applied sciences

MDPI

Article

Constructing a Lightweight Key-Value Store Based on the Windows Native Features

Hyuk-Yoon Kwon

Department of Industrial and Systems Engineering and the Research Center for Electrical and Information Technology, Seoul National University of Science and Technology, 232 Gongneung-Ro, Nowon-Gu, Seoul 01811, Korea; hyukyoon.kwon@seoultech.ac.kr

Received: 22 August 2019; Accepted: 7 September 2019; Published: 11 September 2019

Featured Application: In this paper, we propose a lightweight key-value store for managing various types of data, which are generated from Big data applications, in a very simple form. The proposed technique can be used in any environments where Windows operating systems are running, which encompass from client environments (e.g., Windows 10) to server environments (e.g., Windows Server 2016), with the minimum effort for the installation. For the other environments without Windows operating systems, we can easily migrate data to the other any environments that support existing key-value stores by using the ETL (Extract-Transform-Load) method proposed by this paper.

Abstract: In this paper, we propose a method to construct a lightweight key-value store based on the Windows native features. The main idea is providing a thin wrapper for the key-value store on top of a built-in storage in Windows, called Windows registry. First, we define a mapping of the components in the key-value store onto the components in the Windows registry. Then, we present a hash-based multi-level registry index so as to distribute the key-value data balanced and to efficiently access them. Third, we implement basic operations of the key-value store (i.e., *Get*, *Put*, and *Delete*) by manipulating the Windows registry using the Windows native APIs. We call the proposed key-value store *WR-Store*. Finally, we propose an efficient ETL (Extract-Transform-Load) method to migrate data stored in WR-Store into any other environments that support existing key-value stores. Because the performance of the Windows registry has not been studied much, we perform the empirical study to understand the characteristics of WR-Store, and then, tune the performance of WR-Store to find the best parameter setting. Through extensive experiments using synthetic and real data sets, we show that the performance of WR-Store is comparable to or even better than the state-of-the-art systems (i.e., RocksDB, BerkeleyDB, and LevelDB). Especially, we show the scalability of WR-Store. That is, WR-Store becomes much more efficient than the other key-value stores as the size of data set increases. In addition, we show that the performance of WR-Store is maintained even in the case of intensive registry workloads where 1000 processes accessing to the registry actively are concurrently running.

Keywords: databases; data stores; key-value stores; performance evaluation

1. Introduction

Due to the emergence of Big data, data having various types such as texts, locations, graphs, and images need to be stored and managed in the data store. Usually, in Big data applications, data types cannot be determined in advance, and even new data types are generated. In the relational databases, we need to define data types such as integers, dates, and characters in the database schema and store only the data that follow the defined data types. Hence, we cannot store data having various and undetermined types in the relational databases. We can manage those data easily by using

the key-value store. The key-value store represents all the data in the form of a key and value pair. Specifically, we store all the data regardless of their data types in the key-value store by mapping the entire data to the value (e.g., in the form of strings or binary) and by assigning a unique key for the value. Then, we utilize the stored data by parsing the value according to the required data types.

The representative key-value stores are Redis [1], LevelDB of Google [2], RocksDB of Facebook [3], Berkeley DB of Oracle [4], and Memcached [5]. The key-value store can be categorized into persistent stores and in-memory stores [6]. The former provides the flexibility of the data types to the relational databases. That is, we can use the key-value store to manage various and undetermined data types instead of the relational database. The latter provides the efficiency by caching data stored in persistent stores. LevelDB, RocksDB, and BerkelyDB are categorized into persistent key-value stores; Redis and Memcached into in-memory key-value stores. In-memory key-value stores can be worked with other systems for improving the performance because they can act as the cache between the applications and the underlying systems [7,8]. In this paper, we focus on the persistent stores. In-memory key-value stores can be also worked with the persistent key-value stores to improve the performance.

The key-value stores require index structures to store the data and efficiently search the stored data. LSM-tree [9] is widely used as the index structure in key-value stores, such as LevelDB [2], RocksDB [3], HBase [10], and Cassandra [11]. Much work is publised to enhance the LSM-tree [12–16]. There are some work using hash-based structures [17–21]. The other work uses other data structures such as B+-tree and B^{ε}-tree or combined structures [6,22–24].

In this paper, for designing a lightweight key-value store, we investigate on the Windows registry that is an internal built-in storage provided by the Windows operating system. In the Windows registry, we typically store configurations and meta data needed in the Windows applications and the operating system. For example, Internet Explorer stores the start Web pages in the registry. In addition, the Windows operating system maintains the programs that need to be run when the operating system starts in the registry. However, before this paper, there have been no research efforts that consider the Windows registry as a general-purpose data storage.

In this paper, we propose a method to construct a lightweight key-value store, which we call *WR-Store*, that designs the Windows registry as the key-value store. The novel ideas for WR-Store can be summarized as follows. First, we define a mapping of the components in the key-value store onto the components in the Windows registry. The Windows registry has been considered as a special-purpose data store, but we extend it as a general-purpose data store. Second, we propose a hash-based multi-level index structure. Due to the hash-based method, we can distribute the data uniformly in the structure regardless of data distribution; due to the multi-level structure, we can store large-scaled data. Third, we present the algorithms for the basic operations of the key-value store (i.e., *Get*, *Put*, and *Delete*) using the Windows native APIs. The proposed algorithms depend on a built-in internal storage in Windows, i.e., registry, and Windows native APIs. Thus, it can be well integrated with the operating systems, and consequently, be quite efficient in general. Fourth, we propose an efficient ETL (Extract-Transform-Load) method to migrate data stored in WR-Store into the other key-values stores. This allows us to extend the coverage of WR-Store not only for the environments where Windows operating systems are running but also for any other environments that support the existing key-value stores. Fifth, we conduct the empirical analysis to understand the performance of WR-Store and tune the performance. As a result, we can find the best parameter setting of WR-Store for a given data set.

Through experiments, we show that WR-Store has the following strong points:

1. **Lightness:** WR-Store is a lightweight key-value store using Windows built-in structures and native APIs without installing of additional libraries and applications. As a result, WR-Store requires quite a small-sized executable file (i.e., 26 KB). Specifically, it reduces the size of the executable file by 17.77∼153.73 times compared to the existing key-value stores (i.e., RocksDB [3], BerkeleyDB [4], and LevelDB [2]).

2. **Efficiency and scalability:** Through extensive experiments using synthetic and real data sets, we show that the performance of WR-Store is comparable or even more efficient than the state-of-the-art systems (i.e., RocksDB [3], BerkeleyDB [4], and LevelDB [2]) in terms of the throughput of the read/write/delete operations. Especially, we show the scalability of WR-Store. That is, WR-Store becomes much more efficient than the other key-value stores as the size of data set increases. In addition, we show that the performance of WR-Store is maintained even in the case of intensive registry workloads where 1000 processes accessing to the registry actively are concurrently running.

The organization of the paper is as follows. In Section 2, we describe background. In Section 3, we present the concept of WR-Store and its implementations. In Section 4, we present the empirical analysis of WR-Store and the experimental results compared to the existing key-value stores. In Section 5, we propose an efficient ETL (Extract-Transform-Load) method for WR-Store. In Section 6, we present the related work. In Section 7, we discuss the features of WR-Store. In Section 8, we conclude the paper.

2. Background

2.1. Key-Value Stores

The key-value store [25] represents each data item by a key-value pair (k, v). The key k uniquely identifies the entire data items. The value v describes the actual data item. The key-value stores provide quite simple operations: *Get*, *Put*, and *Delete* operations. The *Get* operation takes a key as the input and returns the value associated to the given key from the key-value store. The *Put* operation takes a key and value pair and stores the key-value pair into the key-value store. The *Delete* operation takes a key as the input and deletes the data item with the given key.

2.2. Windows Registry

The Windows registry is a hierarchical structure to store the system information. It includes application's configurations, devices attached to the system, and user information on the system. The Windows registry has been used in every Windows version including Windows 7/8/8.1/10 and Windows Server 2008/2012/2016. The contents in one Window version are compatible with them in the other Window version. That is, we can easily migrate data stored in the Window registry.

Figure 1 shows a logical view of the Windows registry provided by Registry Editor, which is a default registry editor on Windows. The left pane represents keys for the registry. Here, the keys are constructed as a tree structure. The tree path of the selected key in Figure 1 is "Computer\HKEY_CURRENT _USER\Control Panel\Keyboard". Each key can have multiple subkeys and values. We can make the parent-children relationship between a key and its subkeys.

The right pane shows values for each key. As shown in Figure 1, each value consists of three attributes: name, type, and data. The name attribute uniquely identifies the values in the same key; the type attribute determines the type of the value; the data attribute represents actual value following the specified value type. The representative value types are REG_BINARY that represents binary data, REG_DWORD that represents numbers, and REG_SZ that represents strings (We can refer to the Microsoft's official home page [26] for the entire set of the supported types.). This means that we can manage different types of data in the registry including numbers, strings, and binary.

The Windows registry consists of five root keys: (1) HKEY_LOCAL_MACHINE (simply, HKLM), (2) HKEY_CLASSES_ROOT (simply, HKCR), (3) HKEY _CURRENT_CONFIG (simply, HKCC), (4) HKEY_USERS (simply, HKU), and (5) HKEY_CURRENT_USER (simply, HKCU). HKLM maintains the system information; HKCR and HKCC are symbolic links to certain subkeys in HKLM; HKU maintains the information for all the users in the system; HKCU is a symbolic link to the current user information in HKU. For the security reason, we typically store the information that requires

restricted permissions (i.e., administrators) in HKLM and store the other information that can be accessed without any restrictions (i.e., any logged in users) in HKCU.

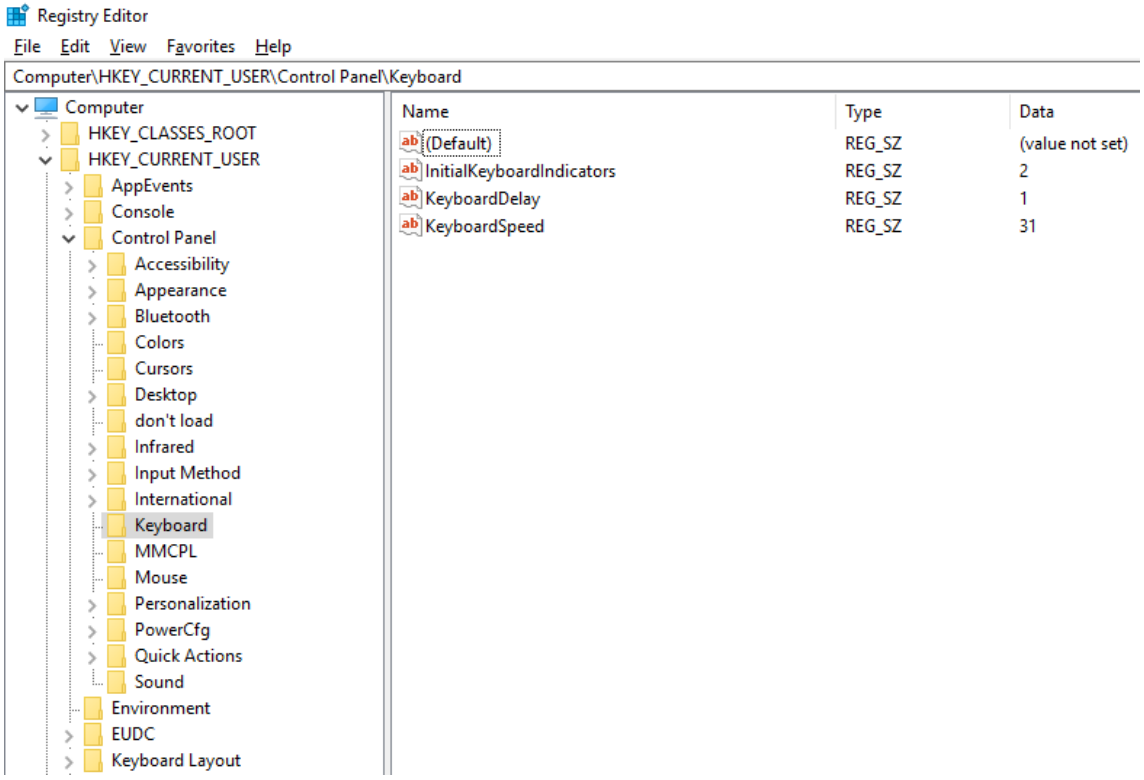


Figure 1. The structure of the Windows registry.

The Windows registry is stored in a special physical location, called Hive,for persistently storing data. The Windows operating system has a responsibility to synchronize between in-memory data and Hive.

Table 1 shows representative APIs to manipulate the Windows registry (We can refer to the Microsoft's official home page [27] for the entire the set of APIs to manipulate the Windows registry.). Basically, we create new information in the registry, query the existing information from the registry, and delete the information from the registry. These APIs are supported by every version of Windows. Thus, the executable file built using those APIs can be worked on any versions of Windows.

Table 1. Windows APIs to manipulate the Windows registry [27].

APIs	Description
RegOpenKeyEx	Open a specified registry key
RegCreateKeyEx	Create a specified registry key
RegSetValueEx	Set the data and type of a specified value in an opened registry key
RegQueryValueEx	Retrieve the data of a specified value in an opened registry key
RegDeleteKeyValue	Remove a specified value from an opened registry key
RegDeleteTree	Delete all the subkeys and values of an opened registry key recursively
RegCloseKey	Close a specified registry key

3. Constructing a Lightweight Key-Value Store Based on the Windows Native Features

3.1. The Concept

In this section, we propose a method to construct a lightweight key-value store based on the Windows native features. We call the proposed key-value store *WR-Store*. For WR-Store, we design

the Windows registry as the key-value store. Because we typically store configurations and meta data of the system and applications in the Windows registry, there have been no efforts that regard the Windows registry as a general-purpose data storage. We design WR-Store by mapping the components in the key-value store onto the components. Table 2 shows the mapping.

Table 2. Mapping of the components in the key-value store onto the components in the Windows registry.

Key-Value Store	Window Registry
Key	The name attribute of the value in the registry
Value	The data attribute of the value in the registry
Database	A registry key under a root key in the registry
Index	Registry subkeys under a registry key for the database

Figure 2 shows the concept of WR-Store. WR-Store consists of two parts: (1) storage for key-value data and (2) index structure. The right pane in Figure 2 shows the storage for key-value data in WR-Store. We can store multiple values under a registry key. In the figure, a Window registry key, "$Computer\backslash HKCU\backslash kvs\backslash fe\backslash 40\backslash 8a\backslash 96$", is storing two key-value pairs. The key of one key-value pair is "792479", and its associated value is "#Scotland"; the key of the other key-value pair is "2149", and the value is "(−115.223125, 36.232915)".

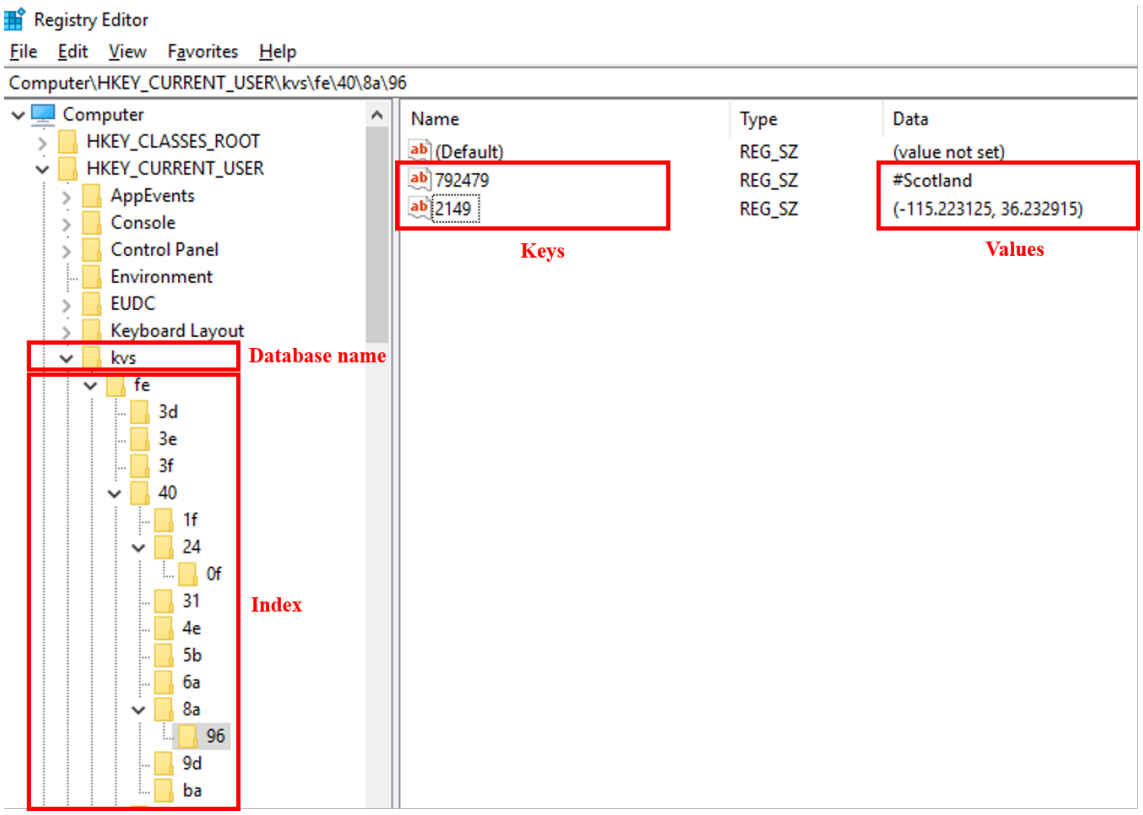


Figure 2. The concept of WR-Store.

The left pane in Figure 2 shows the index structure of WR-Store. In WR-Store, we use keys and subkeys in the Windows registry as the index structure. As a result, we store key-value data in the leaf subkeys of the registry path. For example, in Figure 2, "fe", "40", "8a", and "96" is a path to access key-value data for the keys "792479" and "2149". Because multiple values are allowed to store in a registry key, we identify actual key-value pairs by the name attribute in the value of the

registry. Here, we determine the names of keys and subkeys in the registry path from the given key. The detailed mechanism will be explained in Section 3.2.

We create a new database for WR-Store under a certain root registry. Here, we can choose a root registry key according to the desired security level for the key-value store because any logged in users can access HKCU, but only administrators can access HKLM. Consequently, we make a database in one of the following paths: (1) "Computer\HKLM\DBName" or (2) "Computer\HKCU\DBName". In Figure 2, the used root key is HKCU, and the database name is "*kvs*".

3.2. A Hash-Based Multi-Level Registry Index

In this section, we present how to construct the index structure for WR-Store. For this, we need to determine the names of keys and subkeys in the registry path. One simple way is using the key in the key-value store as it is in the Windows registry. For example, in Figure 2, "2149" becomes the key in the registry, and thus, the registry path for the key "2149" becomes "Computer\HKCU\kvs\2149". In this case, WR-Store has one-level index structure. However, to make a scalable index, the multi-level structure is required.

To make the multi-level structure, we can use substrings of the key in the key-value store as keys and subkeys in the Windows registry. For example, when we have a key "2149", we partition it into two substrings "21" and "49". Then, "21" becomes the key in the first-level registry structure, and "49" becomes the key in the second-level registry structure. As a result, a path for the registry key becomes "Computer\HKCU\kvs\21\49". However, in this case, according to the distribution of the keys, the stored key-value data could be skewed. That is, all the key-value pairs whose keys are starting with "2149" such as "214910", "214911", "214912", and "214913" will be stored under the same registry key "Computer\HKCU\kvs\21\49".

Now, we present a method to distribute key-value data uniformly over the registry keys in the multi-level structure. For this, we use a hash function to determine a registry path from a specified key in the key-value store. The hash function has a property that distributes data uniformly regardless of the distribution of the original keys. In addition, the hash collision is not the problem in WR-Store because the Windows registry allows to store multiple values in the same registry key. Here, we can identify the values in a registry key using the name attribute. In this paper, we use MD5 for a hash function and represent the hash result in hexadecimal. To make the multi-level structure, we partition the hash result into multiple substrings and use each substring as a registry key for each level of the multi-level structure.

Figure 3 shows a registry path for a hash-based multi-level registry index in WR-Store. A registry path for indexing consists of WR_D hash values where each hash value i ($1 \leq i \leq WR_D$) has a length WR_L and is used for an i-th level index in the Windows registry. We can define WR_D as the depth of the registry and WR_L as the length of each subkey in the registry path. Therefore, the total length of a registry path for indexing is $WR_L \times WR_D$.

Figure 3. A registry path for a hash-based multi-level index structure in WR-Store.

Table 3 shows the example of the registry path in WR-Store. We have a key "2149" and $MD5$("2149") = "fe40a2892ab382cd". Let us suppose that WR_D is 2, WR_L is 2, and the database name is "kvs". The first two bytes of the hash result "fe" becomes the registry key for the first level and the next two bytes of the hash result "40" becomes the registry key for the second level.

As a result, a final path for the registry key becomes "Computer\HKCU\kvs\fe\40". If WR_D and WR_L become different, the corresponding registry paths are also changed as presented in Table 3. Consequently, it will generate completely different index structures. Here, we note that the hash results of the keys "214910" and "214911" are totally different even if the original keys are similar. Therefore, the key-value data can be distributed uniformly on the Windows registry.

Table 3. The example of the registry path in WR-Store.

Key	MD5 (Key)	WR_D	WR_L	Registry Path
2149	fe40a2892ab382cd	2	2	Computer\HKCU\kvs\fe\40
2149	fe40a2892ab382cd	1	4	Computer\HKCU\kvs\fe40
2149	fe40a2892ab382cd	4	1	Computer\HKCU\kvs\f\e\4\0

The structure of WR-Store can impact the performance of WR-Store. That is, we can control WR_D and WR_L to find the best setting for WR-Store. In Section 4, we will perform the empirical analysis to check the performance variations as those factors are varied, and then, find the best parameter setting for WR-store.

3.3. Processing Algorithms for WR-Store

Figure 4 shows the processing algorithms of operations for WR-Store. Here, *RootKey* and *DBName* are determined when the database is created. *RootKey* is a root key of the registry and is determined by the security level. *DBName* represents a database name and is a subkey of *RootKey*.

In the *Get*() operation, we pass a key to find a key-value pair, *key_in*, as the input and return a value of the found key-value pair, *value_out*, as the result. First, we call *RegOpenKeyEx*() by passing *RootKey* and *DBName* as the input and obtain the handle of a specified registry key, *hKey*. Then, we call *RegQueryValueEx*() by passing *hKey* and *key_in* as the input and obtain *value_out* as the result. Here, *value_out* becomes the final result of the *Get*() operation. Finally, we call *RegCloseKey*() to finalize allocated resources.

In the *Put*() operation, we pass a key-value pair, *key_in* and *value_in*, as the input to store a new key-value pair. First, we call *RegCreateKey*() by passing *RootKey* and *DBName*. If the given *DBName* has not been created as a registry key under *RootKey* (i.e., it is the first time to insert data), then it creates a registry key of *DBName* under *RootKey*. Otherwise (i.e., *DBName* has been already created), it opens a specified registry key and returns the handle of the registry key, *hKey*. Then, we call *RegSetValueEx*() by passing *hKey*, *key_in*, REG_SZ, *value_in*, and *sizeof*(*value_in*) as the input to store a new key-value pair in the registry. Here, we use REG_SZ for the value type, which deals with strings, and consequently, we can store arbitrary data types as strings. Finally, we call *RegCloseKey*() to finalize allocated resources.

In the *Delete*() operation, we pass a key of a key-value pair to delete, *key_in*, as the input. First, we call *RegOpenKeyEx*() by passing *RootKey* and *DBName* as the input and obtain the handle of a specified registry key, *hKey*. Then, we call *RegDeleteKeyValue*() by passing *RootKey*, *DBName*, and *key_in* as the input to delete the key-value pair with a specified key. Finally, we call *RegCloseKey*() to finalize allocated resources.

```
Algorithm Get
Input: key_in: key of a key-value pair to find
Output: value_out: a value of the found key-value pair

HKEY hKey;
If (RegOpenKeyEx(RootKey, DBName, 0, KEY_QUERY_VALUE, &hKey)
   != ERROR_SUCCESS)
{
   return FAIL_TO_GET;
}
if (RegQueryValueEx(hKey, key_in, NULL, NULL, value_out, NULL)
   != ERROR_SUCCESS)
{
   RegCloseKey(hKey);
   return FAIL_TO_GET;
}
RegCloseKey(hKey);
return value_out;
```

```
Algorithm Put
Input: (key_in, value_in): a key-value pair to store

HKEY hKey;
if (RegCreateKey(RootKey, DBName, &hKey) != ERROR_SUCCESS)
{
   return FAIL_TO_PUT;
}
if (RegSetValueEx(hKey, key_in, 0, REG_SZ, (LPBYTE)value_in, sizeof(value_in))
   != ERROR_SUCCESS)
{
   RegCloseKey(hKey);
   return FAIL_TO_PUT;
}
RegCloseKey(hKey);
```

```
Algorithm Delete
Input: key_in: key of a key-value pair to delete

HKEY hKey;
if (RegOpenKeyEx(RootKey, DBName, 0, KEY_ALL_ACCESS, &hKey)
   != ERROR_SUCCESS)
{
   return FAIL_TO_DELETE;
}
if (RegDeleteKeyValue(RootKey, DBName, key_in) != ERROR_SUCCESS)
{
   RegCloseKey(hKey);
   return FAIL_TO_DELETE;
}
RegCloseKey(hKey);
```

Figure 4. Processing algorithms for WR-Store.

4. Performance Evaluation

4.1. Experimental Data and Environment

We compare the performance of WR-Store with the representative key-value stores: RocksDB of Facebook [3], BerkeleyDB of Oracle [4], and LevelDB of Google [2]. For implementing the *Get*(), *Put*(), *Delete*() operations of WR-Store, we use the algorithms presented in Section 3.3 based on the Windows native APIs. For RocksDB, we build a library for Windows according to the guideline provided by RocksDB (https://github.com/facebook/rocksdb/wiki/Building-on-Windows). We use the source code released in the master branch of Github (https://github.com/facebook/rocksdb); the used source code version is 5.14. For BerkeleyDB, we build a library for Windows according to the document provided by Oracle (https://docs.oracle.com/database/bdb181/index.html). We use the source code released in the Oracle official homepage (http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html); the used source code version is 18.1. For LevelDB, we build a library for Windows using the source code released in Github for the Windows version of LevelDB (https://github.com/ren85/leveldb-windows).

RocksDB, BerkeleyDB, and LevelDB provide interfaces for multiple programming languages such as C, C++, and Java. For the fairness, we use C++ for all the methods. We use Visual Studio 15.0 on Windows 10 64 bit. To verify that WR-Store can run on any Windows environments,

we perform experiments in the other environment on a different Windows version. We perform all the experiments on a system with a quad-core Intel Xeon E5-2660 2.0 GHz processor, 8 GB of main memory, and Windows 2016 server 64 bit. To minimize the intervention of the other processes running on Windows, we perform each experiment right after booting the system.

Table 4 shows characteristics of the data sets. We use eight data sets including synthetic and real data sets. Synthetic data sets are used to measure the scalability of the key-value stores as the size of data set is increased. Each key-value pair consists of a random string with a variable length for a value and an unique number for a key. We generate four data sets of varying sizes: (1) *KVData1*, (2) *KVData2*, (3) *KVData3*, and (4) *KVData4*.

Table 4. Characteristics of the data sets.

Data Sets	Total Number of Objects	Average of Key Size	Average of Value Size	Total Size of Data Set
KVData1	10,000	3.89	52.44	560 KB
KVData2	100,000	4.89	52.53	5.7 MB
KVData3	1,000,000	5.89	52.54	58 MB
KVData4	10,000,000	6.89	52.50	589 MB
ID-Geo	2,656,660	19.00	28.34	122 MB
ID-Hashtag	172,958	19.00	25.31	7.4 MB
ID-Tweet	1,556,300	19.00	135.54	234 MB
User-Followers	140,438	12.27	5.06	2.5 MB

We crawl four real data sets from Twitter. To crawl actual tweets, we use the Tweepy library (https://github.com/tweepy/tweepy) with a different filtering condition and post-processing for each data set. That is, all the data sets are designed to have different data types such as geo locations, texts, and integers. Table 5 shows the samples of real data sets. (1) *ID-Geo* data set consists of the tweet ID and the location information of the tweet. (2) *ID-Hashtag* data set consists of the tweet ID and the hash tags in the tweet. (3) *ID-Tweet* data set consists of the tweet ID and the tweet text. (4) *User-Followers* data set consists of the user ID and the number of followers of the user.

Table 5. The samples of real data sets.

Data Sets	Key	Value
ID-Geo	1020235969885134848	(−115.223125, 36.232915)
ID-Hashtag	1020253494912135168	#OrangeCounty #I4 #Orlando #traffic
ID-Tweet	1020315941345923074	A Chocolate man with a pretty smile
User-Followers	429997348	1016

We measure three kinds of operations to evaluate the performance of the key-value stores: (1) read operations, (2) delete operations, and (3) write operations. To measure each set of operations, we conduct the basic operations in the key-value store continuously, i.e., *Get*() for the read operations, *Delete*() for the delete operations, and *Put*() for the write operations. To check the change of the performance as the number of continuous operations k is increased, we vary k from 1 to 1 million for a data set. For a small (or large) k, initial cost of the key-value store occupies a large (or small) portion. We measure the processing time in microseconds (μs) using a Windows API $QueryPerformanceCounter()$ and convert it into the operations per second (simply, *OPS*) for the presentation.

4.2. The Size of the Executable File

Because WR-Store uses Windows native structures and APIs, it is inherently lightweight. To verify this, we measure the sizes of the executable files of the key-value stores. Table 6 shows the result. The executables of the other key-value stores contain the large size of source codes and even third-party libraries such as zlib, snappy, and lz4 to support full-featured key-value stores.

In contrast, WR-Store only uses the Windows built-in structures and native APIs. As a result, the size of executable file for WR-store is quite small comparable to them of the other key-value stores. Specifically, WR-Store reduces the size of the executable file by 153.73 times compared to RocksDB, by 72.69 times compared to BerkeleyDB, and by 17.77 times compared to LevelDB.

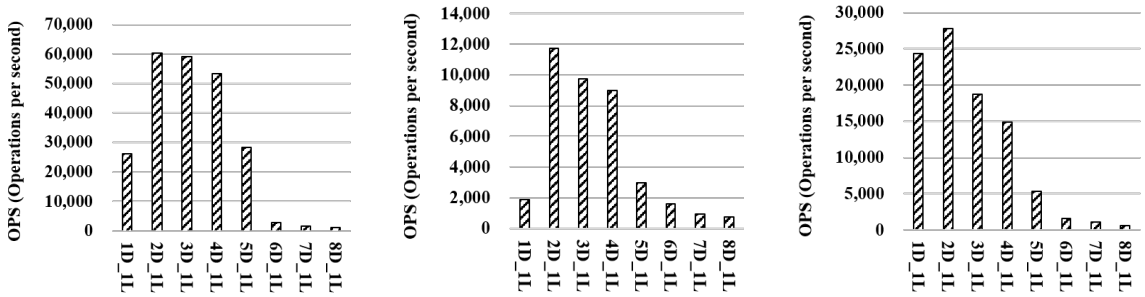
Table 6. The size of the executable file.

Key-Value Stores	Size (KB)	Ratio
WR-Store	26	1.00
RocksDB	3997	153.73
BerkeleyDB	1890	72.69
LevelDB	462	17.77

4.3. Empirical Analysis of WR-Store

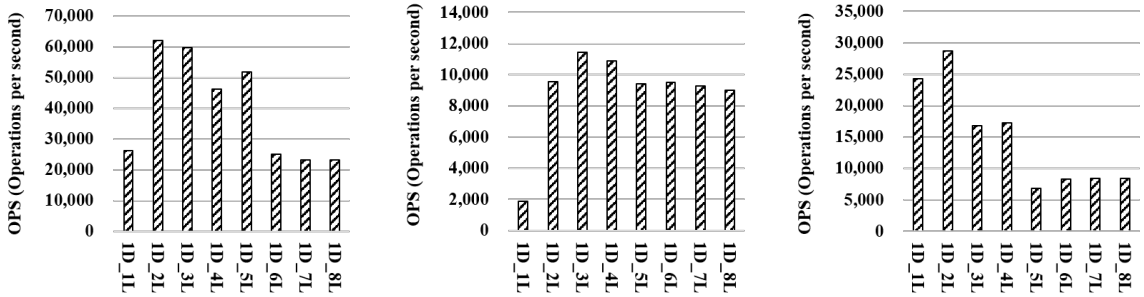
It is important to understand the characteristics of the Windows registry because the performance of WR-Store depends on the internals of the Windows registry. However, the internals of the Windows registry have not been known or reported much. In this section, we perform the empirical analysis of WR-Store to understand the characteristics of its performance. As explained in Section 3.2, we have two parameters affecting the performance of WR-Store: (1) the depth of the registry, WR_D, and (2) the length of each subkey of the registry, WR_L. We represent each setting of WR-Store as xD_yL where x is a value for WR_D and y is for WR_L. Here, we use *KVData2* as the data set and measure OPS (operations per second) while the number of continuous operations is fixed as 1 thousand.

Figure 5 shows the performance of WR-Store as WR_D is increased while WR_L is fixed as 1; Figure 6 shows the performance of WR-Store as WR_L is increased while WR_D is fixed as 1. In both experiments, when $WR_L \times WR_D$ is 2, 3, and 4, the performance of WR-Store is relatively better than the other cases. $WR_L \times WR_D$ determines the number of values under a registry key, and we conclude that it is more important factor to the performance of WR-Store than each of WR_L and WR_D. Therefore, we need to find the most efficient value for $WR_L \times WR_D$, and then, we need to select proper values for WR_L and WR_D for a determined value for $WR_L \times WR_D$.



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 5. Performance of WR-Store as WR_D is varied. (**a**); Read throughput (**b**); Write throughput; (**c**) Delete throughput.



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 6. Performance of WR-Store as WR_L is varied. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.

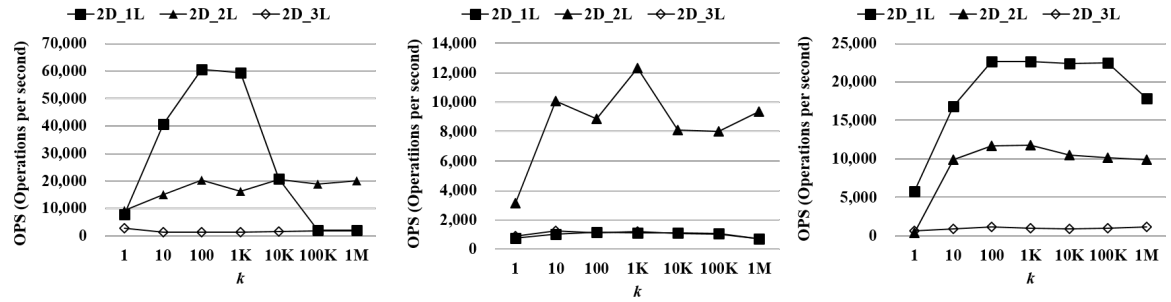
4.4. Performance Tuning of WR-Store

The empirical analysis indicates that WR-Store has the best parameter setting for a given data set.

Performance of WR-Store as $WR_D \times WR_L$ is varied: Figure 7 shows the processing time of WR-Store where $WR_D \times WR_L$ is 2, 4, and 6 as the number of continuous operations k is varied from 1 to 1 million. We use *KVData* as the data set. As $WR_D \times WR_L$ is increased, each key may have less values because the number of distinct registry keys is increased. Hence, when $WR_D \times WR_L$ is small, we need to process a large number of values for the same key, incurring the cost to deal with many values after finding the specified registry. In contrast, when $WR_D \times WR_L$ is large, we have only a small number of values for the same key (e.g., one value for each key), and consequently, have many subkeys. It requires a large size of the registry index, incurring the cost to access the large registry index. As a result, we can expect that there exists a moderate value for $WR_D \times WR_L$ that shows the best performance of WR-Store.

For the read operations, the performance of 2D_2L is the most stable as k increases. Especially, 2D_2L is much more efficient than 2D_1L by 10.41 times and 2D_3L by 11.39 times when k is 1 million. For the write operations, 2D_2L is the most efficient. Specifically, 2D_2L outperforms 2D_1L by 4.12∼12.77 times and 2D_3L by 3.51∼12.85 times as k is varied. For the delete operations, 2D_1L is generally the most efficient, but 2D_2L shows the most stable performance. Specifically, when k is varied from 100 thousand to 1 million, the performance degradation of 2D_1L is about 21.47% while that of 2D_2L is only about 2.25%.

According to the experimental results, we observe that the performance variation of WR-Store is very severe as $WR_D \times WR_L$ is varied. In addition, we show that 2D_2L is the most stable and efficient performance.

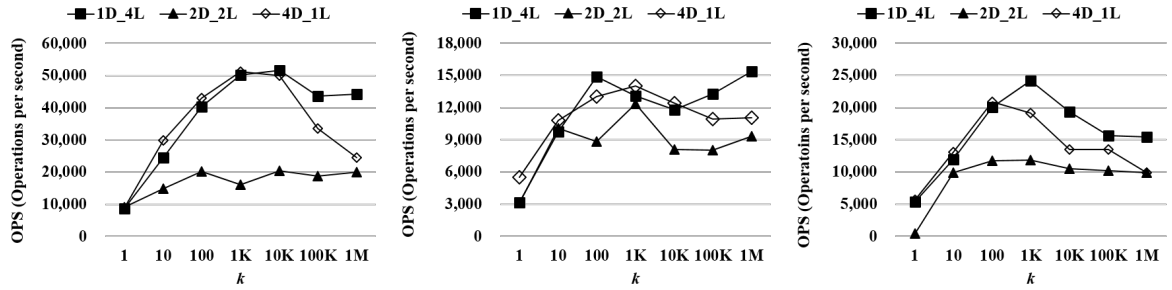


(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 7. Performance of WR-Store as $WR_D \times WR_L$ is varied. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.

Performance of WR-Store as WR_L and WR_D are varied: According to the previous result, the total length of a registry path for indexing, $WR_L \times WR_D$, is fixed as 4. Then, we vary WR_L and WR_D to find the best combination of WR_L and WR_D for a determined $WR_L \times WR_D$. That is, we have three combinations of WR_L and WR_D: (1) $1D\_4L$, (2) $2D\_2L$, and (3) $4D\_1L$.

Figure 8 shows the processing time of $1D\_4L$, $2D\_2L$, and $4D\_1L$ as the number of continuous operations k is varied from 1 to 1 million. We use *KVData* as the data set. Even if the overall performance difference is less than the case where $WR_D \times WR_L$ is varied, there exists meaningful performance difference as WR_L and WR_D are varied. For the read operations, $1D\_4L$ is the most efficient. When k is 1 million, $1D\_4L$ outperforms $2D\_2L$ by 2.21 times and $4D\_1L$ 1.81 times. For the write operations, $1D\_4L$ is slightly more efficient than the other settings. When k is 1 million, $1D\_4L$ outperforms $2D\_2L$ by 1.64 times and $4D\_1L$ 1.39 times. For the delete operations, $1D\_4L$ is the most efficient. When k is 1 million, $1D\_4L$ outperforms $2D\_2L$ by 1.55 times and $4D\_1L$ outperforms by 1.55 times.



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 8. Performance of WR-Store as WR_D and WR_L are varied. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.

According to both experimental results, we confirm that the performance variation of WR-Store is very serious as WR_L and WR_D are varied (especially for $WR_D \times WR_L$). Thus, for the remaining experiments on synthetic data sets, we use the best parameter setting for WR-Store, i.e., $1D\_4L$.

In addition, we conduct the performance evaluation on four real data sets to find their best parameter settings. Table 7 shows the measured throughput on real data sets as WR_L and WR_D are varied. Here, we simply calculate the average throughput by assigning the same weight into read, write, and delete operations. As a result, we can find the best parameter setting, which is represented in bold. For the remaining experiments on real data sets, we use the best parameter setting for each data set as shown in this table.

Table 7. The best settings of WR-Store for real data sets.

Data Sets	Throughput (OPS)	Parameters				
		2D_1L	**2D_2L**	**2D_3L**	**1D_4L**	**4D_1L**
ID-Geo	Read	23,085	54,749	4928	20,095	8791
	Write	368	8516	2045	6140	3903
	Delete	13,057	18,289	2558	11,116	4265
	Average	12,170	**27,185**	3177	12,450	5653
ID-HashTag	Read	25,635	58,849	11,411	40,369	52,908
	Write	9718	20,107	18,081	25,413	24,374
	Delete	38,835	30,053	6077	18,783	23,022
	Average	24,729	**36,337**	11,856	28,188	33,435

Table 7. *Cont.*

Data Sets	Throughput (OPS)	Parameters				
		2D_1L	**2D_2L**	**2D_3L**	**1D_4L**	**4D_1L**
ID-Tweet	Read	10,227	12,334	906	33,327	14,347
	Write	564	5578	727	9279	6889
	Delete	24,539	8079	611	13,254	9125
	Average	11,777	8664	748	**18,620**	10,121
Users-Followers	Read	39,234	61,177	48,506	46,310	49,974
	Write	17,482	21,385	19,187	24,490	24,039
	Delete	36,308	25,321	22,623	21,921	23,972
	Average	31,008	**35,961**	30,105	30,907	32,661

4.5. Experimental Results on Synthetic Data Sets

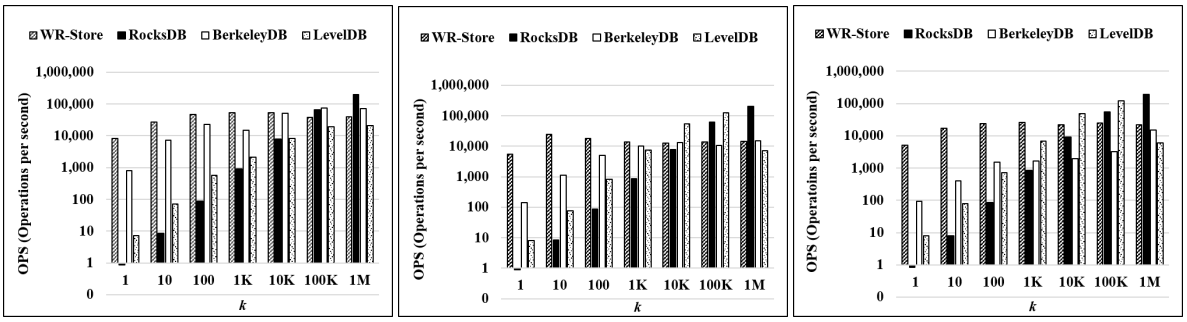
Throughput as the number of continuous operations k is increased: Figure 9 shows the throughput on a synthetic data set, *KVData3*, as k is increased from 1 to 1 million. The result indicates that WR-Store shows the constant performance regardless of k, but RocksDB, BerkeleyDB, and LevelDB do not. As a result, WR-Store is particularly efficient for small k, i.e., from 1 to 1 thousand. Specifically, WR-Store outperforms RocksDB by 59.25∼9479.95 times for read operations, by 1.59∼6182.61 times for write operations, and by 2.36∼5978.49 times for delete operations; WR-Store outperforms BerkeleyDB by 2.09∼10.24 times for read operations, by 1.32∼38.72 times for write operations, and by 15.28∼54.14 times for delete operations; WR-Store outperforms LevelDB 25.29∼1091.79 times for read operations, by 1.78∼695.29 times for write operations, and by 3.76∼642.65 times for delete operations. This stems from the fact that the other key-value stores require much initial cost to perform operations. As a result, when k is small, the performance is quite inefficient compared to WR-Store, which does not require the initial cost because the Windows registry is always ready.

We can observe some part of cases where the other key-value store are more efficient than WR-Store when the value of k becomes large. Specifically, RocksDB outperforms WR-Store by 5.03 times for read operations, by 14.02 times for write operations, and by 9.10 times for delete operations when k is 1 million. LevelDB partly outperforms WR-Store even for the large value of k. That is, LevelDB is more efficient than WR-Store by 9.16 times for write operations and by 4.89 times for delete operations when k is 100 thousand. However, for read operations, WR-Store consistently outperforms LevelDB in all the values of k; for the write and delete operations, WR-Store outperforms LevelDB again when the value of k becomes larger, i.e., 1 million. WR-Store is still comparable to BerkeleyDB even for large k. That is, WR-Store is efficient than BerkeleyDB by 44.34% for the delete operations when k is 1 million; BerkeleyDB is efficient than WR-Store by 83.09% for the read operations and by 5.31% for the delete operations when k is 1 million.

In addition, it is important to check the performance of additional loading of data in WR-Store when data has been stored. Here, a dataset, $KVData3$, has been stored, and then, we measure the performance for additional writes of data whose size is increased from 1 to 1 million. The result shows that the performance of WR-Store is quite constant even as the size of newly inserted data is increased, which means that WR-Store is stable even on additional loading of data.

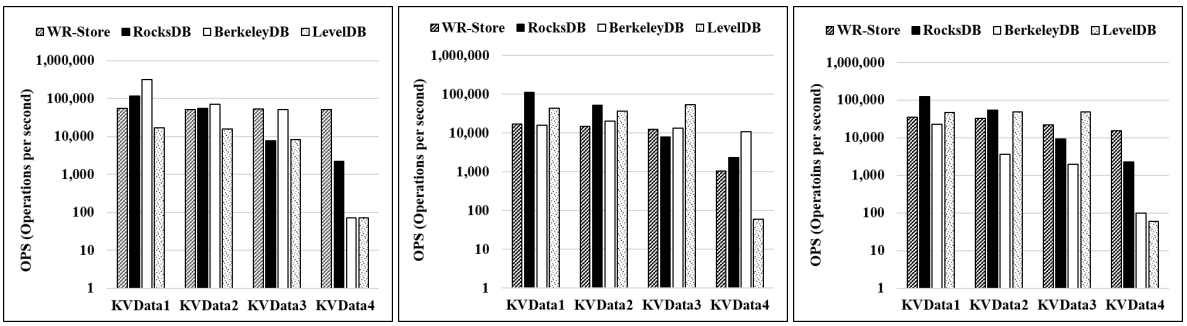
Throughput as the size of data set is increased: Figure 10 shows the throughput on synthetic data sets as the size of data set is increased from 10 thousand (i.e., $KVData1$) to 10 million (i.e., $KVData4$). We fix the number of continuous operations as 10 thousand. The most important point of this result shows the scalability of WR-Store. That is, the performance of WR-Store is quite constant as the size of data set is increased while the performances of the other key-value stores are degraded much. If we measure the performance degradation when the data set is $KVData4$ compared to $KVData1$, that of WR-Store is 1.05 times for the read operations, 2.30 times for the delete operations, and 16.25 times for the write operations, respectively; RocksDB 52.11 times, 55.02 times, and 49.63 times, respectively;

BerkeleyDB 4365.17 times, 230.23 times, and 1.48 times, respectively; LevelDB 239.80 times, 725.28 times, and 766.18 times, respectively. Here, we note that WR-Store is not enough scalable to the write operations compared to the read and delete operations. This stems from the fact that the Windows registry already stores a data set required by the operating systems while the other key-value stores do not. We also note that BerkeleyDB is scalable to the write operations, but its performance degradation is much severe for the read and delete operations.



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 9. Throughput on a synthetic data set as the number of continuous operations k is increased. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.



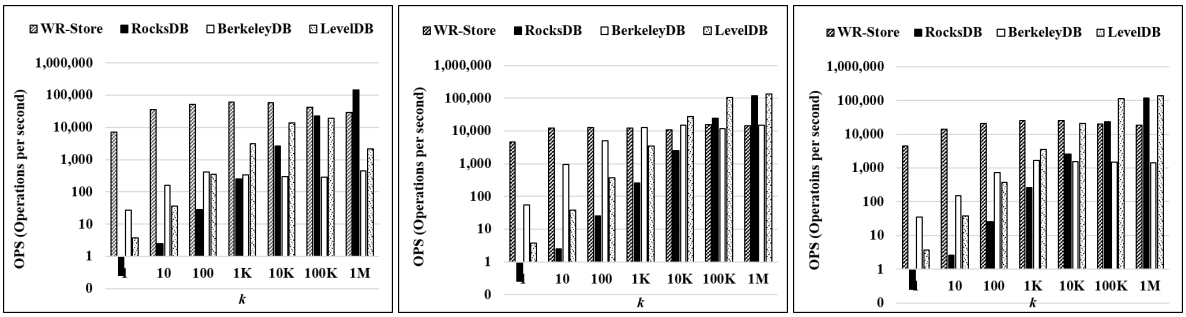
(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 10. Throughput as the size of data set is increased. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.

4.6. Experimental Results on Real Data Sets

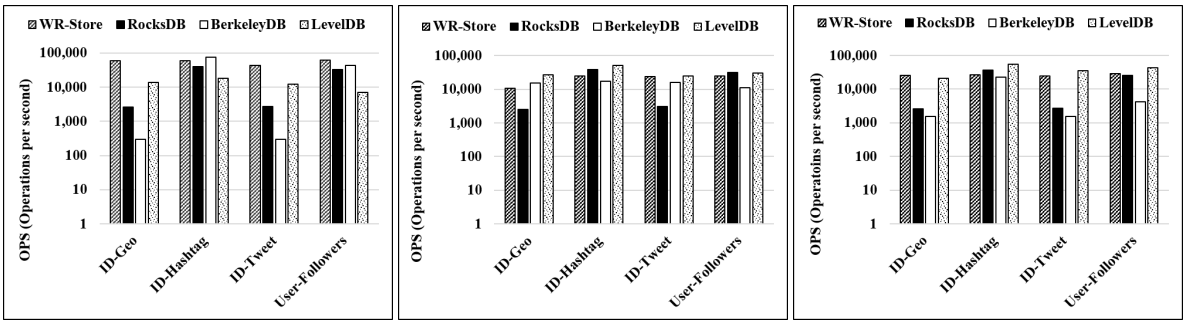
Throughput as the number of continuous operations k is increased: Figure 11 shows the throughput on a real data set as k is increased from 1 to 1 million. Here, we use *ID-Geo* as the data set. Overall trends are similar to the results on synthetic data sets as presented in Figure 9. That is, the performance of WR-Store is quite constant while the other key-value stores require much initial cost. Therefore, WR-Store is much efficient for small k while RocksDB and LevelDB outperform WR-Store for only some values of large k.

Throughput for various real data sets: Figure 12 shows the throughput on various real data sets. We fix the number of continuous operations as 10 thousand. For the read and delete operations, we note that WR-Store outperforms the other key-value stores for the larger data sets, i.e., *ID-Geo* and *ID-Tweet*. For the write operations, BerkeleyDB and LevelDB show quite constant performance while WR-Store is still comparable to them.



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 11. Throughput on a real data set as the number of continuous operations k is increased. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 12. Throughput on various real data sets. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.

4.7. The Stress Test in Intensive Registry Workloads

The Windows applications are always accessing to the registry to store and retrieve data. It is important to observe the performance of WR-Store in intensive registry workloads because WR-Store is based on the registry. First, we analyse the registry accesses of other processes while WR-Store is working. We collect all the registry operations occurred in the Windows operating system using Process Monitor (https://docs.microsoft.com/en-us/sysinternals/downloads/procmon), which is a tool to monitor all the I/O events in Windows including the registry accesses, provided by Microsoft. Table 8 shows the summarization of registry operations collected by Process Monitor. That is, over 3.8 million registry operations are collected during 626.7 s. We confirm that 79 Windows processes are working with WR-Store and they are actively accessing to the registry. That is, the average time interval between registry operations in all the processes is 12.96 milliseconds for each process. Table 9 shows top-10 processes accessing to the registry in the order of the portion. The average time interval between registry operations in top-10 processes is 1.95 milliseconds.

Table 8. The summarization of the collected registry operations.

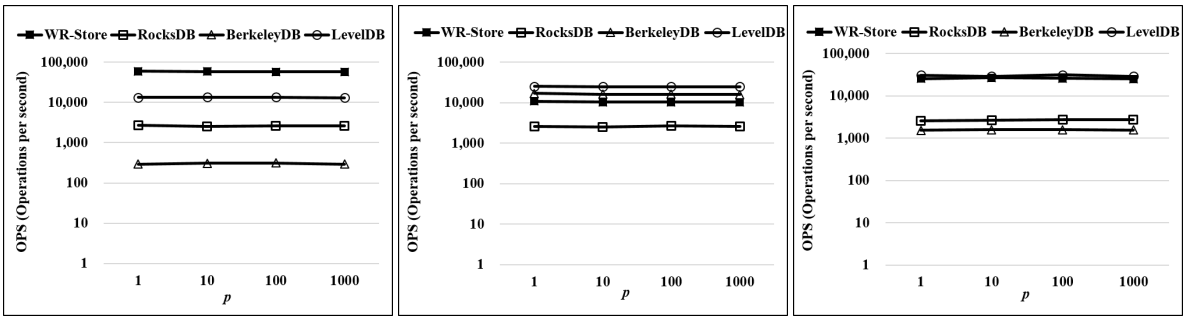
Total Duration for Collection	The Number of Registry Operations	The Number of Processes	The Average Time Interval between Registry Operations in Each Process
626.7 s	3,821,733	79	12.96 milliseconds

Then, we measure the performance of WR-Store in the environments of the intensive registry workloads. Here, we make each synthetic process so as to access to the registry once every 1.95 milliseconds, which is the same as the average frequency of top-10 processes accessing to the registry. Then, we vary the number of concurrently running processes p from 1 to 1000. Figure 13

shows the performance variation as p is increased. Here, we use a real data set, *ID-Geo*, as the data set and fix the number of continuous operations as 10 thousand. As a result, even in the case where 1000 processes accessing to the registry intensively are concurrently running, the performance of WR-Store is maintained constantly like as the other key-value stores that do not depend on the registry.

Table 9. The top-10 processes accessing to the registry.

Process Name	The Number of Registry Operations	Portion
svchost.exe	932,191	24.39%
settingsynchost.exe	667,275	17.46%
mscorsvw.exe	365,202	9.56%
tiworker.exe	275,724	7.21%
sihost.exe	205,835	5.39%
runtimebroker.exe	180,807	4.73%
explorer.exe	177,815	4.65%
ngen.exe	147,814	3.87%
msmpeng.exe	130,601	3.42%
backgroundtaskhost.exe	125,047	3.27%



(**a**) Read throughput. (**b**) Write throughput. (**c**) Delete throughput.

Figure 13. Throughput as the number of concurrently running processes p increases. (**a**) Read throughput; (**b**) Write throughput; (**c**) Delete throughput.

5. An Efficient Method for ETL (Extract-Transform-Load) in WR-Store

In this section, we propose an ETL (Extract-Transform-Load) method to extract data stored in WR-Store and load them into the other key-value stores. Because WR-Store is based on the Windows registry, it is portable only if Windows operating systems are running. However, in the other environments without Windows operating systems, we need a method to migrate data in WR-Store into the other environments.

The key-value store deals with data in a very simple form consisting of the key and value pair. However, WR-Store stores data in a multi-level structure based on the Windows registry as presented in Section 3.2. Thus, we need to investigate the method to extract data from the multi-level structure and to transform them into a form of the key-value pair.

First, let us consider a straightforward method for the ETL process in WR-Store. It extracts data from the registry path for WR-Store into a file, and then, transforms the contents in the file into a form of the key-value pair, which we call *extract-then-transform (E-then-T)*. The E-then-T method depends on the built-in extract feature provided by the Windows registry. That is, if we execute a Window command "regedit /E", it will extract data stored in a specific registry path into a file. Figure 14 shows the details of data extracted from WR-Store storing a real data set, *ID-HashTag*. It represents the registry path in a multi-level, and each registry path contains the corresponding key and value pairs. Then, from this file, we can specify the key part and value part by parsing the contents based on the pattern matching method such as the regular expression.

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\kvs]

[HKEY_CURRENT_USER\kvs\00]

[HKEY_CURRENT_USER\kvs\00\00]
"10202218890782842 93"="#leadership"
"10202218050076016 64"="#prasadam #Lord #prasadam #temples #Vrindavan"
"10202142317808066 56"="#8YearsofOneDirection"

[HKEY_CURRENT_USER\kvs\00\01]
"10202489606808084 48"="#longoverdue"
"10202351156691476 48"="#nba"

[HKEY_CURRENT_USER\kvs\00\02]
"10202512771655843 86"="#APDemandsJustice #NoConfidenceMotion"

[HKEY_CURRENT_USER\kvs\00\03]
"10202220691733422 08"="#TXGX"
```

Figure 14. The details of data extracted from WR-Store (Data set: *ID-HashTag*).

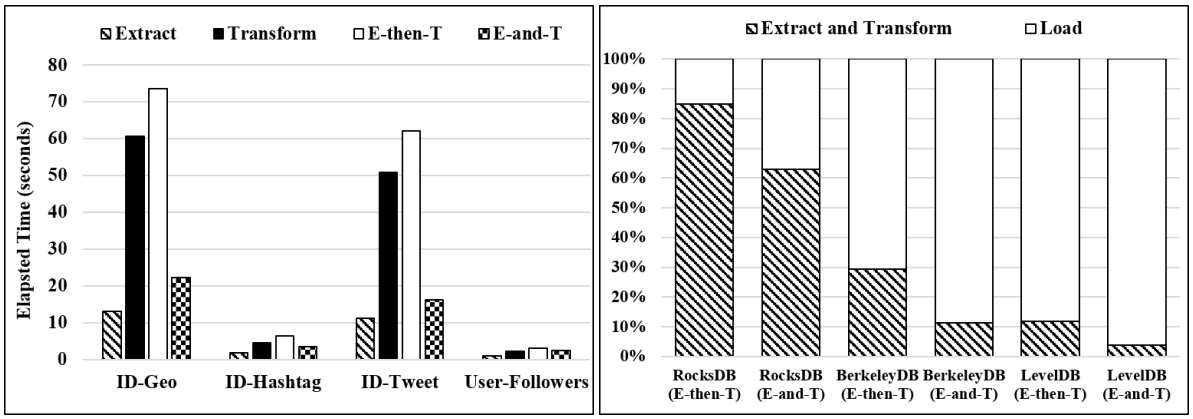
The E-then-T method requires two separate phases, and the second phase (i.e., transform) must be performed on the result of the first phase (i.e., extract). Thus, we need to design a more efficient ETL method that transforms data directly while extracting data, which we call *extract-and-transform (E-and-T)*. For the E-and-T method, we need to directly access to the Windows registry using the native APIs. Using those APIs, we can traverse the entire structure under a registry path that is storing data for WR-Store. Whenever we find the key and value during the traversal, we directly transform each one into a form of the key-value pair. Figure 15 shows the algorithm of the E-and-T method. First, it obtains the information for a given registry path to know the number of subkeys and the number of key and value pairs. If there are subkeys, we need to investigate each one of them by calling this algorithm recursively. Otherwise, we extract key and value pairs one by one. As a result, we can obtain the list of key and value pairs that have been stored in WR-Store.

To show the performance of the proposed E-and-T method, we conduct some experiments. Figure 16 shows the performance analysis of two proposed ETL methods for WR-Store, i.e., E-then-T and E-and-T. Here, we use four real data sets and fix the number of continuous operations as 10 thousand. The elapsed time of the E-then-T method consists of two parts: (1) that of the extract process and (2) that of the transform process. We measure those parts separately to compare each one with the elapsed time of the E-and-T method. Figure 16a shows the elapsed time of extract, transform, E-then-T, and E-and-T. The result shows that E-and-T is much more efficient than E-then-T. Specifically, E-and-T outperforms E-then-T by 1.24 $\sim$ 3.85 times as the data set varies. We indicate that E-and-T needs a slightly more time than the export-only process while E-then-T requires much time for the transform process, which involves the pattern matching. We note that the performance improvement of E-and-T over E-then-T becomes larger for large-scaled data sets, i.e., *ID-GEO* and *ID-Tweet*. The performance improvement is 3.30 times and 3.85 times, respectively.

Figure 16b shows the portion of E-and-T and E-then-T out of the entire ETL process that starts to extract data stored in WR-Store and ends to load them into the other key-value store, i.e., one of RocksDB, BerkeleyDB, and LevelDB. Here, we show a case where the data set is *ID-Geo*. As presented, E-then-T occupies 11.78%$\sim$84.93% out of the entire ETL process; E-and-T occupies 3.89%$\sim$63.04%. We note that E-and-T for BerkeleyDB and LevelDB occupies only 11.21% and 3.89% out of the entire ETL (extract-transform-load) process, respectively.

```
Algorithm Extract-And-Transform
Input:
   rootKey: root key to store data for WR-Store (i.e., HKCU or HKLM)
   subKey: key of a key-value pair to export under rootKey
Output: list of key-value pairs

// Get the number of subkeys and the number of values for subKey
HKEY hKey;

RegOpenKeyEx(RootKey, subKey, 0, KEY_QUERY_VALUE, &hKey);
RegQueryInfoKey(hKey, &nSubKeys, &nValues);

// If there are subkeys, we need to investigate each one of them recursively
if (nSubKeys)
{
   for (i = 0; i < nSubKeys; i++)
   {
      RegEnumKeyEx(hKey, i, subKeyName);
      newSubKey = subKey + "\\" + subKeyName;
      Extract-And-Transform(RootKey, newSubKey);
   }
}

// If there are no more subkeys, we try to extract key and value pairs from subKey

if (nValues)
{
   for (i = 0; i < nValues; i++)
   {
      RegEnumValue (hKey, i, keyName, valueName);
      write(keyName, valueName);
   }
}
RegCloseKey(hKey);
```

Figure 15. The algorithm of Extract-and-Transform (E-and-T) for WR-Store.



(**a**) Performance of E-then-T and E-and-T. (**b**) Portion of E-then-T and E-and-T.

Figure 16. The performance analysis of the proposed ETL methods for WR-Store. (**a**) Performance of E-then-T and E-and-T; (**b**) Portion of E-then-T and E-and-T.

6. Related Work

WR-Store uses the Window's internal structure based on the Windows registry as its index structure. To place the data in the index, we have proposed a hash-based structure as presented in Section 3.2. The distinguishing property of WR-Store from the existing key-values stores is that WR-Store does not require additional data structures, which are necessary in the existing key-value stores, because WR-Store directly utilizes the internal structure that is provided by the Windows operating system.

LSM-tree [9] is widely used as the index structure in key-value stores, e.g., LevelDB [2], RocksDB [3], HBase [10], and Cassandra [11]. There have been much work for enhancing the LSM-tree. bLSM [13] is a LSM-tree with the advantages of B-Trees and log structured approaches by using Bloom filters to improve the index performance. VT-tree [14] extends the LSM-tree to efficiently handle

sequential and file-system workloads. LSM-Trie [15] is a trie structure of LSM-Tree for a more efficient compaction of the structure. WiscKey [12] is a persistent LSM-tree-based key-value store optimized in the SSD storage. Flame DB [16] proposes a grouped level structure to reduce the compaction overhead in the LSM-tree. In addition, it adapts an enhanced bloom filter that reduces the false positive of a bloom filter.

There are various key-value stores based on the hash table. FAWN-KV [21] uses an in-memory hash table and a log structured datastore. It provides not only high performance on the flash storage but also replication and consistency. FlashStore [17] uses a variant of the cuckoo hash table and compacts key signatures to store the key-value pairs on the flash storage effectively. SkimpyStash [18] uses an in-memory hash table using linear chaining to index key-value pairs on the flash storage. It achieves extremely low RAM footprint. SILT [20] combines three basic key-value stores: log store, hash store, and sorted store. Mercury [19] is a chaining hash table connected with the linked list. It achieves to reduce DRAM accesses.

There are some work based on other data structures. Masstree [22] is an in-memory structure combining B+-trees and tries. It offers fast random access and supports range queries. NVMKV [23] is a key-value store that uses advanced FTL capabilities. ForestDB [6] uses a HB+-trie that combines a disk-based trie structure with B+-tree. It is efficient for indexing and retrieval of the keys with various lengths. Tucana [24] uses an extension of B^{ϵ}-tree to provide the asymptotic properties for the insert operations.

7. Discussions

Before this paper, the Windows registry has been considered as a storage for configurations or meta data. In this section, we discuss features of WR-Store in terms of the general-purpose data store.

Large-scaled storage: Table 10 shows some limitations of the Windows registry [28]. Registry keys are used for indexing in WR-Store. 255 characters of the key name and 512 depths of keys are enough for making a scalable index. As discussed in Section 3.2, the hash result is represented in hexadecimal. This means that we can create 16^{255} combinations for one-level of the registry, and we can make 512 levels of the registry. The size limitation for the value name is 16,383, and accordingly, the maximum length of the key for WR-store is 16,383. Typically, keys are represented in decimal. Thus, we can deal with 10^{16383} keys, which is enough to store large-scaled key-value pairs.

Table 10. Limitations of the Windows registry [28].

Registry Element	Size Limitation
Key name	255 characters
Depth of the registry keys	512
Name attribution in the value	16,383 characters
Data attribute in the value	Available memory

Even if the size limitation of the value in the Windows registry is stated as an available memory, Microsoft recommends to store values whose sizes are less than 2 MB for efficiency [28]. Typical recent sources to collect key-value data such as social networking services and search engines deal with data whose sizes are much less than 2 MB. For example, Twitter allows 280 bytes (i.e., less than 1 KB) for a Tweet; Facebook allows 63,206 bytes (i.e., about 61 KB) for a post; Google crawls 500 KB of a Web page as the maximum size (https://developers.google.com/search/reference/robots_txt). Furthermore, we can store large-scale of values by working with the file system. That is, we store a large-scaled value as a file in the file system and maintain only a path for the file in the Windows registry.

Migration: We can easily migrate the data stored in WR-Store into another machines only if the Windows operating system is running on the target system because all the Windows versions support the Windows registry. We can easily export data stored in one WR-Store and import them

into another WR-Store in the other machine by using a Windows built-in command (For exporting data stored in "HKCU\DBName", the following Windows built-in command can be used: "regedit/E export.reg 'HKCU\DBName"', which stores the exported data into export.reg. For importing data, the following command can be used: "regedit/S export.reg", which imports the exported data into the other systems.). In addition, for the other environments that do not support registry, we can migrate the data in WR-Store into any other environments that support existing key-value stores by using the ETL (Extract-Transform-Load) method proposed in Section 5.

Portability: WR-Store can be used in every Windows version (e.g., Windows 7/8/8.1/10 and Windows server 2008/2012/2016) because the Windows registry and associated APIs can be worked in any versions of Windows. The executable file compiled by one environment can be worked on other Windows environments without any modifications.

Stable operations: The Windows registry is always accessed from the Windows applications. Especially, the system processes such as explorer.exe and services.exe are always working even if the users do not execute any applications at all. They use the built-in APIs provided by Windows to access to the Windows registry, which has been shown in Table 1. WR-Store is also developed based on the same APIs. Figure 4 shows that all the algorithms for the basic operations of the key-value store (i.e., *Get*, *Put*, and *Delete*) are based on the built-in APIs. As a result, WR-Store becomes one of normal Windows applications that are working with the Windows registry.

In addition, to show the safety of WR-Store even in extreme cases where the intensive updates to the Windows registry are concurrently occurred, we performed the stress test in Section 4.7. That is, we monitored the performance of WR-Store as we increase the number of the processes that are concurrently accessing to the Windows registry. As a result, we showed that the performance of WR-Store is maintained even in the case of intensive registry workloads where 1000 processes accessing to the registry actively are concurrently running.

8. Conclusions

In this paper, we have proposed a method to construct a lightweight key-value store that designs the Windows registry as the key-value store. First, we have defined a mapping of the components in the key-value store onto the components in the Windows registry. Second, we have proposed a hash-based multi-level index structure so as to distribute the key-value data balanced and to efficiently access them. Third, we have implemented the basic operations (i.e., *Get*, *Put*, and *Delete*) for the key-value store using the Windows native APIs. We call the proposed key-value store *WR-Store*. Fourth, we have proposed an efficient ETL (Extract-Transform-Load) method to migrate from WR-Store into the other key-values stores. This allows us to extend the coverage of WR-Store not only for the environments where Windows operating systems are running but also for any other environments that support the existing key-value stores. Fifth, we have conducted the empirical analysis to understand the performance of WR-Store and have tuned the performance.

We have shown that WR-Store has the following strong points. First, WR-Store is a lightweight key-value store because it uses Windows built-in structures and native APIs without installing of additional libraries and applications. Experimental results show that WR-Store reduces the size of the executable file by 17.77∼153.73 times than the other key-value stores. Second, through extensive experiments using synthetic and real data sets, we have shown that WR-Store is comparable to or even more efficient than the state-the-of-the-art systems (i.e., RocksDB, BerkeleyDB, and LevelDB). Especially, when the number of continuous operations is small, WR-Store outperforms the other key-value stores because the other key-value stores require much initial cost while WR-Store does not. Moreover, we verify the scalability of WR-Store by showing that WR-Store becomes much more efficient than the other key-value stores as the size of data set increases. We also show that the performance of WR-Store is maintained even in the case of intensive registry workloads where 1000 processes accessing to the registry actively are concurrently running.

WR-Store is based on the Windows internal structure, i.e., Windows registry. Thus, it can be lightweight and efficient because it does not require additional structures and the cost to manage them. At the same time, WR-Store inherits the limitations of the Windows registry. For example, WR-Store inherits the limitations of the registry element size, which has been discussed in Section 7. Nevertheless, a very important observation is that this paper is the first effort to show that we can use the Windows registry as a general-purpose data storage. Furthermore, we have shown strengths of WR-Store: (1) *lightness*, (2) *efficiency*, and (3) *scalability*. The proposed technique can be used in any environments where Windows operating systems are running with the minimum effort for the installation. For the other environments without Windows operating systems, we can easily migrate data to the other any environments that support existing key-value stores by using the ETL (Extract-Transform-Load) method proposed by this paper.

For future studies, we have two research directions. First, we investigate how to improve the performance of WR-Store by working with in-memory key-value stores. Second, we extend WR-Store into distributed environments on multiple nodes. WR-Store has a property that can be easily applied into multiple nodes where the Windows operating systems are running, but we need to investigate an efficient method for distributing the data and integrating the results.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1C1B5084424). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2019R1A6A1A03032119).

Acknowledgments: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1C1B5084424). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2019R1A6A1A03032119).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Redis. Available online: http://redis.io (accessed on 10 September 2019).
2. LevelDB. Available online: https://github.com/google/leveldb (accessed on 10 September 2019).
3. Facebook. RocksDB: A Persistent Key-Value Store for Fast Storage Environments. Available online: https://rocksdb.org (accessed on 10 September 2019).
4. Oracle Berkeley DB. Available online: https://www.oracle.com/database/technologies/related/berkeleydb.html (accessed on 10 September 2019).
5. Memcached: A Distributed Memory Object Caching System. Available online: https://memcached.org (accessed on 10 September 2019).
6. Ahn, J.S.; Seo, C.; Mayuram, R.; Yaseen, R.; Kim, J.S.; Maeng, S. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Trans. Comput.* **2015**, *65*, 902–915. [CrossRef]
7. Wu, J.; Yunfei, F.; Peng, S. Sensor Fusion for Recognition of Activities of Daily Living. *Sensors* **2018**, *18*, 4029. [CrossRef] [PubMed]
8. Zakhary, V.; Agrawal, D.; Abbadi, A. Caching at the Web Scale. *Proc. VLDB Endow.* **2017**, *10*, 2002–2005. [CrossRef]
9. Oneil, P.; Cheng, E.; Gawlick, D.; Oneil, E. The Log-Structured Merge-Tree (LSM-tree). *Acta Inform.* **1996**, *33*, 351–385. [CrossRef]
10. Apache Hbase. 2018. Available online: http://hbase.apache.org (accessed on 10 September 2019)
11. Lakshman, A.; Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40. [CrossRef]
12. Lu, L.; Pillai, T.S.; Gopalakrishnan, H.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage (TOS)* **2017**, *13*, 5. [CrossRef]
13. Sears, R.; Raghu, R. bLSM: A General Purpose Log Structured Merge Tree. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 20–24 May 2012; pp. 217–228.

14. Shetty, P.; Richard, P.S.; Ravikant, M.; Binesh, A.; Justin, S.; Erez, Z. Building Workload-Independent Storage with VT-trees. In Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13), San Jose, CA, USA, 12–15 February 2013; pp. 17–30.
15. Wu, X.; Yuehai, X.; Zili, S.; Song, J. LSM-Trie: An LSM-Tree-Based Ultra-Large Key-Value Store for Small Data. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, Santa Clara, CA, USA, 6–7 July 2015; pp. 71–82.
16. Zhang, W.; Yinlong, X.; Yongkun, L.; Yueming, Z.; Dinglong, L. Flame DB: A Key-Value Store With Grouped Level Structure and Heterogeneous Bloom Filter. *IEEE Access* **2018**, *6*, 24962–24972. [CrossRef]
17. Debnath, B.; Sudipta, S.; Jin, L. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.* **2010**, *3*, 1414–1425. [CrossRef]
18. Debnath, B.; Sengupta, S.; Li, J. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-Based Storage. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; pp. 25–36.
19. Gandhi, R.; Gupta, A.; Povzner, A.; Belluomini, W.; Kaldewey, T. Mercury: Bringing Efficiency to Key-Value Stores. In Proceedings of the 6th International Systems and Storage Conference, Haifa, Israel, 30 June– 2 July 2013.
20. Lim, H.; Bin, F.; David, G.A.; Michael, K. SILT: A Memory-Efficient, High-Performance Key-Value Store. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles, Cascais, Portugal, 23–26 October 2011; pp. 1–13.
21. Andersen, D.G.; Jason, F.; Michael, K.; Amar, P.; Lawrence, T.; Vijay, V. FAWN: A Fast Array of Wimpy Nodes. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; pp. 1–14.
22. Mao, Y.; Eddie, K.; Robert, T.M. Cache Craftiness for Fast Multicore Key-Value Storage. In Proceedings of the 7th ACM European Conference on Computer Systems, Bern, Switzerland, 10–13 April 2012; pp. 183–196.
23. Marmol, L.; Sundararaman, S.; Talagala, N.; Rangaswami, R. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In Proceedings of the 2015 USENIX Annual Technical Conference, Santa Clara, CA, USA, 6–7 July 2015; pp. 207–219.
24. Papagiannis, A.; Giorgos, S.; Pilar, G.; Angelos, B. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In Proceedings of the 2016 USENIX Annual Technical Conference, Denver, CO, USA, 22–24 June 2016; pp. 537–550.
25. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogles, W. Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, Washington, DC, USA, 14–17 October 2007; pp. 205–220.
26. Microsoft. Registry Data Types. Available online: https://docs.microsoft.com/en-us/windows/win32/shell/hkey-type (accessed on 10 September 2019).
27. Microsoft. Registry Functions. Available online: https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-functions (accessed on 10 September 2019).
28. Microsoft. Registry Element Size Limit. Available online: https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-element-size-limits (accessed on 10 September 2019).