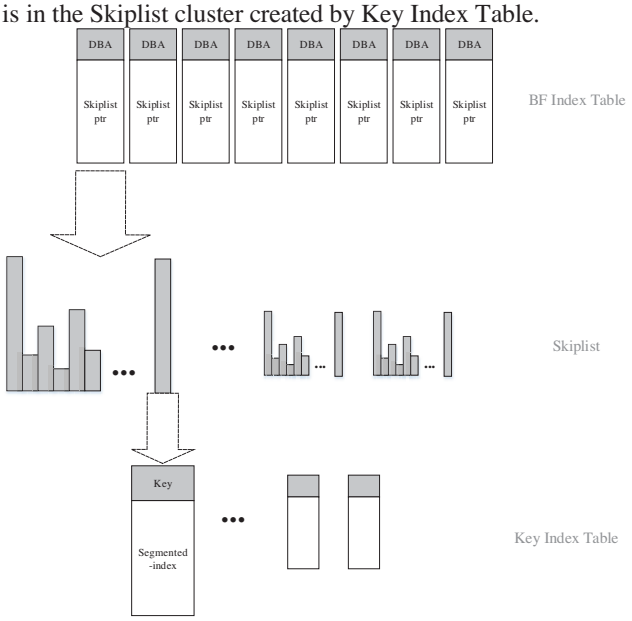

Figure 2. Relationship of RAM index

Segmented index optimization is the optimization method proposed in this paper. The core index structure in the Key Index Table is Segmented-index. Segmented index structure is divided into three parts, one is the AIB, another is the trie index structure, and the last is the hash index structure. As is shown in Fig. 3.

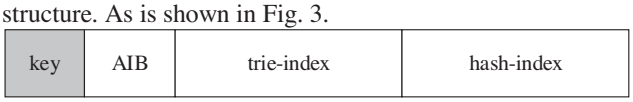

Figure 3. Segmented index structure

B. Array Index Bit (AIB)

Array Index Bit is used to index the position of the storage block of the Key value in the hard disk array. Hard disk storage structure is divided into an 8*8 storage array in the overall design. The AIB is added for each Key to represent the row and column in the storage array. The map between the AIB and storage array is shown in Fig. 4. The first 4 bits of the AIB indicate the row, namely the level of the sBlock stored in. While the left represents the column, namely which one is stored in the data level of the sBlock. For an example, 00100011 indicates the sBlock file in the third column of second row.

An important role of the AIB is to quickly locate the data block storing the Key-Value pairs in the hard disk. 64 sBlocks are indexed through a simple 8-bit binary byte. It can not only improve the index speed, but greatly save RAM space. The physical address of each sBlock can be found in the sBlock address list. Another significant benefit is reducing the frequent update of the index table caused by the compaction operation. The latest data written to the hard disk is firstly compressed and stored to Level 0. The whole data in Level 0 will be compacted to the Level 1 when the data

stored in the Level 0 reaches a set threshold. The physical address of the Key-Value pairs originally stored in the Level 0 changes due to the merge operation, so the index table in RAM should make the appropriate changes. We only need to modify the AIB in the segmented index structure after the introduction of the AIB.

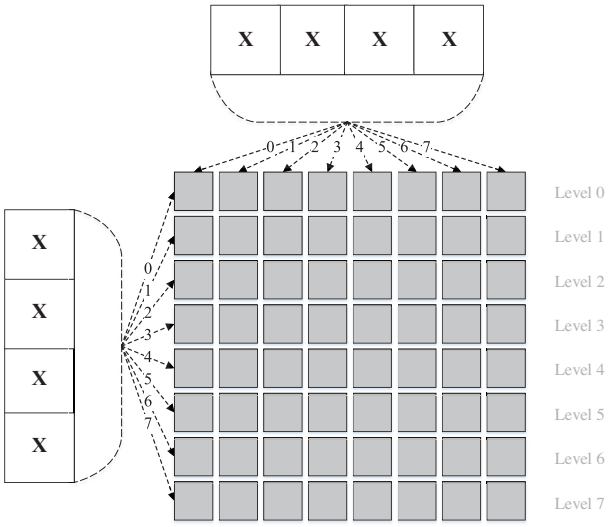

Figure 4. The map between AIB and storage array

C. Trie Index

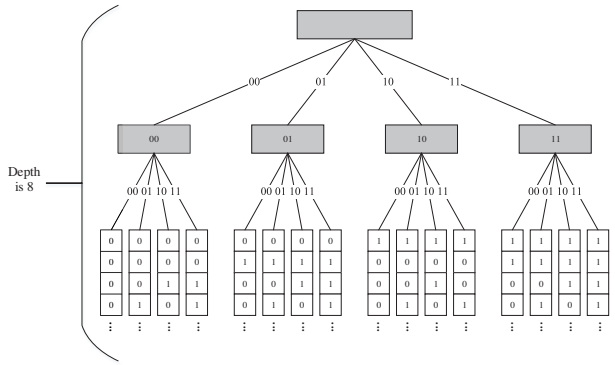

Figure 5. Structure of trie index

Trie-index structure is used to index the location of the sTable of the Key-Value pair in the sBlock. The default size of a sBlock of the Level 0 is set to 63 MB in the system. As each layer of the storage block is 4 times the upper layer, the storage ability of each sBlock in Level 7 will reach 1008 GB. Each 21 SSD Blocks are organized into a sTable in the hard disk. So each sBlock in Level 7 contains 2^15 sTable. As a result, the trie index structure of each Key occupies 2 bytes in RAM. The concrete structure of the trie index structure is shown in Fig. 5. The value of the root node of the trie is null the same as the traditional trie. So the depth of the trie is designed to be 8.

A reason why the index of the sTable of the Key-Value pair in the sBlock is designed as trie is that, the size of full RAM buffer could be an integer multiple of sTable. The relative position of the Key- Value pair stored in the sTable will not change, but the change is the location of the whole data block in the hard disk. Therefore, the way of segmentally identifying the change of the storage space using trie not only improves the index speed, but also reduces the update complexity of the change of index bit caused by compaction operation. Another reason is that the Append Only way is adopted by merge operation. All children nodes of a node of the trie have the same characteristic of the trie. So trie is the most appropriate index structure persistently stored in the hard disk by the way of Append Only.

D. Hash Map

Hash index structure is the last part of the segmented index structure, which is used to index the relative position of Key-Value pairs in the SSD Page. Each sTable is able to store at most 2 ^ 24 Key- Value pairs according to the system default settings. So the hash index structure builds 4 bytes of RAM space which is 32-bit binary hash value. sLSM-Tree structure index the relative position of the Key-Value pair in the sTable by hash mapping in the last part of the segment index in addition to add the RAM index structure with the reference to hash storage engine.

The hash algorithm adopted by sLSM-Tree is MurmurHash [17] algorithm proposed by the Austin Appleby in 2008. MurmurHash algorithm is a non-encrypted hash algorithm, which is generally used to hash index. MurmurHash algorithm has high performance and low collision rate compared to most hash algorithms. The random distribution characteristics is more obvious especially for the Key value with strong regularity. MurmurHash3 algorithm is the current version, which can generate the hash value of the 32 bit or 64 bit. Only the hash value of 32 bit need to index all Key-Value pairs stored in the same sTable in the hash index part of sLSM-Tree.

IV. R/W Operation

A. Write Operation

The writing process of sLSM-Tree is similar with LSM-Tree storage engine. New inserted Key- Value pairs are written to the Write Buffer in RAM. At this time, the Write Buffer is equivalent to the MemTable of the LevelDB. The Key-Value pairs are organized as Skiplist structure. The RAM data is compacted when the insertion of the next Key-Value into Write Buffer makes the size larger than the upper limit of a sTable capacity. The data in the Write Buffer is compressed into Level 0 in the hard disk. Compact operation will be continued when all sBlock in the Level 0 reach the storage upper limit of the layer. That is, the data in the Level 0 is merged into Level 1, and so on.

The system will update RAM index structure after the compaction operation. Key Index Table is updated firstly, which is built as Skiplist cluster in the RAM. Each Skiplist is used to index all Key-Value pairs stored in the sTable. The compact operation is performed when the data cached in the Write Buffer is full of a sTable. All Key in the Write Buffer are written into the Key Index Table. And Skiplist structure will be reconstructed. Each node of reconstructed Skiplist

consists of segmented structure and Key values. The part of the segmented index is used to represent the relative position of Key-Value pairs stored in the hard disk after the compaction and the location of sBlock stored in the 8*8 array.

BF Index Table is next updated. The RAM address of the Skiplist is used to store the address of the previous generated Skiplist in the RAM, while the Skiplist is used to index all Key-Value pairs in the latest sTable. The Dynamic Bloom Filter Array is designed to store all Key in the Skiplist. New RAM index need to be written into the log file for recovery after completing the update of the RAM index structure.

B. Read Operation

The main step of read operation is to lookup. Lookup starts from the Write Buffer in RAM for a given Key value. The Key-Value pairs in the Write Buffer are organized as Skiplist structure. So the lookup in the Write Buffer is the most direct and most efficient. If the Key value is not found in the Write Buffer, we need to continue searching using the RAM index structure.

Lookup begins from the BF Index Table in the RAM index structure. Each Bloom filter is to identify whether the Key stored in the corresponding Skiplist by BF Index Table. It jumps directly to the location of the Skiplist in the RAM according to the Skiplist ptr of the bloom filter and perform lookup operation when finding the first bloom filter possibly contain the Key value. It continues searching in the BF Index Table if there is no specified Key value in the Skiplist. Then it will go to the hard disk and lookup the Value of specified Key according to the segmented index structure in the Key Index Table if the specified Key value is found in the Skiplist.

The part of the segmented index first determines the location of sBlock of the Key-Value pair in the hard disk according to AIB. Then it will find the storage position of the sTable of the Key-Value pair in the sBlock by trie index. Finally, it will find the location of the Key-Value pair in the sTable through the hash index. The system can read directly the Value of the specified Key after finding the location of the Key-Value pair in the hard disk. If the specified Key is not found in all Skiplists, the lookup fails.

V. EXPERIMENT

The size of Key-Value metadata changes largely, ranging from a few bytes to 100KB or larger [18]. The aim of the YCSB framework of Yahoo ! is to simplify the comparison of the NoSQL database performance [19], which is the common testing tool for performance. R/W testing was performed on the LevelDB and sLSM-Tree through YCSB testing tool in this paper in order to further validate the optimization of the r/w performance of the sLSM-Tree.

At first, the write performance was tested by the load instruction of the YCSB. The test files were modified on the basis of the official test files of Yahoo ! in this paper. There are 10 test files of the insertion operation, indicating the test data ranging from 1,000,000 to 10,000,000 respectively. One of the first test file is shown in Fig. 6.

```
  GNU nano 2.2.6

# Yahoo! Cloud System Benchmark
# Workload for measuring leveldb with tsx
#
# load 1M records
# Then we perform 1M operations


recordcount=1000000
operationcount=1000000
workload=com.yahoo.ycsb.workloads.CoreWorkload

readallfields=true



requestdistribution=zipfian

threadcount=10
```

Figure 6. Test file of 1M records

There are 1,000,000, test records, while the operation number is also 1,000,000, indicated in the test file. 10 threads are used for testing in the test system. The results were written to the specified output file. The output file of the write operation presents the running time of the entire test and the average throughput of the system and other information.

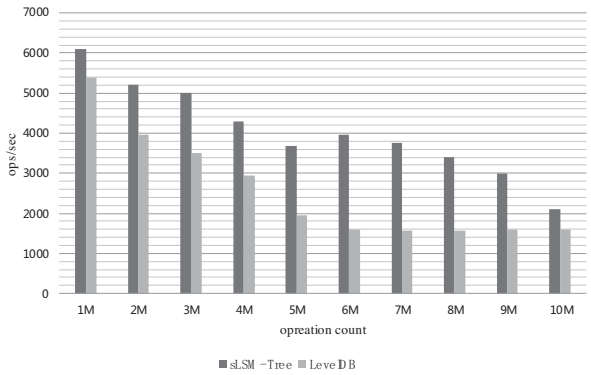

Figure 7. Write operation comparison

The throughput distribution of the LevelDB and sLSM-Tree under 10 test files is shown in Fig. 7. As shown in Fig. 7, the more the test data is, the slower the overall write speed of the two structures is. It is mainly because the compaction operation gradually increased with the increase of written data, thereby reducing the overall write throughout. The write throughout is obviously higher than the LevelDB in comparison with the write speed of two structures, because the write optimization of the LevelDB is mainly designed for the traditional mechanical hard disk. Although the speed is obviously raised when choosing the SSD as the persistent device, the sLSM-Tree structure which is optimized for SSD performs better.

The average delay of the operation is another important indicator of Key-Value Store r/w performance. The average delay of the Read operation and the Update operation is tested between the sLSM-Tree and the LevelDB through loading the testing data of different operation proportion. There are 10,000 pieces of testing data used in each test file.

9 testing files are divided depending on the proportion of the different operation. YCSB outputs the average delay of the operation in this period every 1000ms, and finally gives the total delay of the operation. The test results are shown in the Fig. 8. When the proportion of the Read/Update operation is small, i.e., the read operation is less than the write operation, the average delay of LevelDB is much higher than sLSM-Tree. It is mainly because the write operation of LevelDB is specifically optimized for HDD. Comparing with the sLSM-Tree optimized for SSD, the average delay of the sLSM-Tree is smaller in the case of more write operation. The average operation delay of two system trends to increase in general with declining of the proportion of the write operation. The advantage is write operation, because the LevelDB and the sLSM-Tree focus on the Key-Value Store with frequent write operation. So the overall performance of the system will decline when the proportion of the Read operation is heavy.

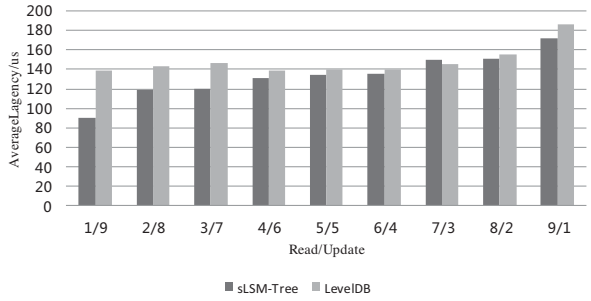

Figure 8. Average operating delay for different operations

VI. CONCLUSION

In this paper, we attempt to combine the advantages of Hash storage engine and LSM-Tree, add the RAM index structure with hash storage characteristics to the LSM-Tree and propose a novel storage structure (sLSM-Tree structure). The segmented index structure proposed in this paper is aimed to build RAM index structure which indexes the hierarchical storage in hard disk of LSM-Tree. The data stored in the different position of hard disk is segmented indexed. Not only it can quickly find the required data, but also the RAM index structure would not frequently update the physical address as the data in hard disk is compacted.

We made a comparative experiment on the optimization of the segmented index structure. The optimization proposed in this paper is benefit to improve the R/W performance of Key-Value Store according to the comparison of the experimental data and the analysis of the experimental results. But we have only studied Standalone storage engine, which is not involved in distributed research. Therefore, in the future, we can extend the advantages of stand-alone storage of sLSM-Tree to multi-machine storage and distributed systems to further improve the R/W performance of the entire storage system.

ACKNOWLEDGMENT

The authors would like to thank teachers and classmates for their help through obtaining and evaluating the results properly. This research was funded by National Natural Science Foundation of China (Grant No.51874300), the National Natural Science Foundation of China and Shanxi Provincial People's Government Jointly Funded Project of China for Coal Base and Low Carbon (Grant No. U1510115), the Qing Lan Project, the China Postdoctoral Science Foundation (Grant No.2013T60574).

REFERENCES

[1] Cattell R. Scalable SQL and NoSQL data stores[J]. Acm Sigmod Record, 2011, 39(4):12-27.

[2] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). Acta Inf., 33(4):351–385,June 1996.

[3] Redis. http://redis.io/.

[4] Memcached. http://memcached.org/.

[5] LevelDB:A fast and lightweight key value database library by google. https://github.com/google/leveldb, Sep.2014.

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

[7] Sears R, Ramakrishnan R. bLSM: a general purpose log structured merge tree[C]// Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012:217-228.

[8] Golan-Gueta G, Bortnikov E, Hillel E, et al. Scaling concurrent log-structured data stores[C]//Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015: 32.

[9] Zhang Z, Yue Y, He B, et al. Pipelined Compaction for the LSM-Tree[C]//Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, 2014: 777-786.

[10] Debnath B, Sengupta S, Li J. FlashStore: high throughput persistent key-value store[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1414-1425.

[11] Debnath B, Sengupta S, Li J. SkimpyStash: RAM space skimpy key-value store on flash-based storage[C]//Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 25-36.

[12] Lim H, Fan B, Andersen D G, et al. SILT: A memory-efficient, high-performance key-value store[C]//Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011: 1-13.

[13] Hu Hao.Reseach on accelerating technique for Key-Value data storage[D]. National University of Defense Technology,2012.

[14] Rocksdb. https://github.com/facebook/rocksdb/wiki.

[15] AWS Documentation. http://aws.amazon.com/documentation/.

[16] Pugh W. Skip lists: A probabilistic alternative to balanced trees[J]. Lecture Notes in Computer Science, 1989, 33(6):668-676.

[17] Appleby A. Murmurhash 2.0[J]. 2008.

[18] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload analysis of a large-scale key-value store[C]//ACM SIGMETRICS Performance Evaluation Review. ACM, 2012, 40(1): 53-64.

[19] Abubakar Y, Adeyi T S, Auta I G. Performance evaluation of nosql systems using ycsb in a resource austere environment[J]. Performance Evaluation, 2014, 7(8).