TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning

Kai Lu, Nannan Zhao, *Member, IEEE*, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng

Abstract—LSM-tree based key-value (KV) stores suffer severe read performance loss due to the leveled structure of the LSM-tree. Especially, when modern storage devices with high bandwidth and low latency are used, the read performance of KV store is seriously affected by inefficient file indexing. Besides, due to the deletion pattern of inserting tombstones, the KV stores based on LSM-tree are faced with the problem of read performance fluctuations that are caused by large-scale data deletion (also referred to as the Read-After-Delete problem). In this article, TridentKV is proposed to improve the read performance of KV stores. An adaptive learned index structure is first designed to speed up file indexing. Also, a space-efficient partition strategy is proposed to solve the Read-After-Delete problem. Besides, asynchronous reading design is adopted, and SPDK is supported for high concurrency and low latency. TridentKV is implemented on RocksDB and the evaluation results indicate that compared with RocksDB, the read performance of TridentKV is improved by 7× to 12× without loss of write performance and TridentKV provides stable read performance even if a large number of deletions or migrations occur. Instead of RocksDB, TridentKV is exploited to store metadata in Ceph, which improves the read performance of Ceph by 20%∼60%.

Index Terms—Key-value store, read optimization, learned index, SPDK, partitioned store

1 Introduction

Persistent key-value (KV) stores are an integral part of storage infrastructure in data centers and have been used in many applications including cloud storage [1], e-commerce [2], web indexing [3], [4], online games [5], social networks [6], [7], databases [8], [9], artificial intelligence (AI)/machine learning (ML) [10], [11], high-performance computing(HPC) [12] and so on.

Read-heavy workloads account for a large proportion of these applications. As for AI/ML applications, the ratio of read operations in model training is more than 90% [13]. Most of the database workloads perform more read than write. Especially, the read-write ratio of the typical OLTP applications reaches 5:1 or even 10:1 [14], [15], while that of the Tiwtter's Memcached workload reaches 7:1 [16]. Besides, most workloads at Yahoo! are read-heavy [17]. In addition, the Get operations account for the largest proportion of UDB and ZippyDB in RocksDB's workloads analysis [18]. These read-heavy workloads pose high requirements and challenges for the read performance of KV stores.

The state-of-the-art persistent KV stores such as BigTable [4], Cassandra [19], LevelDB [3], [20], and RocksDB [7] use a Log-Structured Merge-tree (LSM-tree) [21] as index structure to achieve excellent write performance and scalability. LSM-tree improves system write performance by converting small random writes into sequential writes effectively. Meanwhile, the files in the storage devices are sorted and stored in multiple levels, and they are merged from lower levels to higher levels. Due to the multi-level structure of the LSM-tree, the read operation needs to traverse each level of the structure, causing serious read amplification and read performance loss.

In recent years, many academic studies and industrial projects have tried to optimize the read performance and reduce read amplification of the LSM-tree mainly from the filter, cache, and index structure (Section 5). Among them, the RocksDB [7] from Facebook is one of the most mature and efficient KV storage in the industry, showing excellent performance and high availability. RocksDB is widely used as a storage engine in databases or distributed systems, such as MySQL [9], TiDB [8], Ceph [22], etc. RocksDB improves the read performance from many aspects, including using Bloom filter [23] to reduce extra I/Os, improving point-

- *Kai Lu and Changhong Fei are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {emperorlu, changhongfei}@hust.edu.cn.*
- *Nannan Zhao is with the School of Computer Science, Northwestern Polytechnical University, Xi'an 710129, China. E-mail: nannanzhao@nwpu.edu.cn.*
- *Jiguang Wan is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518000, China. E-mail: jgwan@hust.edu.cn.*
- *Wei Zhao and Tongliang Deng are with SenseTime Research, Shenzhen 518000, China. E-mail: nannanzhao@nwpu.edu.cn, dengtongliang@sensetime.com.*

Manuscript received 25 May 2021; revised 2 Oct. 2021; accepted 2 Oct. 2021. Date of publication 7 Oct. 2021; date of current version 22 Dec. 2021.
This work was supported in part by the Creative Reaserch Group Project of NSFC under Grant 61821003, in part by the National Natural Science Foundation of China under Grant 62072196, in part by the Shenzhen basic Research Project under Grant JCYJ20190809095001781, in part by the National Key Research and Development Program under Grant 2018YFB1004401, and in part by the Beijing Natural Science Foundation under Grant L192027.
(Corresponding authors: Nannan Zhao and Jiguang Wan.)
Recommended for acceptance by W. Yu.
Digital Object Identifier no. 10.1109/TPDS.2021.3118599

lookup through data block hash index [24], saving relevant information of the key ranges at continuous levels to reduce the number of searches, constructing prefix Bloom filters [25] to optimize the scans, supporting priority caching of the metadata to reduce random read I/Os and so on. In RocksDB, by setting and changing these optimization parameters, the read performance can be improved and read amplification can be reduced.

However, it is found that RocksDB does not perform satisfactorily in modern storage systems, where the storage devices have higher bandwidth and lower latency. Especially, as the scale of storage becomes larger, RocksDB often encounters large-scale data migration and failures. In the analysis of using RocsDB in modern storages and read-heavy environments, two observations on performance and availability are obtained, which can well illustrate the problems:

Observation 1. Inefficient file indexing seriously affects the read performance of storage devices, especially the high-speed storage devices. With the development of new storage media, the access performance of storage devices such as NVM and 3D XPoint SSD has improved by 2 to 4 orders of magnitude compared with the traditional HDD [26], [27]. The read latency of RocksDB on high-speed storage devices is analyzed. It is found that the indexing costs contribute almost equally to the latency of data accessing[28], and RocksDB's index search for files is not fast enough, causing read performance loss. Also, the synchronous I/O limits the concurrency of data access.

Observation 2. When undergoing large-scale deletions, the read performance degrade severely. As for the practical use of RocksDB, data migrations or failures often result in large-scale data deletion. In this case, the read performance of RocksDB is severely reduced (called Read-After-Delete), and sometimes access is unavailable or timed out. The analysis shows that this is caused by the mark deletion pattern of RocksDB.

To the end, TridentKV is proposed in this paper, which exploits three techniques to optimize the read performance and availability of RocksDB: *(1) Learned index block.* For file indexing optimization, TridentKV builds a learned index block to accelerate the read operation and effectively improves the efficiency of the learned index for string keys. *(2) Sub Partition.* For Read-After-Delete optimization, TridentKV designs a space-efficient partition strategy to effectively solve the problem of low read efficiency caused by large-scale deletions. *(3) Asynchronous reading design and SPDK.* For data access optimization, TridentKV employs SPDK to reduce the access latency of NVMe SSD and adopts an asynchronous design to improve concurrency.

The TridentKV is implemented based on RocksDB, and the evaluation results indicate that compared with RocksDB, the read performance of TridentKV is improved by 7× to 12×, and there is no performance fluctuation caused by large-scale deletions.

2 Background and Motivation

2.1 LSM-Tree and RocksDB

RocksDB is a popular KV store based on LSM-tree[21]. LSM-tree can improve write performance by converting small random writes into sequential writes effectively. As shown in Fig. 1, the LSM-tree first batches the writes in a fixed-size in-memory buffer called Memtable. When the Memtable is full,

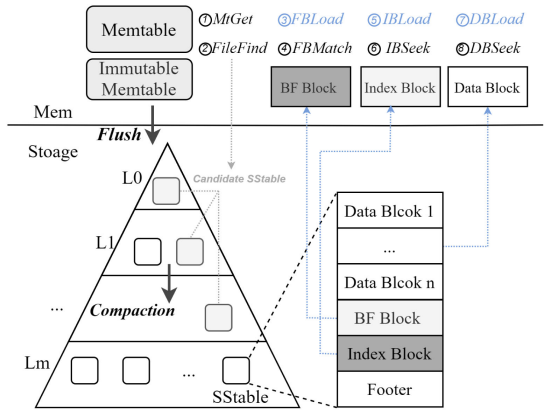


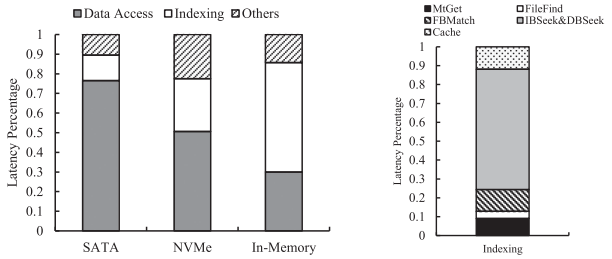
Fig. 1. The architecture of RocksDB.

it is converted to an Immutable Memtable, and the batched data are then flushed to the storage as SStables. The SStables in the storage devices are sorted and stored at multiple levels. Also, they are merged from lower levels to higher levels (e.g., L0, L1..., Ln level). The level size increases exponentially by an amplification factor (e.g., AF=10). The process of merging and cleaning SStables is called compaction. Compaction is conducted throughout the lifetime of an LSM-tree to clean the invalid or stale KV items and keep the data sorted on each level for efficient reads [21], [23]. As shown in Fig. 1, each SStable stores KV items in sorted order, and the KV items are divided into uniformly-sized blocks (called data blocks). Each SStable has an index block with one index entry per SStable block for binary search, a Bloom filter for quickly checking whether a KV pair exists in the SStable, and other metadata (Footer, etc.).

Read Process in RocksDB. As shown in Fig. 1, the process of looking up a key *K* in RocksDB can be roughly divided into eight steps. It should be noted that RocksDB supports multiple filters. The Full type is considered in this study, which configures each SStable with a Bloom filter:

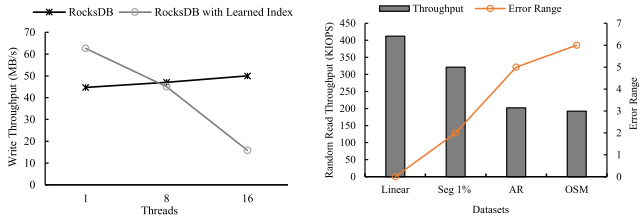
1) *MtGet*: the first step is to search the key in Memtable and Immutable in memory. 2) *FileFind*: if *K* is not found in 1), RocksDB will traverse the metadata of each layer in memory to determine the SStables that may have *K* (called candidate SStable). Since there are overlapping ranges in the L0 layer, each SStable in the L0 layer will be searched until the key is found. The other layers do not have ranges for the key, so RocksDB finds candidate SStabes through binary search. For each candidate SStable, the Bloom filter will be searched first. If the Bloom filter is not found in the cache, go to step 3. 3) *FBLoad*: Load the Bloom filter Block from the disk into the memory. 4) *FBMatch*: The filter is searched to check if *K* is presented in the SStable. Next, a search is performed on the index block. Similarly, if the index block is not found in the cache, go to step 5. 5) *IBLoad*: Load the index block from the disk into the memory. 6) *IBSeek*: Binary search is performed on the index block to determine the data block. Next, a search is performed on the data block. If the data block is not found in the cache, go to step 7. 7) *DBLoad:* Load the data block from the disk into the memory. 8) *DBSeek*: Binary search is performed on the data block to find *K*.

The steps for looking up the key can be divided into two types: in-memory *indexing* steps and *data-access* steps [7], [28]. The indexing steps include *MtGet*, *FileFind*, *FBMatch*,



(a) Indexing vs Data Access latency (b) Indexing latency

Fig. 2. Read latency percentage.



(a) with Training-WW (b) with Training-WR

Fig. 3. Performance with different way of building learned index in RocksDB.

IBSeek, and *DBSeek* to search the files and blocks to find the desired key. The data-access steps such as *FBLoad*, *IBLoad*, and *DBLoad* read the data from storage. The indexing steps include *IBSeek* and *DBSeek* to index files (called *file indexing*).

2.2 Read Performance Bottlenecks

Latency Measurement. The read latencies of RocksDB on different storage devices are analyzed by tools such as perf [29] and db_bench[7]. As shown in Fig. 2a, the first bar and second bar respectively represent the case where the data is stored on a flash-based SATA SSD and an NVMe SSD; the third bar represents the case when the dataset is cached in memory. It can be seen that the percentage of data-access latency decreases as the performance of storage devices improves, and it is even less than that of indexing with caching.

Also, it can be seen from Fig. 2b that the file indexing (*IBSeek* and *DBSeek*) account for more than 60% of the latency of indexing, indicating an obvious performance bottleneck. So, as the performance of storage devices increases, the overhead of indexing, especially file indexing, cannot be ignored. This leads to the *Observation 1* in Section 1.

2.3 A Related Work: Bourbon

Recently, learned index[30] is proposed to replace the traditional index structure to improve the read performance. Learned index is a model that maps from key to the position of a record. Learned index outputs the predicted value and error range and the position of the record can be found in binary search within the error range based on predicted value. Therefore, the accuracy of the learned index depends only on the distribution of keys. However, learned index is limited to the performance improvement of read-only scenarios, because the writing or modification of data will cause the accuracy of the model to decrease significantly, which needs to be retrained frequently. The SStable in LSM-tree is a read-only file, which is very suitable for the learned index.

So, Bourbon [28] discussed the possibility and superiority of using learned index in LSM-tree structure and proposed two ways for learned index, i.e., file learning and level learning. Since the level learning is completely unavailable [31], file learning is mainly used to accelerate file indexing in the KV stores. However, Bourbon does not perform well for read-heavy workloads, and the reading cost is very high for a wrong learned index prediction [31]. The model building process of Bourbon is: 1) Traverse the SStable to obtain the data set {k, pos}, and pos is the address of the block where the key is located; 2) Train the model through the data set; 3) After the model is trained, {pos, error} is obtained by the model to predict the key, and the error range is determined; 4) When the key does not exist in the predicted block, a search is performed on all blocks within the error range. When the model makes inaccurate predictions, lots of blocks need to be loaded into the memory, causing high costs.

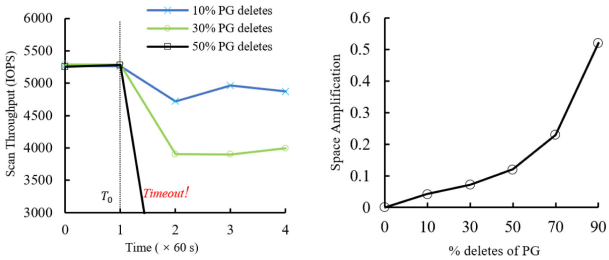
Google[31] adopts a new way to build the model in Bigtable. Specifically, the learned index is first trained through keys and the block number is predicted by the learned index. Then, the key is written to the predicted block location to construct an SStable. By writing the data to match the index, the predicted block must be correct, and an error prediction will only result in a difference in the size of the blocks. The specific process is : 1) The data set {k, S($\langle k, v\rangle$)} of each key in the file is obtained. Assuming that L($\langle k, v\rangle$) is the number of bytes in the record $\langle k, v\rangle$ and S(($\langle k, v\rangle$) denotes the number of bytes preceding k in the SStable A, we have

$$S(\langle k, v\rangle) = \sum_{\langle x,y\rangle \in A | x < k} L(\langle x, y\rangle).$$

2) The learned index is trained; 3) During the creation of SStable, a value S(k) is predicted by the learned index, and the block number can be obtained by $\lfloor S(k)/\alpha \rfloor$, where α is the block size. Then, the key is divided into the corresponding data block; 4) The location of each block is stored in an array and a predicted block number is mapped to a disk location. When a key is obtained, the block number is predicted by the learned index. Based on this, the corresponding block can be obtained, and it must be correct. However, the use of the learned index in the LSM-tree causes the loss of write performance, especially for the write-heavy workloads. The main reason for the performance loss is that the training process competes for the resources with the compaction process and flush process.

It can be seen from the above description that the learned index in Bourbon is built during the reading process (called Training-While-Reading, *Training-WR*), and that in Bigtable is built during the write process (called *Training-WW*). A conclusion can be made: Training-WR does not perform well for read-heavy workloads and Training-WW does not perform well for write-heavy workloads.

As shown in Fig. 3b, the random read performance of RocksDB with Training-WR under different workloads is evaluated, which can be found in Bourbon [28]. Specifically, "linear" means that keys are all consecutive, and there is a gap after a consecutive segment of 100 keys in the seg-1% dataset (i.e., every 1% causes a new segment). The other two data are from real-world datasets: Amazon reviews (AR)



(a) Performance fluctuations (b) Space amplification

Fig. 4. Performance fluctuations and space amplification caused by data migration in Ceph.

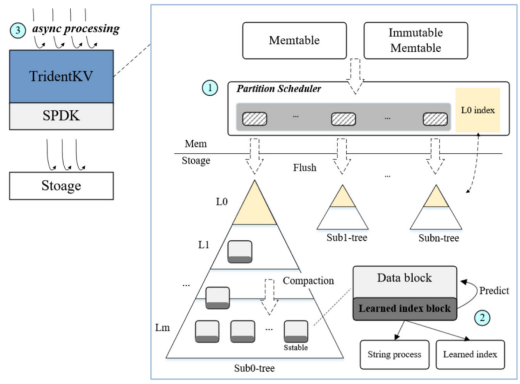


Fig. 5. The TridentKV architecture.

[32] and New York OpenStreetMaps (OSM) [33]. It can be seen that under different workloads, the random read performance drops severely as the error range increases. This indicates that the versatility of Training-WR is very poor, and it is difficult to use Training-WR in real systems. As shown in Fig. 3a, the performance of RocksDB using Training-WW to build learned index under different write requests is evaluated, where the write request is changed by setting the number of writer threads. It can be seen that when there are fewer write requests (e.g., a single write thread), the Training-WW can improve the write performance. But with the increase of write requests, the benefits of Training-WW are diminishing. Specifically, the write performance drops sharply when there are 16 writer threads.

Challenge 1. How to better use the learned index to optimize the file indexing of KV store based on LSM-tree? The use of the learned index is faced with many problems. Specifically, 1) How to make an appropriate trade-off between the Training-WR and Training-WW? 2) Learned index predicts the position of the key by machine learning models, which is not suitable for all workloads; 3) There may be performance loss for building the learned index in memory; 4) Bourbon does not consider the applicability of string keys and the conversion mechanism for strings has flaws in Bigtable.

2.4 Read-After-Delete

In this section, the Read-After-Delete problem is introduced. Large-scale delete operations are very common in RocksDB, including range deletes and secondary-range deletes, and so on [7], [34]. Most delete operations can be triggered by various logical operations, not limited to user-driven deletes, such as data migration and index deletion [34]. Below, two detailed application examples containing the large-scale deletion are given. 1) *Index deletion.* MyRocks [25] is a MySQL fork using RocksDB as its storage engine. In MyRocks, the first four bytes of each key identify the table or index to which the key belongs. Thus, dropping a table or index involves deleting all the keys with that prefix, and this delete operation is translated to a set of point and range deletes in RocksDB. 2) *Data migration.* The object storage Ceph that stores metadata in RocksDB is taken as an example in this study. Ceph organizes objects into PG (placement group), which is the smallest management unit for data recovery and data migration. As some old PGs are deleted one after another during the data migration, a large number of objects in PGs are deleted in a short period of time. The objects in the same PG are secondary (delete) keys in RocksDB, and the deletion of these objects is secondary-range delete.

In practical applications, it is found that in RocksDB, large-scale deletions seriously affect the performance of read operations including Get and Scan. As shown in Fig. 4, the scan performance fluctuations and space amplification caused by data migration of different scales are evaluated. As shown in Fig. 4a, the performance statistics are conducted on the cluster at a time interval of 60s, and data migration occurs at T0. The results show that with the increase of the migration scale, the performance degradation becomes more and more serious, and timeout appears after 50% PGs are deleted, making the system unavailable. Fig. 4b shows that the space amplification increases significantly with the ratio of deletion. This leads to the *Observation 2* in Section 1.

For this reason, RocksDB adopts the pattern of mark deletion, i.e., a delete inserts a tombstone that invalidates older instances of the deleted key. However, when RocksDB encounters a large number of delete operations, it will generate a lot of tombstones, which affects the read performance especially the scan performance (as shown in Fig. 4a). In addition, the tombstones will be stored and repeatedly compacted until the last layer, and they are actually deleted. It will cause serious space amplification, and the larger the deletion scale, the more serious the space amplification caused (as shown in Fig. 4b). This problem can be solved by two approaches: 1) Trigger the compaction operations regularly to merge the deleted data [25], [34]; 2) Organize the deleted data into a format, such as *range_del_table* [7] or reorder the data according to the delete keys [34]).

Challenge 2. How to better solve the Read-After-Delete problem for the KV store based on the LSM-tree? The existing solutions have some drawbacks: 1) Frequent triggering of compaction operations will affect performance and cause system stalls [35]; 2) The data reorganization is very expensive, which makes it not feasible for real systems.

3 TridentKV Design

3.1 Overall System Architecture

To solve these two challenges, TridentKV based on an adaptive learned index and partitioned storage is proposed in this paper. The overall TridentKV architecture is shown in Fig. 5.

Sub Partition. TridentKV adopts a space-efficient partitioned store. The data are stored in different partition structures (called Sub-trees) according to a certain standard, such as key range, secondary (delete) keys, and so on. TridentKV

uses a map to manage the mapping, and each Sub-tree is an LSM-tree structure in a disk. Like RocksDB, the memory structure of TridentKV is based on Memtable and Immutable. However, when TridentKV flushes the memory structure to the disk, instead of flushing the entire Memtable directly, it first stores the KV pairs in the Memtable into buffers through a well-designed partition scheduler and then flushes the buffers separately. The partition scheduler consists of data buffers for partitioning and a lightweight L0 index structure to speed up the L0 layer lookup. Unlike the traditional column family partitioning of RocksDB[7], all the partitions of TridentKV share one Memtable, which reduces storage overhead.

Learned Index Block. Like RocksDB, the disk structure of each Sub-tree is based on SStables. But TridentKV replaces the index block in SStable with an efficient learned index block to improve the read performance of SStable. The learned index block is composed of two modules: string process and learned index. The string process module converts the strings into integer numbers efficiently for model training. The learned index module exploits a two-layer linear model of RMI[30], which is sufficient to handle most workloads [30], [36]. Different from the traditional binary search, the learned index block directly predicts the location of the block through the model, which is faster than the binary search. Especially, the learned index block is much smaller than the traditional index block. In this case, more learned index blocks can be cached in memory, which is conducive to read performance.

Asynchronous Reading Design and SPDK. As shown in the left of Fig. 5, TridentKV manages the NVMe SSD through the high-performance SPDK interface, thus avoiding the system overhead caused by the Linux I/O path. Meanwhile, TridentKV adopts asynchronous IO for the read process and provides asynchronous interfaces for the upper and lower processes. The asynchronous IO makes it possible to reduce the synchronization control overhead caused by competing for I/O resources and optimize the front-end and background processing speed.

Read Process. As shown in Fig. 5, the read process is performed as follows. The upper-level program calls the asynchronous interface to access TridentKV. TridentKV first searches the Memtable, Immutable Memtable, and partition scheduler in turn in the memory. If these structures are not found, TridentKV searches the SStable files in the corresponding Sub-tree according to the map. For read files, TridentKV first reads the Bloom filter to determine the existence of the files. Then, it quickly locates the data block through the learned index block and loads the data block into memory to find the target key.

3.2 Adaptive Learned Index

In this section, the string process of the proposed TridentKV is introduced first. Then, the adaptive learned index building and learned index block are introduced.

String Process. In real applications, the key is often a variable of type string. According to the test of Sindex [37], most learned indexes fail to process strings efficiently. The Google team[31] proposed the String Number Base Coding (called SNBC in this paper), which is quoted as follows: *"we shift and re-base on a character by character basis, based on the range of characters observed at each character position. During training, we determine the minimum and maximum ASCII value of each character position over all keys. We then convert the keys into integers by choosing a different numerical base for each character position according to the range of values that position can take. "* Also, the team gave an example for SNBC: *"consider first how we encode numbers in base 10: 132 = 1 * 100 + 3 * 10 + 2. Now, consider that the only keys are "aab", "bdd", and "bcb". We need 2 values to encode the range of the 0th position, 4 values to encode the range of the 1st, and 3 values to encode the range of the 2nd. Therefore, "bcb" becomes 1 * (4 * 3) + 2 * (3) + 0."*

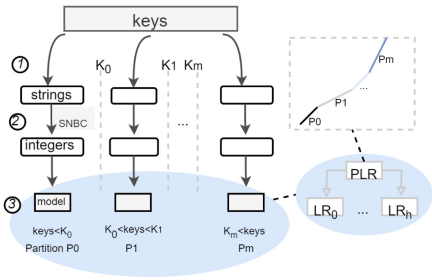


Fig. 6. String process and model structure.

The SNBC coding has obvious drawbacks. Specifically, when the strings in the same group have a large gap, the converted numbers will be very large and even exceed the upper limit of a double number. Based on the SNBC, the data is first grouped and then converted into numbers in this paper. As shown in Fig. 6, the keys are divided into several groups (P0 to Pm) with the ranges of the key. The keys in each group are very close so that the SNBC encoding is efficient. Meanwhile, a greedy strategy [37] is adopted to group the data in three steps. For the data that are sorted lexicographically, each time some records with the size of A_s are added to the current group. Then, SNBC is exploited to convert these strings into numbers, and the largest number is obtained. Also, a linear model is trained on the current group to obtain the average error. The keys are added as long as both the model error and the largest number are smaller than the corresponding thresholds (E_t and Max_t). If an adding process causes a violation of the thresholds, the most recently added records are removed from the current group. All the thresholds and parameters are configurable and have default values.

Model Structure. As shown in Fig. 6, a two-layer model is chosen. The first layer is a Piecewise Linear Regression (PLR) model[38], [39]. After the SNBC coding is finished, the number of each partition is obtained, and it is assigned to the corresponding polyline segment in the PLR. The second layer is composed of multiple Linear Regression (LR) models. This is a variant of the RMI[30] model, and it can handle most workloads in practical applications [36], [37]. If a learned index with Training-WW is used, the model must be monotonic to ensure that the records are still sorted by key and range scans can still be performed [31]. The monotonicity of the LR or PLR models is easy to guarantee because of the simplicity of these models.

Adaptive Training. TridentKV adopts an adaptive training approach that make trade-offs by combing Training-WW and Training-WR. The characteristics and shortcomings of Training-WW and Training-WR are as mentioned in Section 2.3.

The training strategy follows the guidelines: *1)* Training-WW should be avoided as much as possible in write-intensive workloads to prevent serious performance degradation and system unavailability; *2)* Training-WW is preferred for non-write-intensive workloads; *3)* If Training-WW is not used, Training-WR can be tried during the reading process; *4)* If the use of Training-WR cannot guarantee a lower error rate, it should not be used. Specifically, in TridentKV, the following adaptive training algorithm is used:

Algorithm 1. Adaptive Training

Input: the current number of write threads *cur_num*; An write requests threshold N_w; A bool value for whether to use Training-WW *t_w*; A bool value for whether to use Training-WR *t_r*; An error threshold *e_t*
Output: A bool value for whether to use Learned Index *use_mod*

```
1  use_mod ← False
2  t_w ← False, t_r ← False
3  Function TrainWhileWrite:
4    if cur_num < N_w then
5      t_w ← True
6      use_mod ← Train(t_w, t_r)
7      t_w ← False
8    return use_mod
9  Function TrainWhileRead:
10   if t_w == False then
11     t_r ← True
12     use_mod ← Train(t_w, t_r)
13     t_r ← False
14   Return use_mod
15 Function Train t_w, t_r:
16   if t_w or t_r then
17     if string type then
18       convert strings into numbers
19     train model with the numbers
20     if time out then
21       return False
22     if t_r then
23       err ← get model error range
24       if err ≥ e_t then
25         Return False
26     Return True
27   Return False
```

As shown in Algorithm 1, the adaptive training approach consists of the following steps: *1)* During the creation of the SStable, function *TrainWhileWrite()* is called to determine whether to use Training-WW. First the current number of write threads is determined. If it is less than the threshold N_w, the learned index with Training-WW will be used (function *TrainWhileWrite()* in Line 3). To this end, a global variable *cur_num* needs to be maintained to represent the current number of write threads. *2)* After the SStable is flushed, if Training-WW is not used, Training-WR is triggered from the background when the system is idle (function *TrainWhileRead()* in Line 9), and the state of the system can be judged by monitoring the processor usage regularly. *3)* The model is trained with Training-WW or Training-WR (function *Train()* in Line 15). If the training is timed out, the model is not used. If the model is trained with Training-WR, it is necessary to determine whether the error is greater than the error threshold (e_t) (see Section 2.3). If it is, the model is not used. *4)* For the SStables generated by compaction, the Training-WW will be used by default. All the thresholds are specified by users and have default values.

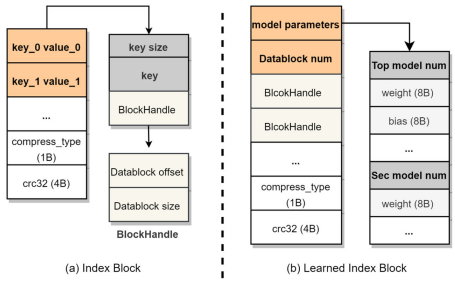


Fig. 7. Learned index block.

Learned Index Block. The traditional learned index is a memory structure suffering from the risk of performance loss. To address this issue, the traditional index block in the SStable is replaced with a learned index block. The format of the learned index block is shown in Fig. 7 B. The biggest difference between the learned index and the traditional index block is that the former does not need to store keys. In the figure, the block is divided into two parts. One part is model parameters that include all the parameters of the two layers. The other part is the BlockHandle of each block that includes the offset and size of each block. Though the offset and size are also available in traditional indexes, they are stored continuously in the learned index block. Meanwhile, some metadata information (checksums, etc.) that exists in the traditional index is also included in the learned index block.

The storage space consumption of the two methods is roughly compared. Assuming that for 1K KV pairs, an SStable with a capacity of 64M has approximately 65536 KVs. Assuming that each key is 16 bytes, 65536 * 16 bytes (about 1 MB) are needed to store all the keys. For the learned index block, each parameter is 8 Bytes. Assuming that there are 10 polyline segments in the first PLR model, the storage space of 10 * 8 * 2 + 1 * 8 bytes is needed, where 1 represents the number of polyline segments to be stored ("Top model num" in the figure). Assuming that there are 1000 sub-models in the second model, the storage space of 1000 * 8 * 2 +1 * 8 bytes is needed, where 1 is the "Sec model num". Besides, with the addition of 8B "Datablock num", the total storage space needed is less than 8 KB, which is more than 100 times smaller than the initial storage space consumption.

The read speed of the learned index block has been improved a lot compared with that of the traditional binary search, and the improvement is more obvious with the increase of the data volume of SStables. In addition, considering that the learned index is much smaller than the traditional index, more blocks of the learned index can be cached, which improves the overall read performance a lot.

3.3 Partition Scheduler

TridentKV solves the Read-After-Delete problem by partitioning. TridentKV distributes delete keys in the same area, and the range deletes directly delete the area. Based on this, the problem of performance stagnation after range deletes is completely avoided, which makes TridentKV efficient and fast. The use of RocksDB Column Family (CF) to implement the partitions is not recommended for TridentKV, because

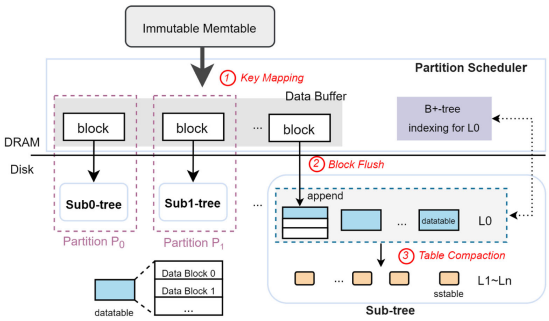


Fig. 8. Partition scheduler.

the CF-based partitioning (each CF maintains one Memtable) consumes too many Memtables and excessive memory space.

To alleviate this problem, a new partition mechanism, Partition Scheduler (called PS) is proposed, which only requires a Memtable to perform partition by two data buffer arrays and a light-weight L0 index structure. Since the additional buffer and index have upper limits of capacity, the memory space needed is much smaller than that of the CF or the method of directly partitioning in memory[40].

As shown in Fig. 8, the workflow of partition consists of three steps: 1) When the Immutable Memtable reaches the flush size, PS stores the KV pairs into data buffers according to the map and packs KV pairs as a block in each partition (called Key Mapping). 2) When the block size of a partition reaches the default size, PS flushes the block to the L0 layer of Sub-tree in the disk and builds B+tree in the background (called Block Flushing). In each Sub-tree, a certain number of blocks in the L0 layer form a Datatable that is similar to the data block part of the SStable, excluding the metadata part. 3) In the compaction process of the L0 layer, the Datatables are merged into SStables (called Table Compaction). That is, the L1 layer and above are all in the form of SStable, and the compaction and read processes are the same as those of RocksDB.

Key Mapping. The CF-based partitioning performs partition according to column families, and there is a Memtable structure in each partition. Different from this, PS uses only one Memtable and performs partition while the Memtable is flushed. Specifically, in Rocksdb, the Memtable flush process can be divided into three steps: 1) Call the function *Add ()* to pre-process KV pairs. In this step, the KV pairs are first processed by type, and their order is ensured. Then, they are inserted into the data block, and whether the KV pairs reach the block size is determined. 2) When the data block reaches the capacity limit of a block (default 4K), the function *WriteBlock()* is called to construct a block and compress the data in the block. Then, verification is performed according to requirements. 3) The function *Finish()* writes the data block into the file. The subsequent metadata blocks such as meta block, meta index block, index block, footer, and so on are constructed by 2) and written to the files by 3). For TridentKV, PS first divides KV pairs into the partition data buffers before 1). Each partition calls *Add()*, *WriteBlock*, and *Finish()* to process keys, construct and flush data blocks and does not construct metadata blocks. *Data buffers* consists of a write buffer and flush buffer. The KV pairs between different Immutables are covered by keys, and only one buffer may be out of order. To maintain the order of the data in every block, each Immutable is added to a write buffer first, and the write buffer is merged into the flush buffer. During the merge process, each partition is kept in order by key. If the capacity of a partition's flush buffer is full, the flush operation is performed.

Block Flushing. As mentioned before, PS performs flush operations in blocks, not SStable. A data block is flushed into the L0 layer in Sub-tree with appending. The current flush size of each partition is maintained in PS. When it reaches a certain size, PS divides different Datatables by recording the start and end position offsets. *Light-weight b +tree index*. However, flush in the unit of block will cause KV pair coverage between the blocks in each Datatable, and the read efficiency is very low. As shown in Fig. 8, PS constructs a light-weight memory b+tree to speed up the lookup of the L0 layer, similar to SLM-DB[41]. But PS only builds indexes for the fixed-size L0 layer, so the overall scale of B+-tree will not be very large. PS provides two implementations of persistent B+-tree and in-memory B+-tree for use. The implementation of persistent B+-tree refers to SLM-DB, which has high concurrency and high performance. However, because there is not much data in the L0 layer, even if the n-memory B-tree is used, the in-memory B +-tree can be restored within tens of seconds to minutes by scanning the L0 data when restarting.

Advantages. In addition to solving the Read-After-Delete problem, the partitioned store can effectively reduce the compaction cost of the real-world workloads with high spatial locality [40], [42]. Especially in LSM-based systems, the partitioned store can also improve read efficiency and reduce write magnification, because each partition has fewer data and levels. *Limitations*. The use of the partition method to solve the Read-After-Delete problem is closely related to the application, that is, the partition method and data mapping must be combined with the application. Besides, the partition method can only solve the problems caused by the deletion of the entire partition data that often appear in data migration, and it cannot solve the single deletion and the cross-partition deletion problem.

3.4 SPDK and Asynchronous Read

As mentioned earlier, the entire point-read process of RocksDB is synchronous. Each disk access is synchronized by calling *pread*. However, the synchronous IO does not scale well because handling the concurrent requests requires one thread for each request. In this case, context switches frequently, thus degrading the performance when the number of in-flight requests is higher than the number of cores. Therefore, asynchronous IO is adopted by TridentKV to exploit the high performance of fast IO devices.

The asynchronous design of the TridentKV reading process organizes the overall request link into layers. Specifically, for each TridentKV reading process, there are mainly four layers of differentiation, as shown in Fig. 9. The first layer is for reading in the memory, including *MtGet*, *FileFind*, and looking up in the cache; 2) The second layer is for reading the Bloom filter, *FBLoad*, and *FBMatch*; The third layer is for reading the index block, *IBLoad*, and *IBSeek*; The fourth layer is for reading the data block, *DBLoad*, and *DBSeek*. The *callback* function is registered at each layer. Then, the *callback* backtracking is performed layer by layer from the bottom to

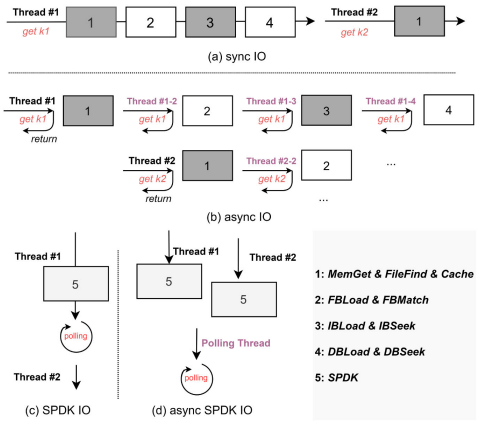


Fig. 9. Synchronous IO and asynchronous IO.

the top based on the asynchronous call relationship of some request links.

Compared with synchronous IO, the asynchronous IO process is similar to a pipeline, and it can make full use of every thread to enhance concurrency. As shown in Fig. 9a, as for synchronous IO, it is necessary to wait for *thread#1* to complete the search for k1 before the search for K2 starts. As for the asynchronous IO shown in Fig. 9b, once the first phase of the search is completed, *sub-thread#1-2* starts the subsequent phases. Once *thread#1* returns asynchronously, *thread#2* starts to search for k2. This process differs from multithreading because the latter requires more CPU resources. Currently, only the asynchronous transformation of point reading is implemented. The scan and other operations are the original interfaces, the asynchronous implementation of these operations will be our future work.

Besides, the use of SPDK directly on TridentKV does not improve performance. Considering the polling-based SPDK I/O, it is difficult to improve the CPU utilization during I/O waits since the threads co-exist on the same cores. However, as shown in Fig. 9d, with asynchronous IO and the use of SPDK, a separate thread (called *Polling Thread* in the figure) is used for polling, and the original IO thread directly returns asynchronously. This greatly reduces the latencies caused by polling.

3.5 Implementation

TridentKV is implemented on RocksDB (V5.4.0) and is successfully packaged into Ceph (v12.2.13). Also, it can be used as a KV storage with high read performance. All the codes in TridentKV are written in C++, and the machine learning model is implemented based on the MKL (Math Kernel Library) library[43]. TridentKV retains the original interface of RocksDB, and the three techniques are implemented as plug-ins. That is, they are optional configurations provided to users.

4 Evaluation

In this section, the evaluation results are presented to demonstrate the advantages of TridentKV. Specifically, extensive experiments are conducted to answer the following questions:

1) What are the advantages of TridentKV in terms of performance? (Section 4.2);
2) What are the effects of the three techniques of TridentKV on the system performance? (Section 4.3);
3) What are the advantages of TridentKV in terms of the performance in Ceph? (Section 4.4).
4) How does TridentKV solves the Read-After-Delete problem? (Section 4.5).

4.1 Experiment Setup

All the experiments are conducted on a server from the Sensetime Research, and the server is equipped with an Intel Skylake Xeon Processor (2.40 GHz) and 54 GB of memory. The operating system running on the server is Centos 7 with the kernel of 64-bit Linux 3.10.0. The storage drive is Intel DC NVMe SSD (P4510, 2 TB). The basic read and write performances of the NVMe SSD are 3200 MB/s (sequential read), 637K IOPS (random read), 2000 MB/s (sequential write), and 81.5K IOPS (random write). All experiments are performed on the NVMe SSD by default, and only one group of tests are performed on SSD and HDD, i.e., Samsung SATA SSD (PM883, 3.84 TB) and Seagate Exos 7E2 3.5 HDD (2 TB).

We compare TridentKV with RocksDB, the base of TridentKV's development and other four state-of-the-art production KV stores. They are: 1) *WiscKey or Titan* [44], [45]: the representative high-performance LSM-based KV stores. The main idea is key-value separation, a very typical and effective technology for LSM-tree. Titan is used in the experiment mainly because it is based on industrial implementation and has higher performance than WiscKey. 2) *Bourdon* [28]: the KV store using learned index. 3) *Kvell* [26]: the state-of-the-art high-performance KV store designed for modern NVMe SSD devices. 4) *Wiredtiger*[46]: the representative B-tree-based KV stores. Ceph using TridentKV (called Ceph-TridentKV) is compared with Ceph using RocksDB (called Ceph-RocksDB). To make a fair comparison, all the databases take one thread for compaction and one thread for flushing. The size of Memtables, Immutable Memtables, and SStable is set to the default values in RocksDB.

4.2 Bench Performance

Because TridentKV is optimized for RocksDB, the basic read and write performance of the two KV stores is evaluated first using the db_bench [7] released with RocksDB. Then, TridentKV is compared with RocksDB and four other state-of-the-art KV stores under the YCSB benchmark [17] proposed by Yahoo!. The overall volume of the inserted data is 40 GB by default. The size of the keys is 16 B, and the values are 1K. To show the optimization effect of read performance, the size of the operating system cache is set to a relatively small value, while that of the KV store cache is set to a normal one, about 10% of the data sets.

Read Performance. Figs. 10 and 11 show the random read (called RR) and sequential read (called SR) performances of the two KV stores varying value size, respectively. The size of the keys is 16 B, and the size of the values varies from 16 B to 16 KB. Based on the comparison of the test results, the following conclusions can be drawn: *1) TridentKV can significantly improve the random read performance*. It can be seen from Fig. 10 that compared to RocksDB, TridentKV's random read performance is improved by 7× to 12×. *2) TridentKV can*

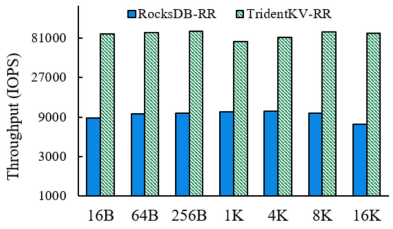


Fig. 10. Random read with different value sizes.

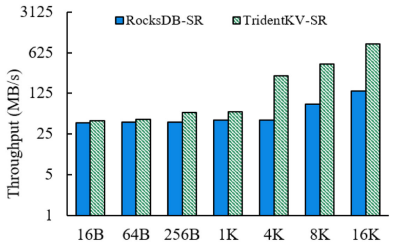


Fig. 11. Sequential read with different value sizes.

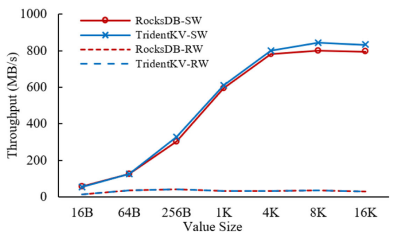


Fig. 12. Write performance.

improve the sequential read performance. The result in Fig. 11 shows that the sequential read performance of TridentKV is improved by 30%$\sim$ 6.5$\times$ compared to RocksDB, and a better performance improvement is achieved for a larger value size. The insignificant performance improvement of SR is mainly because RocksDB exploits an iterator-based implementation of sequential read. After a key is read, the block where the key is located is loaded into the memory to speed up the subsequent key reading. TridentKV is also based on this implementation. Thus, all its read operations are performed in memory, and TridentKV's learned index improves performance. The asynchronous design of TridentKV is effective only when the key size is larger than the block (4K).

Varying Devices. The read performance of the two KVs on different devices using 16B keys and 1K values is compared. Fig. 13 shows that TridentKV's random read performance is improved by 3$\times$ to 6$\times$ and sequential read performance by about 30% in HDD, SSD, and NVMe SSD. Especially, the performance improvement on NVMe SSD is more obvious, because the main latency in slow devices is incurred by data accessing (Section 2.2), and learned index and asynchronous read are more effective for fast devices (Sections 2.3 and 3.4). Besides, SPDK is only available on NVMe-based devices.

Write Performance. It can be seen from Fig. 12 that TridentKV and RocksDB have little difference in the random write (RW) performance, while the sequential write (SW) performance of TridentKV is slightly better than that of RocksDB.

The YCSB Benchmark. The performance of each KV store is evaluated on the YCSB benchmark. The YCSB benchmark

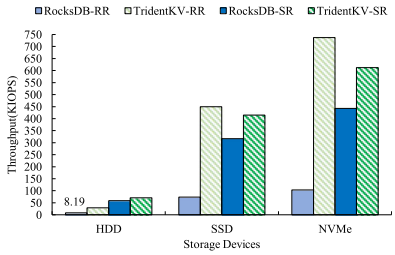


Fig. 13. Read with different storage devices.

TABLE 1
YCSB Workloads

Workloads	A	B	C	D	E	F
Operations	R:50% U:50%	R: 95% U: 5%	R: 100%	R: 95% I: 5%	S: 95% I: 5%	R: 50% M: 50%
Req. Dist.	Zipfian			Latest	Zipfian	

R: Read, U: Update, I: Insert, S:Scan, M: Read-Modify-Write.

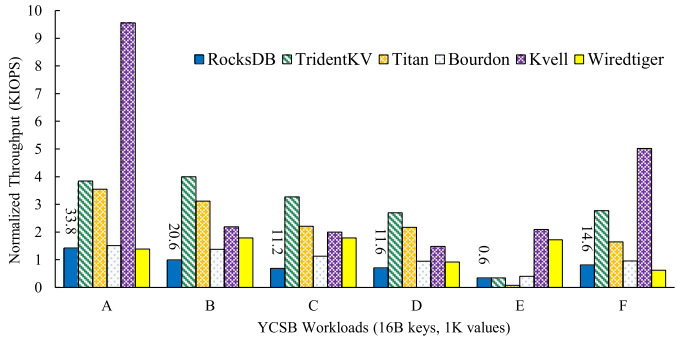


Fig. 14. YCSB benchmark.

is an industrial standard micro-benchmark suite delivered by Yahoo!. The seven representative workloads of YCSB are listed in Table 1. Specifically, the workload Load indicates the load process of constructing a database with a capacity of 40 GB; the workload A is composed of 50% reads and 50% updates; the workload B consists of 95% reads and 5% updates; the workload C includes 100% reads; the workload D consists of 95% reads and 5% latest keys insert; the workload E is composed of 95% range queries and 5% keys insert; the workload F includes 50% reads and 50% read-modify-writes. The cache size is set to 256M in all KV stores.

As shown in Fig. 14, TridentKV outperforms RocksDB by 2.6$\times$ to 5$\times$ on all the workloads except the workload E. Specifically, for *read-intensive workloads* (YCSB B, C, D), TridentKV performs much better than other KVs. With learned index, asynchronous read, and SPDK, the performance of TridentKV is improved by 4$\times$ compared with RocksDB. The read performance of Titan is improved by about 2-3$\times$ compared with RocksDB because the cache can store more keys through key-value separation. Kvell and Wiredtiger shows about 1-2$\times$ performance improvement, and Bourbon presents little performance improvement. For *write-intensive workload* (YCSB A and F), Kvell achieves extremely excellent performance by in-memory indexing and non-sorting of disk data. However, while the full-memory index brings performance improvements, it also brings difficulties for Kvell

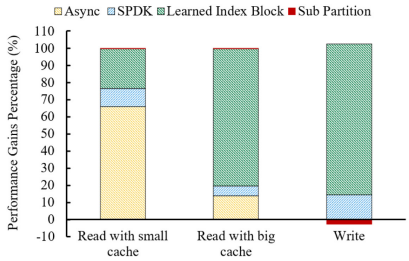


Fig. 15. The performance gains percentage of the three technologies.

to recover when restarting. Except Kvell, TridentKV performs better than other KVs.

For *scan workloads* (YCSB E), TridentKV fails to improve the scan performance mainly because its scan interface is not asynchronous, and the learned index does not improve the scan performance. In Titan, due to key-value separation, the value data needs to be searched separately, and the scan performance of Titan is severely degraded. KVell and wiredtiger perform well. In general, TridentKV is superior to other KVs in overall performance, especially in read-intensive workloads. Besides, Kvell and Bourbon are all research prototypes that do not have many functions and cannot be used as a complete KV store in real systems. By contrast, RocksDB is the most widely used and most stable KV store. TridentKV is based on Rocksdb and retains all the features of Rocksdb. The three technologies of TridentKV are implemented as plug-ins with high applicability and stability for use in different scenarios.

4.3 The Impact of the Three Techniques

The impact of the three techniques on the performance of TridentKV is investigated. Since the three techniques are implemented as plug-ins, the performance gains of the three techniques are tested by changing the control variables. The test uses random read and wirte of db_bench, and the data set is 40G. The small cache is 2G, accounting for 5% of the data set; the large cache is 20G, accounting for 50% of the data set. The percentage of effects of the three techniques on the read and write performance are illustrated in Fig. 15, and the following conclusions can be drawn:

1) For the read operations of various loads, *asynchronous IO brings the greatest performance improvement for a small cache.* 2) *For a large cache, most of the load does not need to access the disk, and the learned index has the maximum effect on the performance.* 3) For writing, none of the three techniques brings significant performance improvements. The write performance of the PG partitions decreases slightly, which is mainly caused by the learned index and SPDK. Next, this result is analyzed by showing the performance of each technique. Since the partition does not affect performance, it is not shown in the performance test.

Learned Index Block. The advantages of the Learned Index Block are verified through two groups of tests. The comparisons include 1) RocksDB using binary search index, 2) RocksDB using learned index block with Training-WW (called RocksDB-with-WW) and 3) with Training-WR (called RocksDB-with-WR), and 4) TridentKV that only uses adaptive training and does not use the other two technologies. In TridentKV, the threshold N_w is set to 8, and the error threshold (e_t) is set to 6. These values are set based on the experiment in Fig. 3, for example, N_w is set to 8 because it is found that when the number of write requests is greater than 8, the performance of RocksDB (with Training-WW) begins to decrease. The workloads include Linear, Seg 1%, AR, and OSM, representing linear, polyline, normal, and irregular distribution datasets, respectively. Besides, two sets of more irregular data, AR-shuffle and OSM-shuffle, are specially added. They are generated by randomly shuffling AR and OSM.

The first group is the robustness evaluation of the adaptive training and learned index. In different workloads, we first perform random write operations under different threads. And we perform random reads after a period of time, when TridentKV may try Training-WR. Fig. 19 shows the write performance in AR workload and the results are similar in different workloads. Overall, TridentKV achieves the best performance under different threads. For the Training-WR way, RocksDB-with-WR has basically no impact on write performance than RocksDB. For the Training-WW way, RocksDB-WW has improved performance when the number of write threads is relatively small (less than 8). This is mainly because the learned index structure of SStable is simpler and can be cached more in memory, and fast index speed up the compaction process, and so on. But when the write thread is greater than 8, its performance is severely degraded, which has been analyzed in Section 2.3. For TridentKV, when the write pressure is less, the Training-WW way is adopted, but not when the write pressure is relatively high. TridentKV can obtain the highest write performance in various situations.

The read performance varying different workloads is shown in Fig. 20, where Fig. 20a shows the read performance after write operation using one thread and Fig. 20b shows eight threads. The error range of the model under different workloads ranges 0 to 9. As shown in Fig. 20a, the performance of RocksDB-with-WW is not affected by the error range and is improved a lot. TridentKV adopts the Training-WW way after write operation using single thread and the performance is similar to RocksDB-with-WW. For the Training-WR way, the random read performance drops severely as the error range increases. In particular, in AR-shuffle and OSM-shuffle, the performance of RocksDB-with-WR is lower than RocksDB. As shown in Fig. 20b, although TridentKV adopts the Training-WR way after write operation using eight threads, but there will be no performance degradation like RocksDB-with-WR. There are two reasons: 1) TridentKV will not use Training-WR in the case of a high error range (greater than 6), which ensures that the performance of TridentKV will not decrease and be lower than RocksDB; 2) The Training-WW way is not used in the SStable generated by the initial insertion, but some SStables generated by compaction may use Training-WW, and the performance of these SStables will not be affected by the error ranges.

The second group is to analyze the advantages of the learned index over the traditional binary search index, and the results using db_bench are shown in Figs. 16 and 17. The random read performances under different value sizes and different cache sizes are compared, and it is found that the learned index mainly improves the read performance from two aspects: 1) It can be seen from Fig. 16 that the

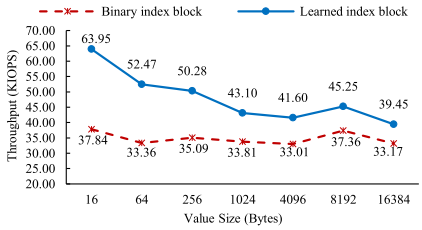


Fig. 16. Random read performance with different value sizes.

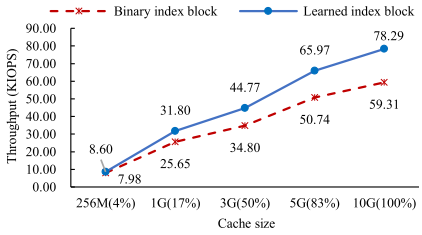


Fig. 17. Random read performance with different cache sizes.

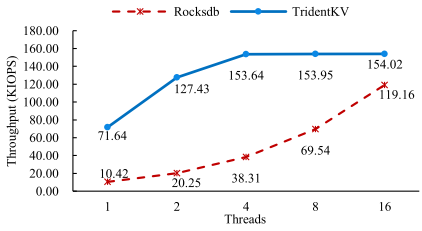


Fig. 18. Random read performance with different number of threads.

smaller the KV pairs, the more obvious the improved performance of the learned index. This is mainly because the smaller the value size, the more KV pairs in the block. This reduces the search speed of the binary index block, but the learned index is not affected. The model becomes more accurate as the number of data increases. 2) It is shown in Fig. 17 that the larger the cache, the more obvious the improved performance of the learned index. This is mainly because the learned index block is smaller than the binary index block and more blocks can be cached. The larger the cache, the more obvious the performance advantage.

Asynchronous and SPDK. To analyze the ways that the asynchronous IO and SPDK improve performance, the impact of TridentKV and RocksDB on performance under different threads is investigated, and the result is shown in Fig. 18. It can be seen that the synchronization performance increases with the number of threads. In contrast, the asynchronous IO achieves very high performance with few threads, but the performance does not increase with more threads. The reason is found by comparing the CPU occupancy. With more processing threads, an improved concurrency is achieved by the asynchronous design. In this case, the CPU occupancy also increases and reaches saturation, which makes the performance bottleneck shift to the CPU slowly.

4.4 Ceph Performance

In this section, the random read performance of Ceph on TridentKV and RocksDB is evaluated to show the advantages of TridentKV. The rados_bench released with Ceph is used for performance evaluation. For the sake of fairness,

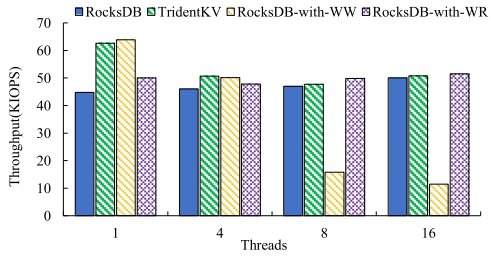
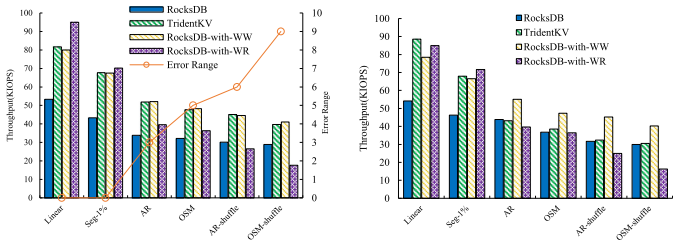


Fig. 19. Random write performance with different number of threads.



(a) Under single write thread (b) Under eight write threads

Fig. 20. Random read performance under different workloads.

Bluestore is selected as the back-end storage, and all parameters are set to the default values. The performance is compared from two aspects: 1) testing the extreme performance of the object with a size of 16K by running multiple bench clients; 2) comparing the extreme performance under different object sizes.

1) The result in Fig. 24a shows that the performance of Ceph reaches its limit when two bench clients are run. The random read performance of Ceph on TridentKV is 30%∼60% higher than that of RocksDB, and it is very stable without fluctuations. 2) The result in Fig. 24b shows that for the extreme random read performance of Ceph on TridentKV under different object sizes is 50% higher than that of RocksDB, which is because of the excellent read performance of TridentKV.

4.5 Read-After-Delete

In this section, the performance of TridentKV for handling the Read-After-Delete problem is tested. To facilitate the test, three strategies in Ceph are compared, including the PG-based partitioning strategy of TridentKV (called PG Partition), column family partitioning strategy (called CF Partition), and non-partitioning. The Ceph data migration is simulated by deleting PGs, and different migration scales are shown by deleting different numbers of PGs.

Performance Fluctuations. The first group compares the performance changes caused by different partitioning strategies before and after data migration. Specifically, 10 PGs are set, and the performance statistics are performed on the cluster every 60 s. Data migration occurs at T_0 when the data of one PG is deleted. Fig. 21 shows that after the data migration, the performance degradation under the non-partitioning strategy is obvious, and it is mainly caused by the mark deletion. The performance of the other two partitioning strategies is not affected by the deletion, and the performance increase after the deletion is because the amount of data is reduced. However, due to the additional index, the performance improvement of the PG-based partitioning strategy is lower than that

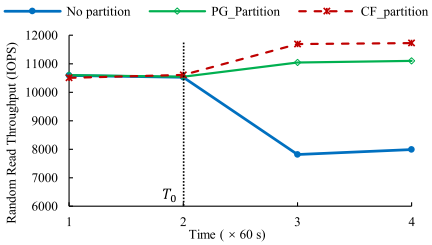


Fig. 21. Performance changes in migration.

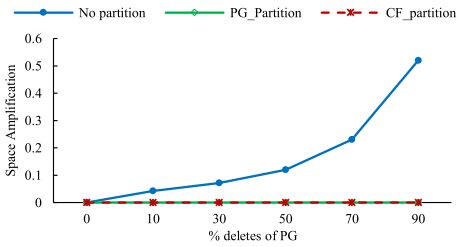


Fig. 22. Space amplification.

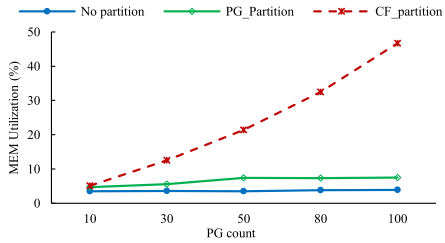


Fig. 23. Mem utilization.

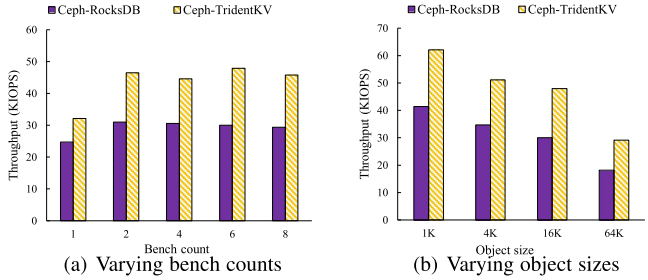


Fig. 24. Random read performance in Ceph with different KV stores.

of the column family partitioning strategy, but the performance difference between the two strategies is not large.

Space Amplification. The second group evaluates the space amplification of different partitioning strategies. According to the calculation method proposed by Lethe[34], the space amplification of the three strategies under different migration scales is counted. Specifically, 10 PGs are set first, and different numbers of PGs are deleted. Then, the space occupied before and after the deletion is counted to evaluate the space amplification. It can be seen from Fig. 22 that the space amplification under the non-partitioning strategy increases significantly with the deletion ratio, while the other two partitioning strategies cause no space amplification due to data migration. This is mainly because in these two partitioning strategies, unlike the traditional mark deletion method, the migration deletes the entire partition immediately.

Memory Usage. The third group compares the memory usage of the three strategies under different numbers of PGs. Specifically, the number of PGs is increased sequentially, and the memory usage under different partitioning strategies is observed. As shown in Fig. 23, as the number of PGs increases, the performance of the non-partitioning strategy is not affected, while the performance of the TridentKV's PG partition increases to about 6% and no longer increases. This is because the memory space used by the PG partition schedule has an upper limit. As for the CF partition strategy, the number of CFs and Memtables increases with the number of PGs, and the memory usage increases significantly. Of course, the size of the Memtable in each column family can be reduced, but each SStable will be small.

Therefore, it can be concluded that the PG partition can ensure that its performance is not affected by Ceph data migration, and it does not occupy too much memory.

5 Related Work

5.1 The Read Optimization for LSM-Based KV Stores

The read performance of the LSM-tree is sacrificed due to the multi-layer structure, and it is mainly optimized by the existing research from three directions: filter, index structure, and cache optimization. *Filter Optimization.* BLSM [23] first used Bloom filter to quickly determine whether the KV item exists without actually reading it. However, the Bloom filter also suffers excessive memory usage, high false-positive rates, unsupported range queries, and so on. To reduce memory usage, ElasticBF [47] adopted a fine-grained heterogeneous Bloom filter management scheme with dynamic adjustment according to data hotness. Succinct Range Filter (SuRF) [48] supported common range queries with a fast and compact data structure for approximate membership tests. *Auxiliary Index Structure.* SLM-DB [41] implemented a persistent global B+ tree index on NVM, and Kvell [26] adopts a variety of memory index structures. Bourbon [28] built learned index through static data stored in SStables to speed up the search on SStables. These studies either changed the multi-layer structure of LSM-tree or add auxiliary indexes. The former greatly affects the write performance of the LSM-tree, while the latter increases memory overhead. TridentKV integrates the learned index instead of an additional memory index structure into SStable to reduce memory overhead. *Cache Optimization.* Ac-Key [49] proposed an adaptive cache strategy for different loads by building a variety of cache components. LSBM [50] found a large number of cache failures after compaction, which is alleviated by increasing the cache size for the merge operation. Though increasing the cache size can improve the read performance greatly, it is not desired because it makes the memory insufficient.

These studies did not carefully analyze the read performance bottleneck of RocksDB, nor pointed out the Read-After-Delete problem. TridentKV greatly improves the read performance of RocksDB and solves the read performance fluctuation caused by large-scale data deletion.

5.2 Partitioned Store

The partitioned store can effectively reduce the compaction cost because of the high spatial locality of real-world workloads. Especially, the partitioned store can reduce the time

and space requirement of the compaction processes in LSM-based systems. Rocksdb provides a way to partition by column families, while EvenDB[40] partitions in memory. These methods all suffer from excessive memory usage, and the Rmix [42] describes the partition method briefly. A partition strategy with an upper capacity limit is adopted in this paper. It is not limited to the partition within the key range and can follow any standard. In Ceph, the PG partition is exploited, and it solves the Read-After-Delete problem perfectly.

5.3 Learned Index

The learned index structure proposed by Tim Kraska[30] has been studied extensively in recent years. The fundamental of this structure is that the index is a model, and the traditional indexes can be replaced with the index structures that are built by efficient machine learning models to achieve good performance. Meanwhile, the original learned index has some drawbacks, such as read-only, single-thread, in memory, and does not support string keys. Researches on ALEX [51], XIndex [52], and so on provide solutions to improve the write performance and concurrency of the learned index. Sindex [37] is committed to supporting string keys. Many studies exploited the learned index in actual systems: Xstore [36] uses it to cache RDMA-based KVs; Bourbon [28] applies it to leveldb, and the Google team [31] uses it in BIgtable. TridentKV applies the learned index to Rocksdb for the first time and improves the way of building the model. Also, it integrates the learned index into a block and supports string keys well.

6 Conclusion

In this study, TridentKV, a read-optimized LSM-tree-based KV store, is developed for read-heavy workloads. In TridentKV, a space-efficient partition strategy is adopted to solve the Read-After-Delete problem. Also, an optimized learned index block structure is designed to speed up file reading. Besides, an asynchronous design is exploited, and SPDK is supported to achieve high concurrency and low latency. TridentKV is implemented on RocksDB with real NVMe devices. The evaluation results indicate that TridentKV achieves high and stable read performance.

References

[1] C. Lai *et al.*, "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–14.

[2] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[3] G. Sanjay and D. Jeff, "LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values," 2016. [Online]. Available: https://github.com/google/leveldb/

[4] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[5] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: Ram space skimpy key-value store on flash-based storage," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 25–36.

[6] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A database benchmark based on the facebook social graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1185–1196.

[7] "Rocksdb: A persistent key-value store for fast storage enviroments," 2021. [Online]. Available: https://github.com/facebook/rocksdb/

[8] D. Huang *et al.*, "TiDB: A raft-based HTAP database," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[9] "Mysql," 2021. [Online]. Available: https://www.mysql.com/

[10] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[11] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 283–296.

[12] A. Anwar *et al.*, "Customizable scale-out key-value stores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2081–2096, Sep. 2020.

[13] A. Vidwansa, "Architecting high performance storage for AI, HPC, and big data," 2019. [Online]. Available: https://www.youtube.com/watch?v=vuVKOtOCasg&t=4s/

[14] R. Rehrmann, C. Binnig, A. Böhm, K. Kim, and W. Lehner, "Sharing opportunities for oltp workloads in different isolation levels," *Proc. VLDB Endowment*, vol. 13, no. 10, pp. 1696–1708, 2020.

[15] R. Rehrmann, M. Keppner, W. Lehner, C. Binnig, and A. Schwarz, "Workload merging potential in sap hybris," in *Proc. Workshop Testing Database Syst.*, 2020, pp. 1–6.

[16] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 191–208.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[18] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–223.

[19] L. Avinash and M. Prashant, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[20] D. Andy, "Getting started with LevelDB," Packt Publishing Ltd., 2013.

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

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Oper. Syst. Des. Implementation*, 2006, pp. 307–320.

[23] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 217–228.

[24] F. Wu, "Improving point-lookup using data block hash index," 2018. [Online]. Available: https://rocksdb.org/blog/2018/08/23/data-block-hash-index.html

[25] Y. Matsunobu, S. Dong, and H. Lee, "MyRocks: LSM-tree database storage engine serving facebook's social graph," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[26] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: The design and implementation of a fast persistent key-value store," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 447–461.

[27] K. Kourtis, N. Ioannou, and I. Koltsidas, "Reaping the performance of fast NVM storage with udepot," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 1–15.

[28] Y. Dai *et al.*, "From wisckey to bourbon: A learned index for log-structured merge trees," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 155–171.

[29] A. C. De Melo, "The new linux'perf'tools," in *Proc. Slides Linux Kongress*, 2010, pp. 1–42.

[30] T. Kraska, A. Beulearntel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 489–504.

[31] H. Abu-Libdeh, "Learned indexes for a google-scale disk-based database," 2020, *arXiv:2012.12501*.

[32] "Amazon customer reviews dataset," 2021. [Online]. Available: https://registry.opendata.aws/amazon-reviews/

[33] "Open street maps," 2021. [Online]. Available: https://www.openstreetmap.org/#map=4/38.01/95.84/

[34] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A tunable delete-aware LSM engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 893–908.

[35] T. Yao *et al.*, "Matrixkv: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. USENIX Annu. Tech. Conf. USENIX*, 2020, pp. 17–31.

[36] X. Wei, R. Chen, and H. Chen, "Fast RDMA-based ordered key-value store using remote learned cache," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 117–135.

[37] Y. Wang, C. Tang, Z. Wang, and H. Chen, "Sindex: A scalable learned index for string keys," in *Proc. 11th ACM SIGOPS Asia-Pacific Workshop Syst.*, 2020, pp. 17–24.
[38] J. Acharya, I. Diakonikolas, J. Li, and L. Schmidt, "Fast algorithms for segmented regression," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2878–2886.
[39] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *Proc. IEEE Int. Conf. Data Mining*, 2001, pp. 289–296.
[40] E. Gilad *et al.*, "EvenDB: Optimizing key-value storage for spatial locality," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.
[41] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 191–205.
[42] W. Zhong, C. Chen, X. Wu, and S. Jiang, "REMIX: Efficient range query for LSM-trees," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 51–64.
[43] E. Wang *et al.*, "Intel math kernel library," in *Proc. High-Perform. Comput. Intel® Xeon Phi™*, 2014, pp. 167–188.
[44] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, 2017.
[45] "Titan: A rocksdb plugin to reduce write amplification," 2021. [Online]. Available: https://github.com/tikv/titan
[46] "WiredTiger storage engine," 2018. [Online]. Available: https://docs.mongodb.com/manual/core/wiredtiger/
[47] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 739–752.
[48] H. Zhang *et al.*, "SuRF: Practical range query filtering with fast succinct tries," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 323–336.
[49] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du, "AC-Key: Adaptive caching for LSM-based key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 603–615.
[50] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang, "LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 68–79.
[51] J. Ding *et al.*, "ALEX: An updatable adaptive learned index," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 969–984.
[52] C. Tang *et al.*, "XIndex: A scalable learned index for multicore data storage," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 308–320.

Kai Lu received the bachelor's degree in computer science from Huazhong University of Science and Technology(HUST), China, in 2018. He is currently working toward the PhD degree with Computer Science Department, HUST. His research interests include computer architecture and key-value storage system.

Nannan Zhao (Member, IEEE) is currently an associate professor with the School of Computer Science, Northwestern Polytechnical University. Her research interests include distributed storage systems, key-value stores, flash memory, and Docker containers.

Jiguang Wan received the bachelor's degree in computer science from Zhengzhou University, China, in 1996 and the MS and PhD degrees in computer science from the Huazhong University of Science and Technology (HUST), China, in 2003 and 2007, respectively. He is currently a professor with the Wuhan National Laboratory for Optoelectronics, HUST. His research interests include computer architecture and key-value storage system.

Changhong Fei received the bachelor's degree in computer science from the Huazhong University of Science and Technology (HUST), China, in 2019. He is currently working toward the MS degree with Computer Science Department, HUST. His research interests include computer architecture and key-value storage system.

Wei Zhao is currently a senior researcher with SenseTime Research. His research interests include machine learning and storage system.

Tongliang Deng is currently a system researcher with SenseTime Research. His research interests include machine learning and storage system.