Computers, Materials & Continua
DOI: 10.32604/cmc.2022.029566
Article

Tech Science Press

Key-Value Store Coupled with an Operating System for Storing Large-Scale Values

Jeonghwan Im[1] **and Hyuk-Yoon Kwon**[2,*]

[1]Graduate School of Data Science, Seoul National University of Science and Technology, Seoul, Korea
[2]Department of Industrial Engineering, Graduate School of Data Science, Research Center for Electrical and Information Science, Seoul National University of Science and Technology, Seoul, Korea
*Corresponding Author: Hyuk-Yoon Kwon. Email: hyukyoon.kwon@seoultech.ac.kr
Received: 07 March 2022; Accepted: 20 April 2022

Abstract: The key-value store can provide flexibility of data types because it does not need to specify the data types to be stored in advance and can store any types of data as the value of the key-value pair. Various types of studies have been conducted to improve the performance of the key-value store while maintaining its flexibility. However, the research efforts storing the large-scale values such as multimedia data files (e.g., images or videos) in the key-value store were limited. In this study, we propose a new key-value store, WR-Store++ aiming to store the large-scale values stably. Specifically, it provides a new design of separating data and index by working with the built-in data structure of the Windows operating system and the file system. The utilization of the built-in data structure of the Windows operating system achieves the efficiency of the key-value store and that of the file system extends the limited space of the storage significantly. We also present chunk-based memory management and parallel processing of WR-Store++ to further improve its performance in the GET operation. Through the experiments, we show that WR-Store++ can store at least 32.74 times larger datasets than the existing baseline key-value store, WR-Store, which has the limitation in storing large-scale data sets. Furthermore, in terms of processing efficiency, we show that WR-Store++ outperforms not only WR-Store but also the other state-of-the-art key-value stores, LevelDB, RocksDB, and BerkeleyDB, for individual key-value operations and mixed workloads.

Keywords: Key-value stores; large-scale values; chunk-based memory management; parallel processing

1 Introduction

As big data platforms have been actively used in various fields such as banking and Social Network Services (SNS), not only has the data size increased, but also the data types become various [1–8]. Accordingly, if we fix the data types to be stored as in the relational databases, the supported data types are not flexible, which becomes the limitation of the platform. For this case, the key-value store

could be an alternative because it does not need to specify the data types to be stored in advance and can store any type of data as the value of the key-value pair [9]. The representative key-value stores are RocksDB, LevelDB, BerkeleyDB, and Redis.

For some applications, we need to manage large-scale values with various data types. For example, various data types such as photos and videos can be used in SNS posts and email attachments. In relational databases, these large-scale media data are defined and stored as a special data type, called Binary Large Object (BLOB). BLOB refers to a collection of binary data stored as one entity in the Database Management Systems (DBMS). BLOB data types cannot be efficiently handled in the DBMS because the benefit of the structured storage cannot be utilized because of their unfixed data types and variable data length [10]. Since the key-value store does not need to enforce a fixed data type, it can be an appropriate alternative[1,2].

There have been lots of research efforts to improve the performance of the key-value store while maintaining its flexibility of data types: the performance optimization by reducing amplification that occurred during compaction [11,12], the efficient merge of key ranges in the memory to resolve the hardware bottleneck [13–15], improvement of the filter algorithm to support fast search of the keys [16–19], performance improvement of mapping between keys and values by storing the hashtable in GPU [20], and parameter tuning to optimize the performance [21]. However, the research efforts to store the large-scale values such as multimedia data files (e.g., images, videos) in the key-value store were limited. Recently, object storages such as Amazon S3, Apache Ozone, and Microsoft Azure have been widely used in cloud environments to store large-scale data. Nevertheless, it is required to support large-scale values in key-value stores because they have the advantage of storing data of various types and sizes.

In this study, we propose a new key-value store, WR-Store++, by enhancing the existing key-value store, WR-Store [22], so as to store the large-scale values. WR-Store is tightly coupled with the built-in data structure in the Windows operating system, and consequently, it provides lightness and comparable or even better performance than the existing key-value stores. However, it has a limitation to the maximum data size to be stored due to the inherent characteristics of the used built-in data structure. WR-Store++ overcomes the limitation by storing only the index part in the built-in data structure and the entire data in the file system. Through the extensive experiments, we show that WR-Store++ dramatically enhances the limitation in the data storage of WR-Store and has performance advantages compared to the representative key-value stores, i.e., LevelDB, RocksDB, and BerkeleyDB, in particular, for large-scale values. The contributions of this study can be summarized as follows:

1. Lightness and portability: WR-Store++ maintains the WR-Store's advantages by utilizing the built-in structure and the file system provided by the operating systems. It allows easy installation and supports various computing environments including data centers, servers, desktop PCs, and mobile devices. Once the Windows operating system is installed, we can easily migrate the data between even different computing environments. Even if WR-Store++ depends on the Windows operating system, we can easily migrate the key-values stored in WR-Store++ into the other key-value stores.
2. Extensibility: Due to the extensible structure of WR-Store++, it can provide much larger storage than the existing WR-Store. In the experiments, we confirm that WR-Store++ can support at least 32.71 times larger dataset than WR-Store while even some of the commercial

[1] BerkeleyDB C API Reference, https://docs.oracle.com/cd/E17276_01/html/index.html

[2] RocksDB Integrated BlobDB, http://rocksdb.org/blog/2021/05/26/integrated-blob-db.html

key-value stores cannot support such dataset or their performances degrade dramatically by the actual experiments.

3. Efficiency: We show that the overall performance of WR-Store++ is comparable to or even better than the commercial key-value stores, LevelDB, RocksDB, and BerkeleyDB. In particular, the performance improvement of WR-Store++ over the other key-value stores becomes larger as the value size increases: by up to 3.71∼44.78 times for the PUT operation; 1.91∼2.19 times for the GET operation; 37.97∼2456.91 times for the DELETE operation. Three kinds of workloads combining multiple operations confirmed the efficiency of WR-Store++. Specifically, WR-Store++ outperforms BerkeleyDB by 1.69∼1.79 times, RocksDB by 2.59∼2.99 times, and LevelDB by 2.22∼2.39 times, respectively. We also present chunk-based memory management and parallel processing of WR-Store++ to further improve its performance in the GET operation by employing the chunk-based memory management and parallel processing, improving the basic WR-Store++ by 6.39 times and 8.66 times, respectively.

In Section 2, we describe the related work. In Section 3, we explain the background knowledge required to understand the proposed method. In Section 4, we propose the structure of WR-Store++ and processing algorithms. We also present the ideas for further improvement of WR-Store++. In Section 5, we show the experimental results by comparing WR-Store++ and the commercial key-value stores. In Section 6, we describe the discussions for the proposed WR-Store++. In Section 7, we present conclusions and future work.

2 Related Work

The key-value store is designed to store data with arbitrary types as a key-value pair by storing the entire data as the value in the form of binary. The key is the unique identifier of the data entries; the value is the actual data to be stored. The data types supported for the value can be various from structured data such as numbers and fixed-size characters to unstructured data [9]. The key-value stores provide simple three operations: (1) PUT, (2) GET, and (3) DELETE. The PUT operation takes a key-value pair as the input and stores it in the storage. If the input key already exists in the storage, it overwrites the value for the key. The GET operation takes a key and returns the corresponding value for the key from the storage. The DELETE operation takes a key and deletes the corresponding key-value pair from the storage.

Various types of studies conducted to improve the performance of the key-value store. It has been known that storages based on the Log-Structured-Merge tree (LSM tree), such as LevelDB, have performance problems that consume more resources than the original data size during the compaction, called amplification that leads to the CPU bottleneck. Consequently, its performance degrades as the data size increases [11,23]. To relieve the amplification, keys and values were separately managed [11]; the index was optimized [12]. To resolve the CPU bottleneck, the frequency of compaction, which requires heavy CPU costs, was adjusted [13,14]; the compaction process was eliminated by residing the entire index in the memory [15]; GPU was adapted to store and manage the hashtable [20]. The hashtable in GPU was further improved so as to flexibly respond to various workloads [24]; the workload was separately analyzed according to various data domains [25]. To improve the overall storage performance, the data replication, which allows for improving the data availability, was delayed [26]; a programmable network interface card was adapted to improve remote direct key-value access performance [27]; the settings such as cache and write buffer sizes were tuned depending on the underlying hardware [21].

There have been various studies to efficiently search the keys. The performance of the Bloom filter was improved [16]; a decoupled secondary index was employed [17]; an elastic bloom filter was adapted to respond flexibly to data hotness [18]. Instead of the Bloom filter, the Cuckoo filter was employed [19] to improve the query performance.

There also have been studies to improve key-value store performance in a distributed environment composed of multiple nodes. Multidimensional hashing was adapted for managing attributes other than the key [28]; remote direct memory access was employed [29]; an active distributed key-value store was proposed to specify a node where a new key-value pair is inserted [30]; the arrangement of data items to nodes was optimized according to the access frequency [31].

There have been limited research efforts for dealing with large-scale values in the key-value stores. Atikoglu et al. [25] performed an analysis on five large-scale workloads where data up to 1 MB are used as the large-scale value. However, it is still small-sized compared to the dataset we are targeting in this study. In our experiments, the size of a value reaches 914.77 MB and the total size of the dataset is 13.59 GB in a defined mixed workload. Such large-scale datasets have been mainly managed by object-based storages [32]. Recently, the object-based storages are mainly provided on cloud services such as Amazon S3, Microsoft Azure Blob Storage, and Google Cloud Storage. Studies to compare their performance have been conducted [33,34], but their detailed implementations are not disclosed. We aim to support large-scale values in the key-value stores while maintaining their advantages to support flexible data types. Therefore, we do not compare our method with the object-based storages, but with the existing representative key-value stores.

3 Background

3.1 Windows Registry

The Windows registry is a hierarchical database, which is originally designed to be used by the Windows operating system. The registry contains information such as users' profiles and application-specific information, which are referenced by the operating system and the applications during the operation[3]. Because the access to the registry occupies a large portion of the Windows programs, the access patterns to the registry of the program were analyzed and used to improve the program performance [35]. The registry consists of multiple branches according to the characteristics of the stored data. Each branch is stored in the disk as an individual file, called a hive, for persistent data storage. Tab. 1 shows the list of the hives and their description[4].

Table 1: Names of hives and their description

Hive names	Description
HKEY_CURRUNT_USER(HKCU)	Contains the configuration information for the user who is logged on. This information is associated with the user's profile.

(Continued)

[3]Microsoft Description of the registry, https://docs.microsoft.com/en-us/troubleshoot/windows-server/performance/windows-registry-advanced-users#description-of-the-registry

[4]Microsoft Inside the Registry, https://docs.microsoft.com/en-us/previous-versions//cc750583(v=technet.10)

Table 1: Continued

Hive names	Description
HKEY_USERS(HKU)	Contains all the actively loaded users' profiles on the computer
HKEY_LOCAL_MACHINE(HKLM)	Contains configuration information particular to the computer (for any users)
HKEY_CLASSES_ROOT(HKCR)	Contains information about registered applications
HKEY_CURRENT_CONFIG(HKCC)	Contains information about the hardware profiles that are used by the local computer at system startup

Fig. 1 shows the structure of the registry. As shown in the left panel of Fig. 1, we can represent the parent-child relationship using the key and subkey of the registry. Hence, the registry has a tree structure, and consequently, we can utilize it as the index for the data storage, which has not been considered before WR-Store [22]. For example, the registry key shown in Fig. 1 can be modeled as a leaf node in the index, and the path from the root node to the leaf node becomes "Computer\HKEY_CURRENT_USER\ AppEvents\EventLabel\ActivatingDocument." Hence, the path to the leaf node consists of multiple subkeys, such as AppEvents and EventLabel, and we can store multiple values in the key as shown in the right panel of Fig. 1.

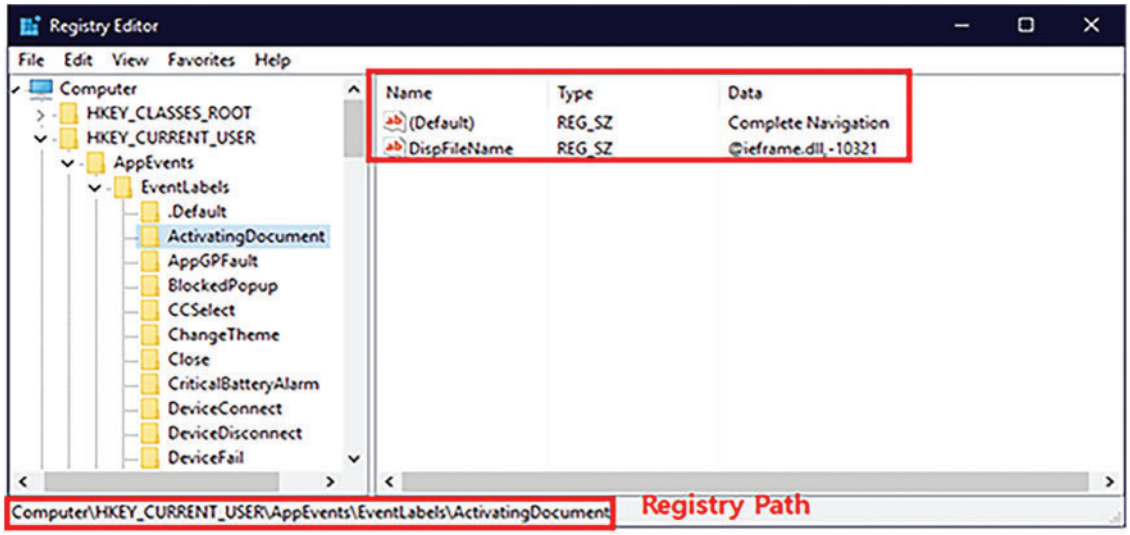

Figure 1: The structure of the windows registry

Each value in the registry consists of multiple attributes: (1) name, (2) type, and (3) data. The name attribute serves as a unique identifier to distinguish the values of the key. The type attribute provides various data types. The representative types are REG_BINARY for storing binary data, REG_SZ for storing text data, and REG_DWORD for storing 32-bit integers. The data attribute stores the actual value to be stored following the defined type. In the example of Fig. 1, the name attribute is "DispFileName"; the type attribute is REG_SZ; the data attribute is "(@ieframe.dll, -10231)".

3.2 WR-Store

WR-Store [22] was the first key-value store utilizing the Windows Registry as the storage. WR-Store directly stores the key-value pair in the Windows Registry. Hence, WR-Store stores the key of each key-value pair in the name attribute of the registry value and its value in the data attribute of the registry value. With various types of registry values, the Windows Registry can store data of variable sizes. In addition, WR-Store utilizes the hierarchical structure of the Windows Registry based on the key-subkey relationship for indexing the key-value pairs. Fig. 2 shows an example of how WR-Store stores the key-value pairs. WR-Store builds its own storage under a specific path in the registry. Here, we suppose that "Computer\HKEY_CURRENT_ USER\test" is used as the path for the storage where "test" is the key-value store name. WR-Store can build the index with multiple depths using the key-subkey relationship of the registry. Here, we assume that the depth of the index is 2. Then, for inserting a key-value pair {13, "2091534"}, WR-Store finds the leaf subkey to store the key, 13. Finally, {13, "2091534"} is stored in the found subkey, "Computer\HKEY_ CURRENT_USER\test\07\2b". Hence, 13 is stored in the name attribute of the registry value and "2091534" in the data attribute of the registry value, respectively. We note that each registry entry can store multiple key-value pairs. Hence, in the example, another key-value pair, where the key is 9752, was already stored.

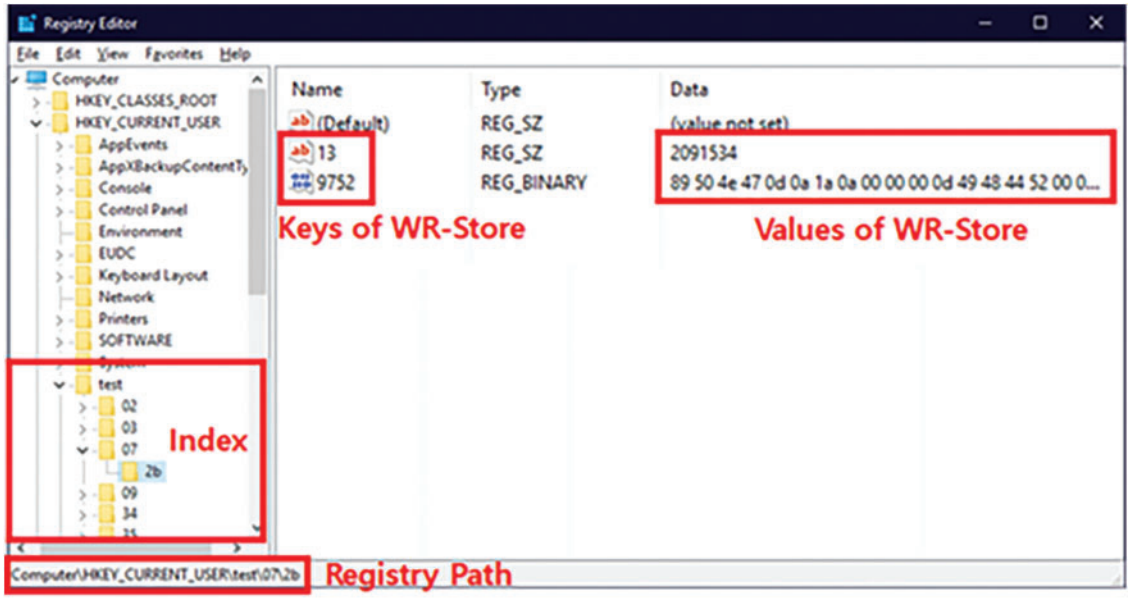

Figure 2: The structure of WR-Store

Through the comparative experiments with the representative persistent key-value stores, i.e., LevelDB, BerkeleyDB, and RocksDB, WR-Store showed comparable or even better performance. In particular, as the data set size increases, it showed better performance, indicating the potential for supporting large-scale values. However, WR-Store has an inherent limitation in storing the entire large-scale data sets in the registry because the registry is originally designed so as to store metadata in small sizes. Microsoft recommends storing values under 2 MB, limiting the application. In this study, we target the applications to store large-scale values such as images and videos. To this end, we need to resolve the limitation of WR-Store in storing the large-scale values.

4 WR-Store++: A Key-Value Store Based On Windows Registry for Storing Large-Scale Values

4.1 Basic Concept

Fig. 3 shows the structure of WR-Store++ consisting of two parts: index and data storage. WR-Store++ employs key-value separation [11,17] for supporting large values in key-value stores. Key-value separation is a technique separating the key and the value and storing only the pointer of the value along with the key in the index and actual values in a separate region. For WR-Store++, in the data storage, actual values are stored as an individual file in a specified path in the file system; in the index, the path for the data file is stored, instead of storing the value itself. As shown in Fig. 3, values stored in the data storage can be accessed through the paths for them stored in the index. In addition, we maintain metadata of the value in the index for the convenience of processing: (1) the size of the value and (2) and the stored time. Consequently, by extending the storage in the Windows Registry into that of the file system, we can utilize the data storage of WR-Store++ as large as the disk space.

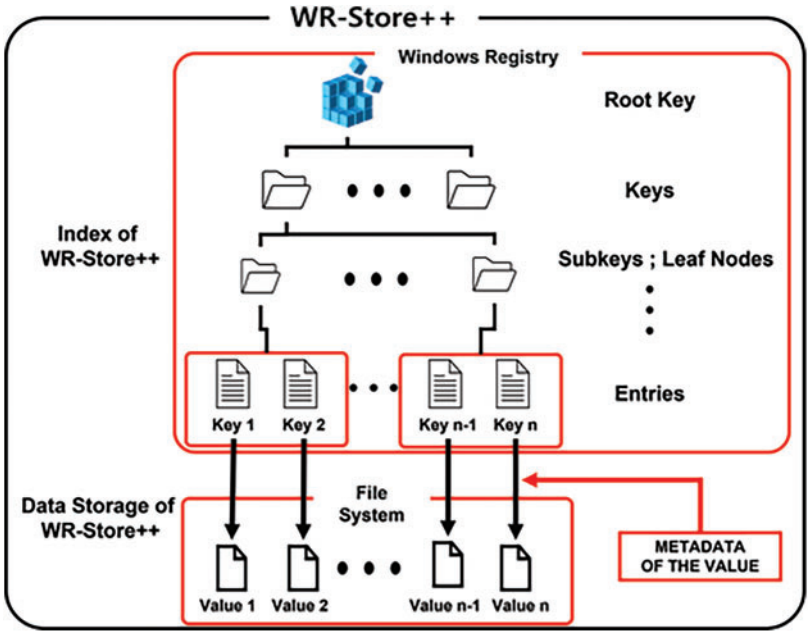

Figure 3: The structure of WR-Store++

4.2 Operations for WR-Store++

Algorithm 1 shows the algorithm of the PUT operation for WR-Store++. It takes the key-value pair as the input; it stores the input value in the data storage and stores metadata in the index. First, we obtain a path in the file system to store the value and store the input value as a new file in the data storage. Next, in the index, we search the leaf node to store the input key-value pair and create a new registry entry in the leaf node. We store the path of the value stored in the data storage and the metadata in the index. If the key to be inserted already exists in the index, we update the file in the data storage and insert a new entry for the key-value pair in the index. Hence, the existing entry for the key-value pair is not immediately removed, but just marked as invalid. These invalid entries are removed by periodical compaction of the index.

Algorithm 1: PUT operation of WR-Store++

```
Algorithm1 PUT
Input: (key_in, value) - key and value pair to store

FILE* file_out;
HKEY h_key;

 //Store the value in the data storage
if(fopen_s(&file_out, VALUE_PATH_, "wb") == NULL)
{
   return FAIL_TO_PUT;
}
if(fwrite(value, sizeof(value)/count, count, file_out) == NULL)
{
    fclose(file_out);
    return FAIL_TO_PUT;
}
 fclose(file_out);
//Create a new entry. If it already exists, open the entry
if(RegCreateKey(RootKey, DBName, &h_key) != ERROR_SUCCESS)
{
    return FAIL_TO_PUT;
}
//Store metadata in the index
if(RegSetValueEx(h_key, key_in, 0, REG_SZ, (LPBYTE)METADATAS, sizeof(METADATAS) ) != ERROR_SUCCESS)
{
    RegCloseKey(h_key);
    return FAIL_TO_PUT;
}
RegCloseKey(h_key);
```

Algorithm 2 shows the algorithm of the GET operation. It takes the key as the input and retrieves the value. First, we search the index; if the input key has the corresponding entry in the index, we refer to the absolute path indicating a file stored in the data storage. Then, we read the value from the file and copy it into the memory.

Algorithm 3 shows the algorithm of the DELETE operation. It takes the key to delete the key-value pair from the WR-Store++. First, we search the index; if the corresponding key exists in the index, we delete the corresponding value, i.e., a file storing the value, from the data storage and delete the entry from the index.

Algorithms 2 and 3: GET and DELETE operations of WR-Store++

```
Algorithm2 GET
Input: key_in – key of key-value pair to read
Output: value_out

FILE* file_out
HKEY h_key;

//Open the index
if(RegOpenKeyEx(RootKey, DBName, 0, KEY_QUERY_VALUE, &h_Key) != ERROR_SUCCESS)
{
   return FAIL_TO_GET;
}
//Retrieve the value path from the metadata in the index
if(RegQueryValueEx(h_key, key_in, NULL, NULL, VALUE_PATH, NULL) != ERROR_SUCCESS)
{
   RegCloseKey(h_key);
   return FAIL_TO_GET;
}
RegCloseKey(h_key);
//Fetch the value from the data storage
if(fopen_s(&file_out, VALUE_PATH, "rb") == NULL)
{
   return FAIL_TO_GET;
}
if(fread(value_out, sizeof(value)/count, count, file_out) == NULL)
{
    fclose(file_out);
    return FAIL_TO_GET;
}
fclose(file_out);
```

```
Algorithm3 DELETE
Input: key_in - key of key-value pair to delete

HKEY h_key;

//Open the index
if(RegOpenKeyEx(RootKey, DBName, 0, KEY_QUERY_VALUE, &h_Key) != ERROR_SUCCESS)
{
   return FAIL_TO_DELETE;
}
//Retrieve the metadata from the index
if(RegQueryValueEx(h_Key, key_in, NULL, NULL, VALUE_PATH, NULL) != ERROR_SUCCESS)
{
   return FAIL_TO_DELETE;
}
//Delete the Value from the data storage
if(access(VALUE_PATH, F_OK) != NULL && remove(VALUE_PATH) != NULL)
{
   return FAIL_TO_DELETE;
}
//Delete the metadata from the index
if(RegDeleteKeyValue(RootKey, DBName, key_in) != ERROR_SUCCESS)
{
   RegCloseKey(h_key);
    return FAIL_TO_DELETE;
}
 RegCloseKey(h_key);
```

4.3 Chunk-Based Storage Structure

In the basic version of WR-Store++, we store the entire value in a separate file. However, it incurs repeated disk input/output (IO) and random access to the file system when a large number of key-value pairs is accessed. Thus, we present chunk-based WR-Store++ where we map a file into a large-scale chunk and a value into a block in the chunk. This improves the performance of the GET operation because we can retrieve each value only by moving the file pointer to the target block in the chunk, which could already reside in the memory by the previous operations to access other blocks in the same chunk, instead of occurring a disk IO whenever the operation occurs. Fig. 4 shows chunk-based WR-Store++. Because the block size would be various, we need to store additional metadata of the value, i.e., a starting point of the block and the block size. In addition, we assign the chunk key to distinguish the chunks and manage the metadata of the chunks, i.e., the absolute path of the chunk file, the last offset of the file stream of the chunk file, and the entire key range that is covered by the chunk.

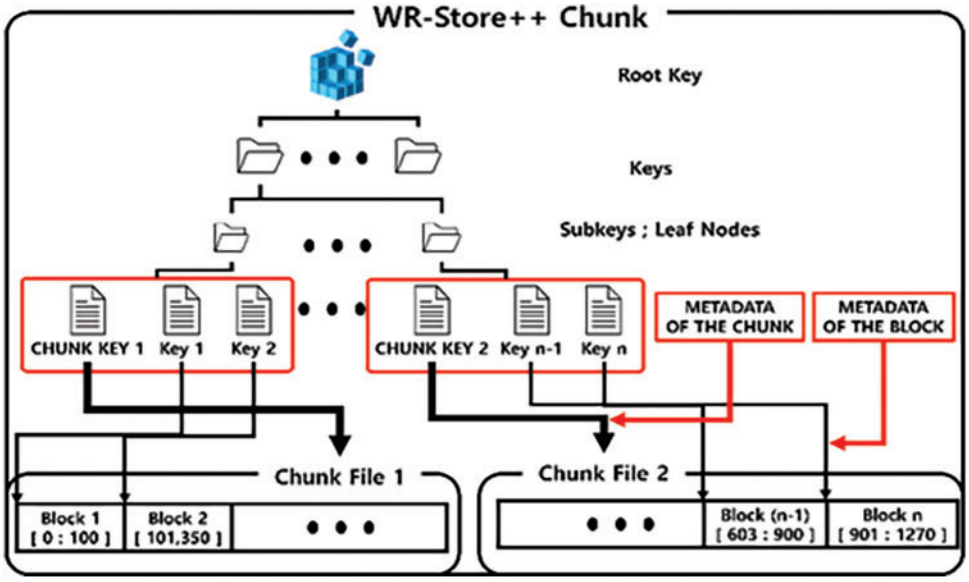

Figure 4: The structure of chunk-based WR-Store++

In the PUT operation, we retrieve the chunk file using the absolute path stored in the index. Then, we move the file pointer to the last offset of the chunk file and write the input value as a new block at the last offset of the chunk file. Finally, we update the last offset of the chunk file and the key range of the chunk. We store the metadata of the block, i.e., the starting offset in the chunk file and the block size. If the input key already exists, the old ones are marked as invalid and removed through reallocation, which is completely the same process as basic WR-Store++.

In the GET operation, if the input key exists in the index, we read the path of the chunk file and the metadata of the block, i.e., the starting offset and the block size. Then, we move the file pointer to the starting offset for the block in the chunk file and read the block as much as the block size and copy it to the memory.

In the DELETE operation, we only mark the block invalid in the index, which will be removed by the periodic reallocation. For the reallocation in chunk-based WR-Store++, we reorganize the chunk only with valid elements, excluding the others. Hence, for each chunk file, we relocate the blocks in

ascending order by the key. This process is similar to the compaction of the LSM tree, and through this, invalid blocks and the corresponding entries in the index are removed to reduce storage waste.

4.4 Parallel-GET for WR-Store++

To further improve the performance of the GET operation in WR-Store++, we design Parallel-GET for WR-Store++. For this, we divide each block into multiple sub-blocks, and then, assign each sub-block to each thread, enabling the parallel processing for the block. First, we obtain the block size and the starting offset of the block in the chunk file. Second, we calculate the starting offset and size of each sub-block considering the number of threads. Third, we move the file pointer of each thread to the starting offset of each sub-block. Then, the assigned threads read all the sub-blocks simultaneously. Without storing additional metadata in the index, we can calculate the starting offset and size for each sub-block by dividing the sub-blocks evenly.

5 Performance Evaluation

5.1 Experimental Environments and Methods

Tab. 2 shows the list of the datasets used in the experiment. All data sets are copyright-free items collected from Wikimedia[5]. To use large-scale values, the datasets are composed of images and videos, the maximum size of data is 645 MB. In order to exclude the performance variation due to caching of the operating system, 100 different datasets of the same size for each DATA ID are used. That is, for each DATA ID, we repeat the experiments to measure the processing time for PUT, GET, and DELETE operations by 100 times with random keys. Then, we remove the outliers, the top 7% and the bottom 7%, and obtain the averaged processing time for the remaining results. We use the bytes processed per millisecond (bytes/millisecond) to measure the throughput of the key-value store.

Table 2: Datasets used in the experiment for a single operation performance evaluation

DATA ID	1	2	3	4	5	6	7	8
SIZE (kb)	19	415	2,043	4,626	12,774	96,156	270,295	661,410

We fix the DELETE operation of all the key-value stores to completely erase data including both the actual values and the corresponding index entries. In the case of LevelDB and RocksDB, the values are not deleted immediately, instead, they are marked as invalid and deleted at once when executing the compaction. Therefore, for the fair performance comparison, we perform one compaction after all DELETE operations in a single experimental set, removing all the data.

For the performance evaluation of the key-value store when various types of operations perform continuously, we defined three kinds of mixed workloads according to the different portions of read and write operations: (1) read heavy, (2) write heavy, and (3) read write average. Tab. 3 shows the ratio of the size of the used datasets and the portion of operations used in each workload. Each workload consists of a total of 100 operations. Each UPDATE operation is defined as a pair of DELETE for an existing key-value pair and PUT of a new key-value pair. Initially, we load all the datasets used for the workload, and then, each workload was performed. We measured the total elapsed time taken to perform the workload.

[5]Wikimedia, https://commons.wikimedia.org

Table 3: Datasets used in the experiment for mixed workload performance evaluation

Value size	Ratio	Operation	Read heavy	Write heavy	Read write average
~100 MB	70	GET	65	20	45
~500 MB	25	PUT	20	40	30
~1 GB	5	UPDATE	15	40	25

5.2 Experimental Results

5.2.1 Comparison with WR-Store

Fig. 5 shows the experimental results of WR-Store [22] and WR-Store++ as the data size increases. We observe that WR-Store becomes significantly slow in DATA ID 6, and the operating system becomes unstable from DATA ID 7, even failing to measure the performance. On the other hand, WR-Store++ works stably even in DATA ID 8, i.e., inserting DATA ID 8 100 times in a row (i.e., datasets of 64.5 GB), showing that WR-Store++ can manage at least 32.74 times larger datasets than the existing WR-Store considering only a given dataset.

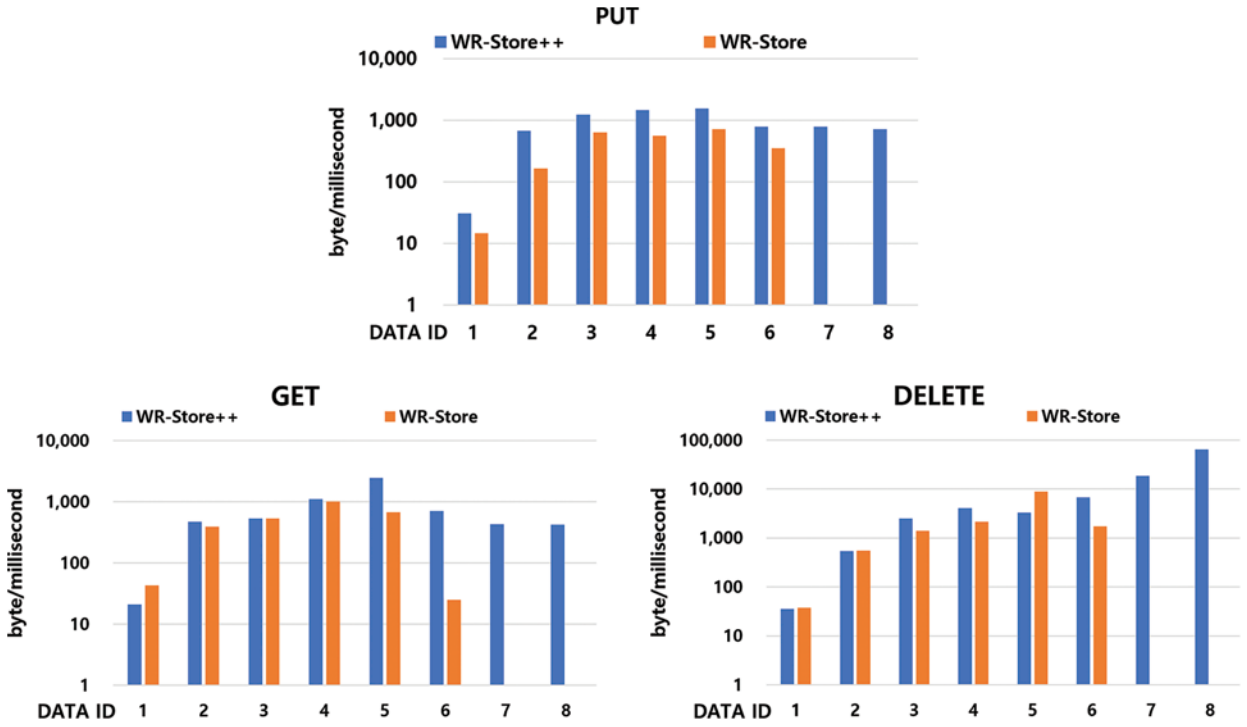

Figure 5: Performance evaluation of WR-Store and WR-Store++

When we compare the performance between WR-Store and WR-Store++ from DATA ID 1 to 6, in the PUT operation, WR-Store++ outperforms WR-Store about 1.96~4.07 times, showing the significant overhead of WR-Store in storing large-scale values. In the GET operation, WR-Store++ outperforms WR-Store more significantly as the data set increases, i.e., 28.09 times faster for DATA

ID 6. In the DELETE operation, they are comparable to each other until DATA ID 5, however, there is a clear improvement of WR-Store++ over WR-Store for DATA ID 6.

5.2.2 Comparison With the Existing Key-Value Stores

Fig. 6 shows the experimental results of WR-Store++ and the existing key-value stores as the data size increases. The results show that WR-Store++ outperforms the existing key-value store for large-scale values. In the PUT operation, WR-Store++ significantly outperforms the other storage. Specifically, it shows a performance improvement of 1.74∼44.78 times compared to BerkeleyDB, 2.53∼15.8 times to LevelDB, and 1.69∼3.72 times to RocksDB. In the GET operation, we confirm that the performance of WR-Store++ is rather degraded compared to the other storages for the datasets up to DATA ID 4, which are relatively small-sized. However, in DATA ID 5, the performance of WR-Store++ overwhelms those of the other storages, and its performance advantage is maintained until DATA ID 8. WR-Store++ shows a performance improvement of 1.16∼1.93 times compared to BerkeleyDB, 1.03∼1.91 times to LevelDB, and 1.17∼2.19 times to RocksDB from DATA ID 5 to DATA ID 8. In the DELETE operation, the performance advantage of WR-Store++ is constantly maintained for all the data sets, and in particular, the performance improvement of WR-Store++ increases as the data size increases. Specifically, WR-Store++ shows rather poor performance compared to BerkeleyDB at DATA ID 1, but outperforms BerkeleyDB in DATA ID 8 by about 2456.91 times. WR-Store++ shows a performance improvement of 1.09∼37.97 times compared to LevelDB and by 2.22∼162.9 times to RocksDB.

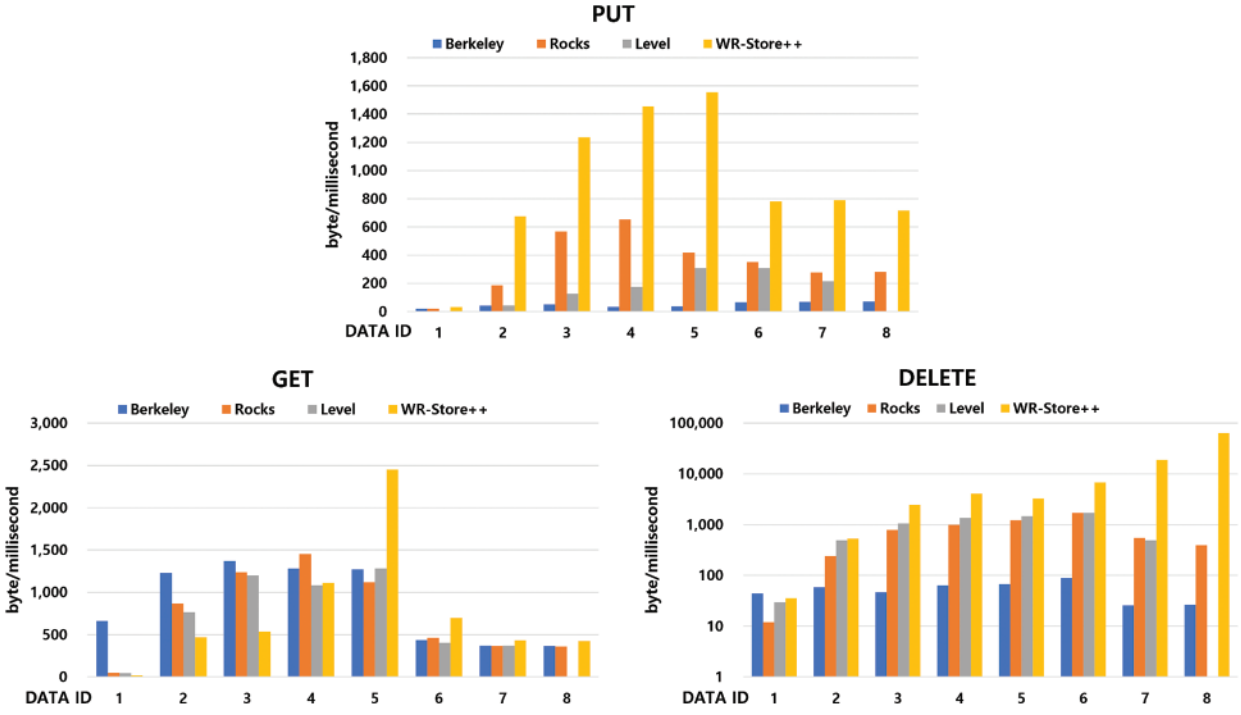

Figure 6: Performance evaluation of WR-Store++, BerkeleyDB, RocksDB, and LevelDB for individual operations

We also observe that LevelDB and RocksDB, which are based on an LSM tree that is efficient for write operations, show better performance in PUT and DELETE operations compared to BerkeleyDB

based on B+-tree. Instead, BerkeleyDB shows relatively better performance in the GET operation. We note that, in the case of LevelDB, the PUT operation does not work due to a memory error of LevelDB in DATA ID 8, and thus, all the operations on DATA ID 8 cannot be performed. This supports the results of previous studies [11] reporting that LevelDB has limitations in processing large amounts of data.

Fig. 7 shows the experimental results of WR-Store++ and the existing key-value stores on mixed workloads. We indicate that WR-Store++ clearly outperforms the existing key-value stores for all the workloads. Specifically, in the write-heavy workload, WR-Store++ outperforms BerkeleyDB by 1.69 times, RocksDB by 2.99 times, and LevelDB by 2.29 times. In the read-heavy workload, WR-Store++ outperforms BerkeleyDB by 1.74 times, RocksDB by 2.59 times, and LevelDB by 2.39 times. In the read-write average workload, WR-Store++ outperforms BerkeleyDB by 1.79 times, RocksDB by 2.64 times, and LevelDB by 2.22 times.

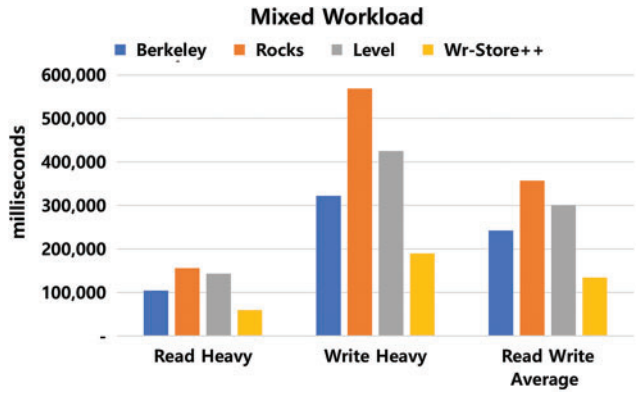

Figure 7: Performance evaluation of WR-Store++, BerkeleyDB, RocksDB, and LevelDB for mixed workloads

5.2.3 The Effectiveness of Chunk-Based and Parallel-GET WR-Store++

To validate the effects of chunk-based and Parallel-GET WR-Store++, we compare the performance of basic WR-Store++, chunk-based WR-Store++, and Parallel-GET WR-Store++ with 2 threads for the GET operation. Fig. 8 shows their experimental results. Chunk-based WR-Store++ generally outperforms basic WR-Store++ because a whole chunk is read into memory once, and then, the data is read by moving only the file pointer within the chunk. In particular, the performance of chunk-based WR-Store++ improves as the data size increases, showing that it outperforms basic WR-Store++ up to 19.67 times in DATA ID 5.

The performance of Parallel-GET WR-Store++ is rather poor when the data size is small due to the additional overhead of dividing the target block into small sub-blocks and allocating the resources to manage them. However, its performance overwhelms that of basic WR-Store++ as the data size increases. Specifically, as the data size increases, the performance of Parallel-GET WR-Store++ is improved up to 22.87 times compared to basic WR-Store++ in DATA ID 5 and up to 1.24 times compared to chunk-based WR-Store++ in DATA ID 6.

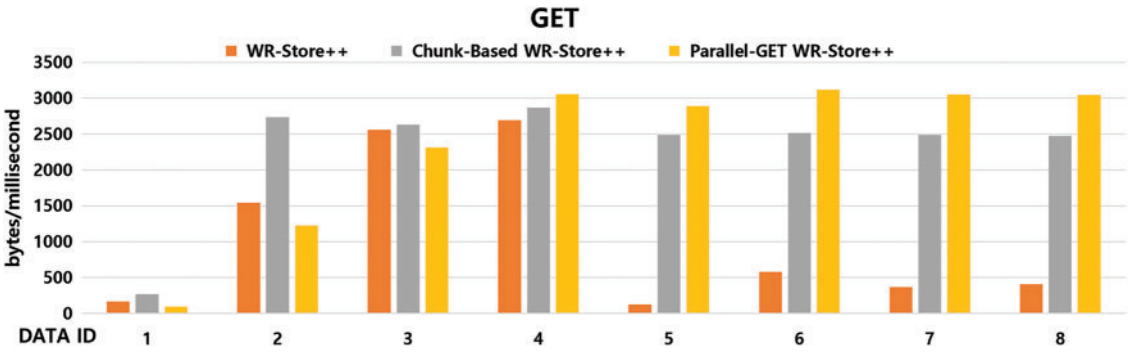

Figure 8: Performance evaluation of WR-Store++, chunk-based WR-Store++, and Parallel-GET WR-Store++ for the GET operation

Fig. 9 shows the performance of Parallel-GET WR-Store++ as the number of threads varies. The performance is affected by the following two factors: (1) overhead of allocating tasks to each thread and (2) improvement due to parallel processing of multiple threads. Regardless of the data size, the overhead for (1) is fixed, but it increases as the number of threads increases. Therefore, when the data size is small, the overhead affects much. The larger the size of the dataset is, the greater the improvement of parallel processing is. That is, in DATA ID 1, the performance becomes degraded as the number of threads increases. However, as the size of the dataset increases, a clear performance improvement is observed as the number of threads increases: in DATA ID 8, Parallel-GET WR-Store++ with 5 threads outperforms that with 2 threads up to 1.2 times, which outperforms basic WR-Store++ up to 22.26 times.

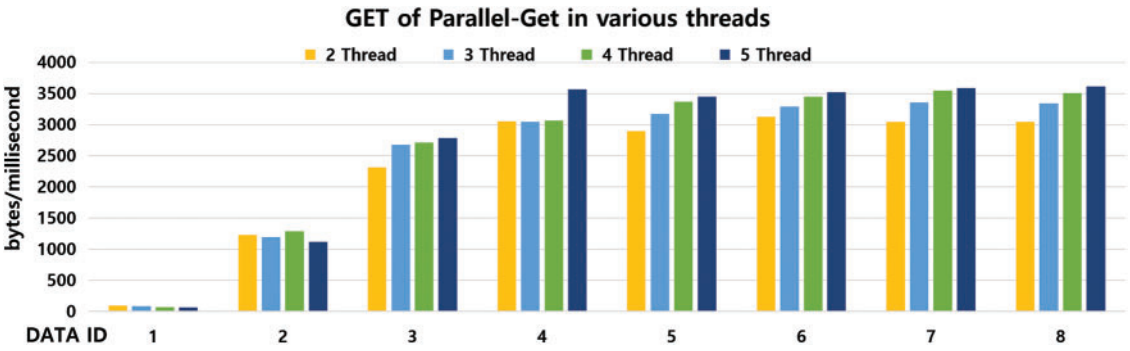

Figure 9: Performance evaluation of Parallel-GET WR-Store++ as the number of threads varies

6 Discussion

Stability of WR-Store++: WR-Store++ manipulates the Windows registry that is also accessed by the other processes. However, it only accesses a newly created path in the registry (i.e., . . . /HKEY_CURRENT_USER/DBName/ . . .), and it does not cause conflict because the other processes do not access the path.

Portability of WR-Store++: We can divide the migration into two cases considering target environments. First, we can migrate the entire datasets stored in WR-Store++ into any environments

where the Windows operating system runs, including the desktops, servers, and cloud environments. The detailed steps are as follows:

1. Step A) Export the index from a source machine. We store the index as a file with the extension of. reg, which is a file format for directly importing the exported registry in the file into another Windows registry, by using the Windows Registry API[6] and Windows CLI[7].
2. Step B) Retrieve the dataset from a source machine. We retrieve a directory in a file system in which the data storage for WR-Store++ resides.
3. Step C) Migrate the exported index to a target machine. We migrate the index to the target machine by executing the exported file in Step A) in the target machine.
4. Step D) Migrate the dataset to a target machine. We copy the entire directory retrieved by Step B) under the same file path in the target machine.

Second, we can migrate WR-Store++ to other key-value stores regardless of the running operating system, including from the local machines to cloud environments, as follows.

1. Step A) Export the index from a source machine. We perform the previous Step A) in the same way.
2. Step B) Retrieve the dataset by the key-value pair. We retrieve the key from the index by Step A) directly and the data by following the file path stored in the value of each key from the index.
3. Step C) Convert the exported index and data into key-value pairs. We construct the key and the value retrieved in Step B) to the data structure used in each key-value store: *Slice* for LevelDB and RocksDB, and *Dbt* for BerkeleyDB.
4. Step D) Migrate the key-value pairs into the other key-value stores. We insert the key-value pairs created in Step C) using the insert API of each key-value pair: *Put*(*WriteOptions*(), *key*, *value*) for LevelDB and RocksDB, and *put*(*db*, NULL, &*key*, &*value*, 0) for BerkeleyDB).

Consistency of WR-Store++: While a process accesses an entry in the Windows registry, other processes are allowed to modify the same entry. The transacted APIs[8], to open and close the registry entry, provide a strong consistency to the entry. In this study, however, we use the normal open and close APIs to the registry entry because the registry entries for WR-Store++ are only used by WR-Store++.

7 Conclusions and Future Work

In this study, we proposed a new key-value store, WR-Store++, by enhancing the existing key-value store, WR-Store, which is tightly coupled with the built-in data structure in the Windows operating system. WR-Store becomes more efficient as the data size increases, but it has a limitation to the maximum data size to be stored due to the inherent characteristics of the used built-in data structure. WR-Store++ overcomes the limitation by storing only the index part in the built-in structure and the entire data in the file system. Through the extensive experiments, we showed that WR-Store++ can dramatically extend the data storage of WR-Store and has the advantage to manage large-scale

[6]Windows Registry reg export : https://docs.microsoft.com/ko-kr/windows-server/administration/windows-comm ands/reg-export

[7]A Windows built-in command "REG EXPORT [Target Key Path] [Extracted File]" exports all the entries stored under [Target Key Path] into a file [Extracted File].

[8]Windows Registry Writing and Deleting Registry Data, https://docs.microsoft.com/en-us/windows/win32/sysinfo/w riting-and-deleting-registry-data

values for the key-value store compared to the representative key-value stores, LevelDB, RocksDB, and BerkeleyDB.

WR-Store++ was designed for a single-machine key-value store in this study. However, since there is a limitation in the amount of information that can be managed in a single machine, the necessity of extending the key-value store to a distributed environment increases. WR-Store++ has the advantages in deploying it on various computing environments because it can be run in any environment where Windows operating system runs and can be easily migrated into the other key-value stores. Accordingly, we plan to extend WR-Store++ to a distributed environment involving multiple nodes with different computing environments. First, we need to design architectures in distributed environments. We can extend WR-Store++ into distributed environments in two directions: (1) master-slave architecture and (2) serverless architecture. In the master-slave architecture, the master node manages the status of slaves and the datasets covered by each slave, and the slaves are responsible for storing and managing the actual datasets. In the serverless architecture, the managed datasets are automatically assigned to the nodes by partitioning the entire scope of datasets by a hash-based method. We evaluate the performance of those architectures using real-world workloads, and then choose the most appropriate architecture for a given workload. We expect that this architecture can be further extended to an edge computing-based federated learning framework composed of a Windows server and Windows mobile devices. Second, transacted APIs are required to guarantee strong consistency while multiple processes are running. However, in distributed environments, strong consistency will generate massive performance overheads, and they need to be compromised considering the availability. We plan to design a new sophisticated locking mechanism for the distributed environments.

Funding Statement: This study was supported by the Research Program funded by the SeoulTech(Seoul National University of Science and Technology).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] Y. Liu, J. Peng and Z. Yu, "Big data platform architecture under the background of financial technology: In the insurance industry as an example," in *Proc. of the 2018 Int. Conf. on Big Data Engineering and Technology*, Chengdu, China, pp. 31–35, 2018.

[2] A. Farseev, I. Samborskii and T. S. Chua, "Bbridge: A big data platform for social multimedia analytics," in *Proc. of the 24th ACM Int. Conf. on Multimedia*, Amsterdam, Netherlands, pp. 759–761, 2016.

[3] L. Linder, D. Vionnet, J. P. Bacher and J. Hennebert, "Big building data-A big data platform for smart buildings," *Energy Procedia*, vol. 122, no. 2, pp. 589–594, 2017.

[4] D. Geng, C. Zhang, C. Xia, X. Xia, Q. Liu *et al.,* "Big data-based improved data acquisition and storage system for designing industrial data platform," *IEEE Access*, vol. 7, no. 7, pp. 44574–44582, 2019.

[5] J. You, J. Lee and H. Y. Kwon, "A complete and fast scraping method for collecting tweets," in *2021 IEEE Int. Conf. on Big Data and Smart Computing (BigComp)*, Jeju Island, Republic of Korea, pp. 24–27, 2021.

[6] H. Shin, H. Kwon and S. Ryu, "A new text classification model based on contrastive word embedding for detecting cybersecurity intelligence in twitter," *Electronics*, vol. 9, no. 9, pp. 1–21, 2020.

[7] J. Park and H. Kwon, "Cyberattack detection model using community detection and text analysis on social media," *ICT Express*, vol. 18, no. 5, pp. 1–8, 2021.

[8] H. Kwon, "Real and synthetic data sets for benchmarking key-value stores focusing on various data types and sizes," *Data in Brief*, vol. 30, no. 18, pp. 1–10, 2020.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman *et al.,* "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.
[10] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin *et al.,* "F4: Facebook's warm {BLOB} storage system," in *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, Colorado, United States, pp. 383–398, 2014.
[11] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau and R. H. Arpachi-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.
[12] K. Ren, Q. Zheng, J. Arulraj and G. Gibson, "SlimDB: A space-efficient key-value storage engine for semi-sorted data," in *Proc. of the 8 Very Large Databases Endowment (VLDB)*, vol. 10, no. 13, pp. 2037–2048, 2017.
[13] B. Zhang and D. H. C. Du, "NVLSM: A persistent memory key-value store using log-structured merge tree with accumulative compaction," *ACM Transactions on Storage (TOS)*, vol. 17, no. 3, pp. 1–26, 2021.
[14] Y. Hu and Y. Du, "Reducing tail latency of LSM-tree based key-value store via limited compaction," in *Proc. of the 36th Annual ACM Symp. on Applied Computing*, Republic of Korea, Virtual Event, pp. 178–181, 2021.
[15] B. Lepers, O. Balmau, K. Guptaand and W. Zwaenepoel, "Kvell: The design and implementation of a fast persistent key-value store," in *Proc. of the 27th ACM Symp. on Operating Systems Principles*, Huntsville, Ontario, Canada, pp. 447–461, 2019.
[16] N. Dayan, M. Athanassoulis and S. Idreos, "Optimal bloom filters and adaptive merging for LSM-trees," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 1–48, 2018.
[17] F. Li, Y. Lu, Z. Yang and J. Shu , " SineKV Decoupled secondary indexing for LSM-based key-value stores," in *2020 IEEE 40th Int. Conf. on Distributed Computing Systems (ICDCS)*, Singapore, pp. 1112–1122, 2020.
[18] 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 *2019 USENIX Annual Technical Conf. (USENIX ATC 19)*, Renton, Washington, United States, pp. 739–752, 2019.
[19] N. Dayan and M. Twitto, "Chucky: A succinct cuckoo filter for LSM-tree," in *Proc. of the 2021 Int. Conf. on Management of Data*, China, Virtual Eventpp. 365–378, 2021
[20] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee *et al.,* "Mega-KV: A case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the Very Large Databases Endowment (VLDB)*, vol. 8, no. 11, pp. 1226–1237, 2015.
[21] Y. Jia and F. Chen, "Kill two birds with one stone: Auto-tuning RocksDB for high bandwidth and low latency," in *2020 IEEE 40th Int. Conf. on Distributed Computing Systems (ICDCS)*, Singapore, pp. 652–664, 2020.
[22] H. Y. Kwon, "Constructing a lightweight key-value store based on the windows native features," *Applied Sciences*, vol. 9, no. 18, pp. 3801, 2019.
[23] W. S. Jeong, Y. Won and W. W. Ro, "Analysis of SSD internal cache problem in a key-value store system," in *Proc. of the 2nd Int. Conf. on Software Engineering and Information Management*, Bali, Indonesia, pp. 59–62, 2019.
[24] D. Miller, J. Nelson, A. Hassan and R. Palmieri, "KVCG: A heterogeneous key-value store for skewed workloads," in *Proc. of the 14th ACM Int. Conf. on Systems and Storage*, Haifa, Israel, pp. 1–12, 2021.
[25] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of the 12th ACM Sigmetrics/Performance Joint Int. Conf. on Measurement and Modeling of Computer Systems*, London, England, United Kingdom, pp. 53–64, 2012.
[26] M. Jang, W. Kim, Y. Cho and J. Hong, "Impacts of delayed replication on the key-value store," in *Proc. of the 29th Annual ACM Symp. on Applied Computing*, Gyeongju, Republic of Korea, pp. 1757–1758, 2014.
[27] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong *et al.,* "Kv-direct: High-performance in-memory key-value store with programmable NIC," in *Proc. of the 26th Symp. on Operating Systems Principles*, Shanghai, China, pp. 137–152, 2017.

[28] R. Escriva, B. Wong and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *Proc. of the ACM SIGCOMM, 2012 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Helsinki, Finland, pp. 25–36, 2012.

[29] H. Huang and S. Ghandeharizadeh, "Nova-LSM: A distributed, component-based LSM-tree key-value store," in *Proc. of the 2021 Int. Conf. on Management of Data*, China, Virtual Event pp. 749–763, 2021.

[30] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy and H. M. Levy, "Comet: An active distributed key-value store," in *9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Vancouver, British Columbia, Canada, pp. 323–336, 2010.

[31] J. Paiva, P. Ruivo, P. Romano and L. Rodrigues, "Autoplacer: Scalable self-tuning data placement in distributed key-value stores," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 4, pp. 1–30, 2014.

[32] M. Mesnier, G. R. Ganger and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, 2003.

[33] M. Samar and H. Hajjdiab, "Comparison between Amazon S3 and google cloud drive," in *Proc. of the 2017 2nd Int. Conf. on Communication and Information Systems*, Wuhan, China, pp. 250–255, 2017.

[34] Z. Daher and H. Hajjdiab, "Cloud storage comparative analysis amazon simple storage vs. Microsoft azure blob storage," *International Journal of Machine Learning and Computing*, vol. 8, no. 1, pp. 85–89, 2018.

[35] J. Lee and H. Y. Kwon, "Redundancy analysis and elimination on access patterns of the windows applications based on I/O log data," *IEEE Access*, vol. 8, pp. 40640–40655, 2020.