KT-Store: A Key-Order and Write-Order Hybrid Key-Value Store with High Write and Range-Query Performance

Haobo Wang[1,2], Yinliang Yue[1,2(✉)], Shuibing He[3], and Weiping Wang[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{wanghaobo,yueyinliang,wangweiping}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] School of Computer Science, Wuhan University, Wuhan, China
heshuibing@whu.edu.cn

Abstract. With the data volume increasing, key-value (KV) store plays an important role in today's storage systems due to its flexible architecture and good scalability. There are two types of data organization in current KV stores: *key-order* layout and *write-order* layout, which organize records according to key order and write sequence, respectively. While the former and the latter layouts deliver high throughput for range-query and write operations respectively, neither of them can perform well for both write and range-query operations. In this paper, we propose a hybrid KV store, KT-Store, which combines the key-order and write-order layout together to improve performance. More specifically, KT-Store stores keys and value metadata into a LSM-tree, and stores values into multiple tables called TrieTables. By inserting the value among multiple TrieTables in a key-order fashion leveraging a trie, and into a specific TrieTable in a write-order fashion, KT-Store can obtain the advantages of existing two layout types and avoid their shortcomings. We implement KT-Store in RocksDB 5.7.2. Extensive evaluations demonstrate that KT-Store can simultaneously obtain encouraging write and range-query performance: compared with key-order based RocksDB, the write performance is improved by $4.3\times - 12.6\times$ on HDDs; compared with write-order based Wisckey, KT-Store has $54.2\times - 112.6\times$ range-query performance on HDDs. Besides, KT-Store also has encouraging performance on SSDs.

1 Introduction

Key-value stores play a critical role in today's large-scale, high-performance, data-intensive applications in recent years. Compared with conventional SQL databases and other NoSQL data stores, key-value stores have stronger horizontal scalability, more flexible architectures, and more portable supports of different types of applications [1]. Due to their importance and benefits, KV

Published by Springer Nature Switzerland AG 2018
F. Zhang et al. (Eds.): NPC 2018, LNCS 11276, pp. 64–76, 2018.
https://doi.org/10.1007/978-3-030-05677-3_6

stores are widely used in distributed storage systems, such as BigTable, HBase and local storage systems, such as LevelDB and RocksDB.

Similar to traditional databases, KV stores need to support basic system workloads, such as data inserts, data updates, and range-queries. Data-intensive applications often run with massive data. These operations involve a large number of I/O read and write activities on hard disk drives (HDDs), which are the dominate media in current KV storage systems. As different workloads exhibit various data access characteristics, previous KV stores use different index structures to organize the key-value items, such that the system can provide desirable performance for different workloads.

There are two main types of data layouts in the data organization of current KV systems. The first type is the *key-order* layout that organizes the key-value items according to lexicographical order. Typical systems include conventional LSM-tree, its variants [2–4], B^+-tree [5], and its variants [6,7]. As all the key-value items are organized in key-order, such data layout can greatly improve range-query performance by utilizing the sequential read I/Os on HDDs. The second type is the *write-order* layout that organizes the key-value items based on the order of write sequence, which is inspired by the idea of Log-structured file system [8]. The typical systems applying this policy are Wisckey [9] and LSM-Trie [10]. By performing the append operations, random write I/O operations are translated into sequential ones on the HDDs, which means high I/O efficiency, thus such data layout can bring high throughput for write operations.

While the above approaches show decent performance for write and range-query workloads respectively. Unfortunately, to the best of our knowledge, none of them can perform well for write and range-query simultaneously. For example, while write-order layout can get high write throughput, it has inherent shortcoming of poor random read performance in the range-query operations; the key-order layout would cost lots of time to sort the key-value items like LSM-tree or to search the targeted storage location like B^+-tree, which results in limited write throughput.

To bridge this gap, we propose a hybrid KV store, which combines the key-order and write-order layout together to organize key-value items in the systems. KT-Store consists of three parts, one LSM-tree, one trie and multiple TrieTables. These components are used to store the keys and the metadata of values, to index the TrieTables according to a given key, and to store the values, respectively. In KT-Store, the values among multiple TrieTables are organized by the key-order while the values within each TrieTable are organized by the write-order. Thus, such hybrid structure can achieve high performance for both write and range-query operations.

KT-Store separates values from keys and stores them into different locations to eliminate the unnecessary compaction of values for enhanced write performance. While this design is inspired by the idea of Wisckey [9], it differs from Wisckey in that it stores the values into multiple tables while Wisckey [9] stores them into a single table. By leveraging a trie and organizing all the TrieTables in a key-order fashion, KT-Store can also achieve high throughput for range-query.

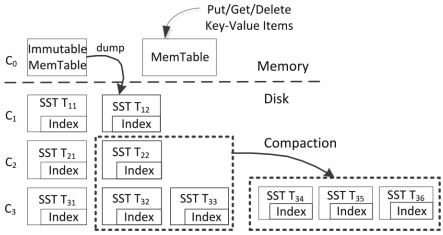


Fig. 1. LSM-tree data structure.

We implement KT-Store in RocksDB 5.7.2. Extensive evaluations demonstrate that KT-Store can simultaneously obtain encouraging write and range-query performance: it significantly outperforms RocksDB with $4.3\times - 12.6\times$ write performance and Wisckey with $54.2\times - 112.6\times$ range-query performance.

The proposed hybrid data layout scheme creates a better balance between write performance and range-query throughput. It can be applied in both HDDs and SSDs.

The following of this paper is organized as follows. Section 2 describes the related work. Section 3 describes the system design and implementation. Section 4 presents and discusses the evaluation results. Finally, we conclude this paper in Sect. 5.

2 Related Work

Key-order layout organizes key-value pairs ordered by the key. Log-Structured Merge-Tree (LSM-tree) is the typical structure, which was proposed by Patrick O'Neil et al. in 1996 [11]. LSM-tree is composed of multiple components, generally including one memory resident component and multiple disk resident components, as shown in Fig. 1. The key-value pairs in each component is sorted and arranged in lexicographical order. Each component size is limited to a predefined threshold, which grows exponentially. LSM-tree first uses an in-memory buffer, called MemTable, to hold the incoming KV items and keeps them sorted. When an MemTable exceeds its capacity threshold, it will be dumped into the hard disk as an immutable SSTable, such as T_{12}. Every disk component consists of multiple SSTables. Each SSTable contained the sorted KV items which have been sorted in the compaction procedure. During compaction procedure, KV items are merged and sorted. KV items of both C_i and C_{i+1} within the same key-range are firstly read into memory, then merged and sorted, and finally written back to C_{i+1} as the fix-sized SSTables. The compaction procedure is extremely I/O-intensive for repeated reads and writes, and dominates the disk I/Os of LSM-tree [2]. As a result, compactions that keep the key-value pairs in key-order layout bring write throughput decrease seriously. Several methods have been adopted to improve the write throughput key-order layout LSM-tree-based systems. First, fully utilizing the available hardware/software resources. PCP [12] makes use of

the parallelism of CPUs and I/O devices. LOCS [13] leverages the multiple channels of an SSD. Second, reducing the unnecessary data blocks moving. Skip-tree [2] skips some compaction procedures. PebblesDB [14], VT-tree [3] reduce the data rewrite. Third, accelerating the data flow. bLSM [15] and PE [16] partition the key range into multiple sub-key range and confine compactions in hot data key ranges. Others, like [17] applies LSM-tree to non-volatile memories, and GTSSL [18] uses layered mix of storage devices such as Flash SSDs and magnetic disks.

Write-order layout organizes key-value pairs ordered by the write sequence like LSM-trie [10] and Wisckey [9]. LSM-trie [10] stores data in a hierarchical structure by sacrificing the supporting of range-query operations. Wisckey organizes values in a value-log file, named vLog. The values in vLog is ordered by the write sequence. When a key-value pair is inserted, Wisckey first separates the key and the value, and append the value to the vLog. Then Wisckey inserts the key and the value metadata into LSM-tree. The values are appended to vLog as the insert procedure goes on. The write throughput keeps a high level because Wisckey spends no time to sort the values. As the value are arranged in vLog by write-order, Wisckey implements the range-query operations by parallel search the targeted key-value pairs. Under SSD environment, Wisckey could get comparable range-query performance as RocksDB [9]. Although SSDs are widely used nowadays, hard disks are still the main devices for conventional data stores. The range-query operations of Wisckey performs undesirable in hard disks for the reason that the search procedure is through random I/Os and can't utilize the sequential I/Os.

Key-order data layout can obtain attractive range-query performance but bad write performance, while write-order data layout can obtain attractive write performance but bad range-query performance. In RocksDB, we insert 100 GB data volume with value size as 4 KB by random order and by sequential order. The result shows that random insert performance only reaches about 30% of sequential insert performance. In Wisckey, the range-query performance with the values is randomly arranged only reaches about 22% of that with the value is sorted when the value size is 4 KB [9] with SSDs. There remains a need to well balance the write performance and range-query performance. Consequently, we are motivated to propose a hybrid KV store, KT-Store, to obtain attractive write and range-query performance simultaneously in one typical key-value store.

3 Design and Implementation

3.1 The Basic Idea of KT-Store

For effective balancing write and range-query throughput, we suggest KeyTrie-Store (KT-Store), which is designed as a replacement of RocksDB or Wisckey. The major distinction of KT-Store is that it uses a new key-and-write hybrid data layout to organize values. Figure 2 depicts the overall architecture of KT-Store. Each KT-Store instance consists of three parts, one LSM-tree, one trie, and multiple TrieTables. The basic concept of organizing keys and light-weighted

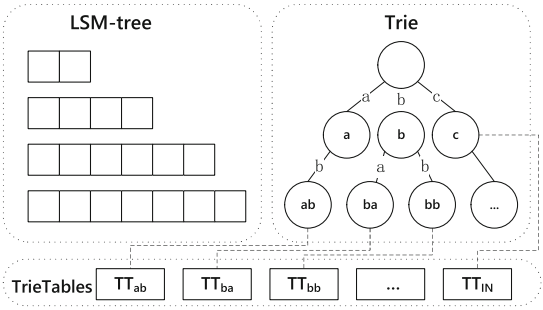


Fig. 2. The overall architecture of KT-Store (the strings in circles present the path from the root node to the child node or leaf node).

Table 1. The data structure of the trie node

Type	Property	Value
Internal node	ID	Denote the trie node and corresponding to one TrieTable
	Leaf flag	False
	Children mapping	Map to the child node with a mapping of <Character, Trie Node> (the character is the edge value in Fig. 2)
Leaf node	ID	Denote the trie node and corresponding to one TrieTable
	Leaf flag	True

value metadata into LSM-tree is similar to Wisckey. But different from Wisckey arranging values to one vLog, KT-Store leverages trie to split the input values into multiple TrieTables. Trie divides the whole key-range into multiple sub-ranges and sorts the sub-ranges in key-order. Every sub-range is correspond to a TrieTable, while trie keeps track of TrieTable locations in memory. TrieTables are in key-order among multiple TrieTables. Within a specific TrieTable, values are organized by write-order. Each leaf node of trie is correspond to a TrieTable, while all the internal nodes are correspond to one TrieTable TT_{IN}. In the write operation, all incoming values are appended at the end of the corresponding TrieTable to get the desirable write performance. In the range-query operation, the required values are stored in TrieTables under the key-and-write layout for the random I/Os can be avoided.

3.2 The Three Parts of KT-Store

The LSM-tree, which is originated from the conventional LSM-tree structure, is designed to store the key-metadata pairs. The value metadata includes its corresponding TrieTable ID, the offset in TrieTable, and the value size. We could locate the value based on the metadata.

One index structure of KT-Store, trie, is one of the most popular search trees, where keys are arranged in order. As shown in the Fig. 2, the root node

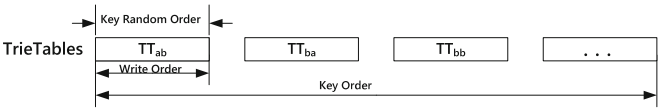


Fig. 3. The key-and-write hybrid order layout.

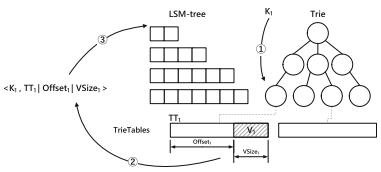


Fig. 4. The insert procedure of KT-Store.

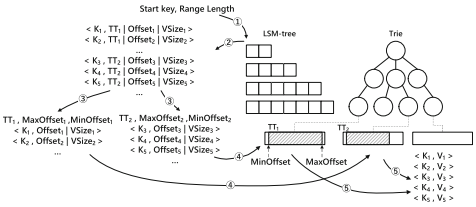


Fig. 5. The range-query procedure of KT-Store.

is associated with empty string and each edge presents one character. The path from the root node to the child node contributes to a key prefix or a key. Hence, the child nodes of one common node share the same key prefix. When one key-value pair is inserted, trie nodes are searched to match the key prefix and the matched node determines the TrieTable.

The two node formats of trie, also shown in Table 1, are: (1) internal node; and (2) leaf node. For example, the leaf node 'ab' in Fig. 2 has the properties of $<ab, true, TT_{ab}>$. If there exists one key-value pair with key as 'b', its value would be appended to TrieTable TT_{IN}, since all the internal nodes are corresponding to one TrieTable TT_{IN}.

TrieTables are used to store the values. The values are organized by key-and-write hybrid order as shown in Fig. 3. For example, TT_{ab}, TT_{ba} and TT_{bb} are in lexicographical order which denote the key prefix of 'ab', 'ba' and 'bb' respectively. In a specific TrieTable, the values are arranged by write-order which is in key random sequence.

3.3 The Main Procedures in KT-Store

Write Procedure. When a insert request of a key-value pair K_1V_1 arrives, KT-Store would separate K_1V_1 into the key and the value. Figure 4 presents the insert procedure. First, search the trie nodes to match the key prefix. A node and corresponding TrieTable are created when the node doesn't exist. Second, the value of K_1V_1 would be appended at the end of corresponding TrieTable. Third, original key and value metadata are bounded as $<key,\ TrieTable\ ID|offset|value\ size>$ to be stored into LMS-tree. For example, for a key-value pair $<abh, Value>$, we search trie to get the 'ab' node first. Next, the value 'Value' will be append to TrieTable TT_{ab}. We presumes that the start offset is 500. Then the $<abh,\ T_{ab}|500|6>$ would be inserted into LSM-tree

as one key-value pair. Since disk writes are performed sequentially for appending to the TrieTable, the write performance of KT-Store is much better than that of RocksDB.

Range-Query Procedure. A range-query operation first does the range-query operation in the LSM-tree to get the LSM-values in ② of Fig. 5. Then parse the LSM-values and group them by the TrieTable ID and compute the minimal offset and the maximal offset for each TrieTable in ③. Contiguously Read each TrieTable and get the data from the minimal offset to the maximal offset. This makes that every TrieTable would be read no more than once in one range-query operation. As known that disk I/Os cost a lot of time, KT-Store utilizes the sequential I/Os to decrease the range-query latency. Then, we pick the data that is read from TrieTables into key-value pairs according to their metadata. Last, aggregate the key-value pairs from TrieTables as output and return them in ⑤. For the TrieTable, we contiguously read the part rather than the whole TrieTable. Values are arranged on TrieTable by time sequence. Thus, in small length range-query operations or sequential insert, the targeted values only locate in part of one TrieTable.

Read Procedure. The read procedure begins with searching LSM-tree to find the key and the metadata, then it gets the value according to the metadata. KT-Store first searches the targeted TrieTable through the TrieTable ID, then it reads the targeted TrieTable to obtain the value. The beginning location is the offset address and the length is the value size. As this procedure is relatively straightforward and the page space is limited, here we ignore the detailed procedure.

3.4 Implementation

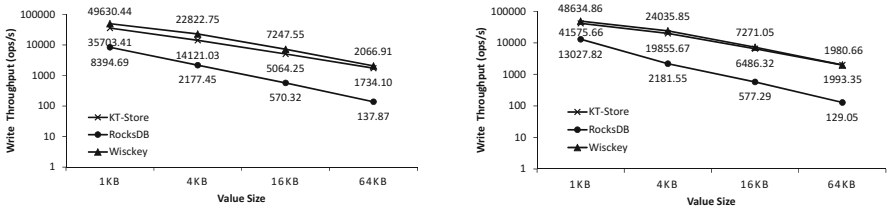
We build KT-Store with insert, update, read, delete and range-query interfaces. We integrate RocksDB 5.7.2 into KT-Store as the LSM-tree part. As RocksDB is written by C++, we develop KT-Store by C++. Except RocksDB, we implement all the other structures without utilizing other existing code. We implement the range-query API through the iterator in RocksDB referred to the RocksDB wiki in GitHub. We implement Wisckey also by integrate RocksDB 5.7.2 and develop it by C++ according its design.

3.5 Discussion

Reliability Mechanisms. When the system crashed, KT-Store needs to restore two types of data during the recovery procedure. One type is the key-value pairs that have been written into the TrieTables but have not been written into LSM-tree. The other is the key-value pairs that stored in MemTable of LSM-tree. For the former type, we utilize the write-ahead log to rerun the operations and ignore the data that have been appended to the TrieTable. For the latter

one, we utilize the existing LSM-tree reliability mechanism to recover the LSM-tree. Trie works as an index of the TrieTable in the insert procedure. As the trie changes infrequently, we persist the trie to the disk when the trie changes. When the storage server crashed, we recover trie from the disk.

Trie Scalability. In the current implementation, we only use fixed levels in trie part of KT-Store. We acknowledge that dynamic level numbers that varies with data volume would further improve the performance of KT-Store and adapt to the workloads with un-uniform key distribution. However, the focus of this study is to balance the write and range-query performance well. Thus, we believe the fixed levels do not hurt the conclusions and contributions of this study. We will develop adaptive policy in the future work.



(a) On HDDs (b) On SSDs

Fig. 6. The average write throughput of KT-Store, RocksDB and Wisckey for the value sizes vary from 1 KB to 64 KB for a 100 GB dataset.

4 Evaluation

4.1 Experiment Setup

We evaluate KT-Store and RocksDB, Wisckey on one Linux servers with hard disk devices and Solid State Drives. RocksDB is a persistent key-value store based on LSM-tree, started by Facebook. In every evaluation, we compare KT-Store with RocksDB and Wisckey. Except the compression type, which we set it as non-compression, the parameter values of RocksDB are applied to its defaulted settings as described following. The compaction style is level compaction which is the same as LSM-tree designed to be. The SSTable size is 64 MB and the ratio of C_{i+1} size to C_i size is 10. We use YCSB to generate workload traces, which are replayed in a light-weight workload generator. YCSB generates synthetic workloads with various degrees of read/write ratio, statistical distribution and value size. We configure YCSB to generate different datasets that are described in following subsections.

4.2 Write Performance

We load datasets with different value sizes and different scales into KT-Store, RocksDB and Wisckey to evaluate the write performance. The YCSB workload is set to 100% insert operations and insert key-value pairs randomly with uniform key distribution. We use the parameter of insert operations per second to evaluate the write performance.

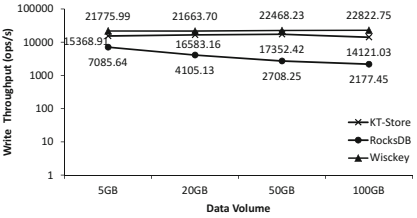


Fig. 7. The average write throughput of KT-Store, RocksDB and Wisckey for the data volumes vary from 5 GB to 100 GB for a 4 KB value size on HDDs.

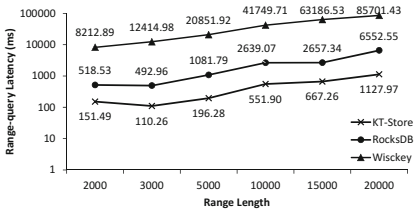
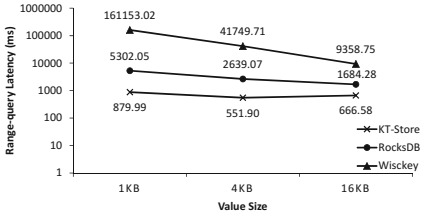
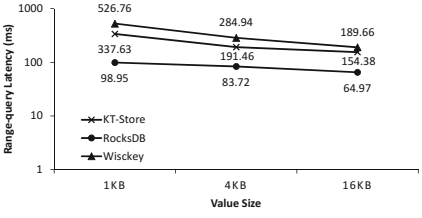


Fig. 8. The average range-query latencies of KT-Store, RocksDB and Wisckey for different range length with the value size as 4 KB on HDDs.



(a) On HDDs



(b) On SSDs

Fig. 9. The average range-query latencies of KT-Store, RocksDB and Wisckey for different value sizes with querying 40 MB data from a 100 GB database.

We conduct experiments on KT-Store, RocksDB and Wisckey with the value size grows from 1 KB to 4 KB, 16 KB and 64 KB, and the data volume is 100 GB. Figure 6(a) shows the write throughput of KT-Store is about 4.3 × −12.6× of that of RocksDB with HDDs. With the value size increasing, KT-Store obtain better write throughput than RocksDB. Moreover, KT-Store has comparable write throughput with Wisckey, which is about 16% decrease in best case. For SSDs, KT-Store outperforms RocksDB in all the cases and has almost the same performance with Wisckey in the best case as Fig. 6(b) depicts. To evaluate the write performance under different scales, we conduct experiments with the data volumes are 5 GB, 20 GB, 50 GB and 100 GB and the value size as 4 KB as

Fig. 7 shows. With the data volume increasing, KT-Store outperforms RocksDB further. KT-Store matches Wisckey for about only 23% decrease. Without sorting every key-value pairs, KT-Store obtain attractive write performance than RocksDB, and get comparable write performance than Wisckey.

4.3 Range-Query Performance

YCSB supplies the range-query interface required two parameters, the 'startkey' and the 'recordcount'. The former denotes the first key searched in range-query operation. The latter denotes the number of key-value pairs that this range-query operation requires, that is, the range length. We measure range-query performance for workloads with different range length of 2000, 3000, 5000, 15000 and 20000, and with different value sizes of 1 KB, 4 KB and 16 KB respectively. The data that already in the store is with uniform key distribution and inserted randomly. And the data volume is 100 GB.

Figure 8 presents the comparison of average range-query latencies in KT-Store, RocksDB and Wisckey with HDDs. It can be found that the range-query performance of KT-Store is 54.2 × −112.6× of that of Wisckey. And is 3.42 × −5.81× of that of RocksDB. Figure 9(a) and (b) depict the range-query latencies of KT-Store, RocksDB and Wisckey for querying 40 MB data from a 100 GB database on HDDs and on SSDs. KT-Store outperforms Wisckey in all our cases. The range-query operation in RocksDB is complemented by the iterator, and the index block and data block have iterators respectively. First, the data block of targeted key-value pair is determined by the index block iterator. Then according to the index which contains the offset information, RocksDB reads the targeted key-value pairs. KT-Store utilizes the sequential I/Os to read the parted TrieTable into the memory. Then KT-Store matches each key with its value and returns the range-query result.

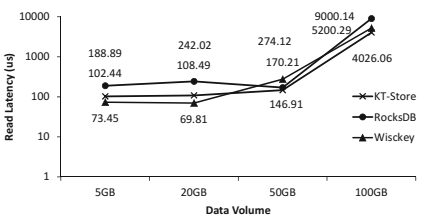


Fig. 10. The average read latencies of KT-Store, RocksDB and Wisckey for the different data volume with the value length of 4 KB.

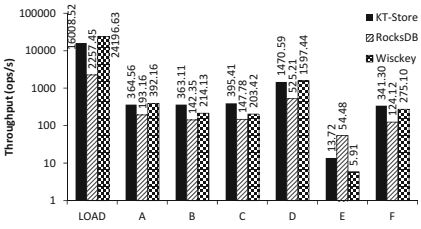


Fig. 11. The throughput of KT-Store, RocksDB and Wisckey in terms of the load and the six standard workloads with the key distribution is Zipf.

We attribute the difference of above evaluation to the following observations. First, RocksDB reads key-value pairs one by one with the key-order, but multiple versions of a same key can exist in different components in the same time.

Multiple component searches make that RocksDB can't fully utilize the sequential I/Os of hard disk and affects the range-query performance seriously. Second, KT-Store reads TrieTables one by one to fully utilize the sequential I/Os. The key-and-write hybrid order layout of TrieTables decreases the number of TrieTables that need to be read. As for Wisckey, the range-query operation relies on multiple read operations which results in massive random I/Os.

4.4 Read Performance

We conduct read operations on 5 GB, 20 GB, 50 GB and 100 GB YCSB datasets and evaluate the average read latencies on KT-Store, RocksDB and Wisckey on HDDs. We set the number of read operations as 1000 in each experiment and the value size as 4 KB. The dataset that already in the store is inserted randomly. Figure 10 shows the average read latencies of KT-Store, RocksDB and Wisckey. We can see that KT-Store shows a average read performance with RocksDB and Wisckey.

4.5 YCSB Standard Workload Evaluation

Our final set of experiments compares the performance with YCSB standard workloads, which can be treated as a basic benchmark for storage systems. Each of the six standard workload combines one or two operation types and can make us understanding the performance of the system. All the standard workloads are based on the Zipf distribution of key-value pairs.

Workload A is an update heavy workload. Workload B is a read mostly workload. Workload C is a read only workload. Workload D is a read latest workload which has 95% reads of the most recently inserted KV pairs. Workload E is a short ranges workload which does the short range-query operations. In Workload E, the max range-query length is 100 and the range-query length is under uniform distribution. Workload F is a read-modify-write workload. In Workload F, the key-value pairs will be read first, be modified next, and then be written back to the storage system.

We perform the six workloads on KT-Store, RocksDB and Wisckey with the 4 KB value size on HDDs. For each value size, we load 100 GB dataset with Zipf key distribution, then perform each workload and evaluate the throughput.

Figure 11 presents the operations throughput of KT-Store, RocksDB and Wisckey. In load stage, Workload A-D and F, KT-Store outperforms RocksDB by $1.8\times - 7.0\times$ throughput and obtains comparable performance with Wisckey. In Workload E, KT-Store performance is about $2.32\times$ of that of Wisckey. As have been discussed in Subsect. 3.5, each range-query almost read the whole TrieTable since the key-value pairs are organized by write-order in every TrieTable. In short range-query, KT-Store would only read one TrieTable in most case, while RocksDB maybe only read several blocks. Since the I/Os cost most time, reading the TrieTable in KT-Store would take much more time than reading several blocks in RocksDB. As a consequence, KT-Store performs a little worse than RocksDB in short range-query, but outperforms RocksDB in normal or long

range-query as described in Subsect. 4.3. Moreover, KT-Store gets attractive performance compared with Wisckey in Workload E.

5 Conclusion

In this paper, we propose KT-Store, based on a key-and-write hybrid order data layout. KT-Store well balance the write and range-query performance. Extensive evaluations demonstrate that KT-Store can simultaneously obtain encouraging write and range-query performance: compared with key-order based RocksDB, write performance is improved by $4.3 \times -12.6\times$; compared with write-order based Wisckey, KT-Store has $54.2 \times -112.6\times$ range-query performance. The YCSB standard workload evaluation shows that KT-store balances RocksDB and Wisckey well in various workload. In the future, we will dynamically extend the trie and limit each TrieTable in a threshold size, which would make KT-Store adapted to more workloads. We will also do some research to collect garbage.

Acknowledgments. This work was partially supported by Youth Innovation Promotion Association of Chinese Academy of Sciences No. 2016146, the National Science Foundation of China under Grant No. 61303056, No. 61572377, No. 61602467 and No. 6173396, and the Natural Science Foundation of Hubei Province of China under Grant No. 2017CFC889.

References

1. Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P., et al.: Finding a needle in haystack: Facebook's photo storage. In: OSDI, vol. 10, pp. 1–8 (2010)
2. Yue, Y., He, B., Li, Y., Wang, W.: Building an efficient put-intensive key-value store with skip-tree. IEEE Trans. Parallel Distrib. Syst. **28**(4), 961–973 (2017)
3. Shetty, P., Spillane, R.P., Malpani, R., Andrews, B., Seyster, J., Zadok, E.: Building workload-independent storage with VT-trees. In: FAST, pp. 17–30 (2013)
4. Yao, T., et al.: A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In: Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017) (2017)
5. Comer, D.: Ubiquitous B-tree. ACM Comput. Surv. (CSUR) **11**(2), 121–137 (1979)
6. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley DB. In: Conference on USENIX Technical Conference, p. 43 (1999)
7. Frühwirt, P., Huber, M., Mulazzani, M., Weippl, E.R.: InnoDB database forensics, pp. 1028–1036 (2010)
8. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. In: Thirteenth ACM Symposium on Operating Systems Principles, pp. 1–15 (1991)
9. Lu, L., Pillai, T.S., Gopalakrishnan, H., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: WiscKey: separating keys from values in SSD-conscious storage. ACM Trans. Storage (TOS) **13**(1), 5 (2017)
10. Wu, X., Xu, Y., Shao, Z., Jiang, S.: LSM-trie: an LSM-tree-based ultra-large key-value store for small data. In: 2015 Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference, pp. 71–82. USENIX Association (2015)

11. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Informatica **33**(4), 351–385 (1996)
12. Zhang, Z., et al.: Pipelined compaction for the LSM-tree. In: IEEE International Parallel and Distributed Processing Symposium, pp. 777–786 (2014)
13. Wang, P., et al.: An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In: Proceedings of the Ninth European Conference on Computer Systems, p. 16. ACM (2014)
14. Raju, P., Kadekodi, R., Chidambaram, V., Abraham, I.: PebblesDB: building key-value stores using fragmented log-structured merge trees. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 497–514. ACM (2017)
15. Sears, R., Ramakrishnan, R.: bLSM: a general purpose log structured merge tree. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 217–228. ACM (2012)
16. Jermaine, C., Omiecinski, E., Yee, W.G.: The partitioned exponential file for database storage management. VLDB J.- Int. J. Very Large Data Bases **16**(4), 417–437 (2007)
17. Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Redesigning LSMs for nonvolatile memory with NoveLSM. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 993–1005. USENIX Association (2018)
18. Spillane, R.P., Shetty, P.J., Zadok, E., Dixit, S., Archak, S.: An efficient multi-tier tablet server storage architecture. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 1. ACM (2011)