algorithms

MDPI

Article

A Novel Global Key-Value Storage System Based on Kinetic Drives †

Xiang Cao * and Cheng Li

School of Computing and Information Systems, Grand Valley State University, Allendale, MI 49401, USA; lich@mail.gvsu.edu

* Correspondence: caox@gvsu.edu

† This paper is an extended version of our paper published in the 20th Int'l Conf on Internet Computing and Internet of Things (ICOMP'19), Las Vegas, NV, USA, 29 July–1 August 2019.

Received: 7 June 2020; Accepted: 24 September 2020; Published: 29 September 2020

Abstract: NoSQL databases are flexible and efficient for many data intensive applications, and the key-value store is one of them. In recent years, a new Ethernet accessed disk drive called the "Kinetic Drive" was developed by Seagate. This new Kinetic Drive is specially designed for key-value stores. Users can directly access data with a Kinetic Drive via its IP address without going through a storage server/layer. With this new innovation, the storage stack and architectures of key-value store systems have been greatly changed. In this paper, we propose a novel global key-value store system based on Kinetic Drives. We explore data management issues including data access, key indexing, data backup, and recovery. We offer scalable solutions with small storage overhead. The performance evaluation shows that our location-aware design and backup approach can reduce the average distance traveled for data access requests.

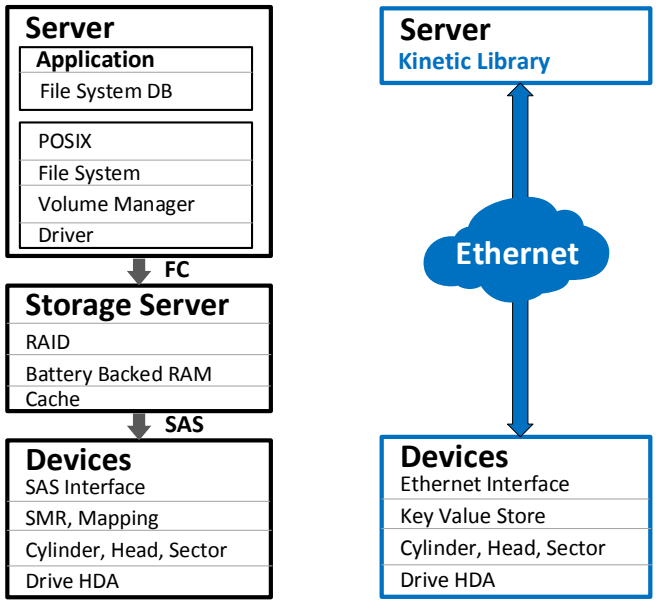
Keywords: NoSQL database; key-value store; kinetic drive; key indexing; global scale

1. Introduction

Currently, with more and more data being generated every day at a rapid rate, Big Data applications are very popular. It is important to provide better data storage and processing for these applications. Recently, in addition to traditional relational databases, NoSQL databases [1,2] offer more flexible and higher performance, especially for data in irregular formats.

As one of the NoSQL databases, key-value stores (KVS) [3] provide a simple, but efficient data access. KVS systems (e.g., Amazon Dynamo [4], Facebook Cassandra [5], and LinkedIn Voldemort [6]) have been widely adopted for many data intensive applications. In a key-value store system, a data record consists of a *key* and a *value*. A *key* is an index for identifying the data record. A *value* stores the actual data of any type, such as numbers, web pages, or images. With a given *key*, the data record can be quickly retrieved, changed, or deleted.

Recently, Seagate [7] developed a new disk drive device called the "Kinetic Drive" [8,9]. Specifically designed for storing key-value pairs, the Kinetic Drive is an example of object-based storage devices (OSDs) [10,11] and active disks [12–14]. With its own CPU, RAM, and built-in LevelDB system [15], the Kinetic Drive can independently run key-value operations. The Kinetic Drive significantly changes the storage stack and data management for key-value stores. As shown in Figure 1a, there is a storage server in the traditional storage stack for accessing data stored in block-based disk drives. However, as shown in Figure 1b, key-value pairs stored in Kinetic Drives can be directly accessed by the application via Ethernet connection. Hence, the Kinetic storage stack offers easy and flexible data access.



(**a**) Traditional Storage Stack (**b**) Kinetic Storage Stack

Figure 1. Traditional vs. Kinetic storage stack [8,16].

In Big Data environments, huge amounts of key-value data are generated from different regions worldwide. It is necessary to manage those data by a global key-value store system. Many Kinetic Drives form a very large pool of key-value store devices when they are connected to networks. Users should be provided with efficient data access by this global key-value store system. Given a *key*, this KVS system should quickly find which Kinetic Drive stores the data. In that case, we need metadata servers to manage key-value data stored in Kinetic Drives. Hence, this global KVS system requires a good design and architecture of data management for these metadata servers.

It is challenging to propose such data management for metadata servers in a global KVS system. First, it is important to map huge amounts of key-value pairs to Kinetic Drives in various regions. Second, data usually are stored in multiple copies in this global KVS system due to redundancy and fault tolerance requirements. Third, the fact that key-value pairs are dynamically received should be considered. Fourth, a location-aware design is necessary for quick data access. Fifth, the data management scheme should have a small and affordable overhead. Last but not least, with multiple metadata servers, coordination among them is needed. A good design should include these considerations.

This paper is an extended version of the work published in [17]. In this paper, we propose a novel global key-value store system based on Kinetic Drives. Several indexing scheme designs of Kinetic Drives were proposed in [16,18], which only considered data allocation in a cluster from one location rather than focusing on a global system. In contrast, we offer a comprehensive design and architecture for a global key-value store system using Kinetic Drives among various regions.

The contributions of this paper are as follows.

- A data management scheme with small storage overhead is proposed for metadata servers to manage Kinetic Drives in different regions.
- A comprehensive solution is offered considering several practical aspects, such as data access, *key* indexing, data backup, and recovery.
- Quick data access is provided by a location-aware design.
- For the global KVS system, our solution includes the coordination among different metadata servers in various regions.

The rest of paper is organized as follows. Section 2 gives the related work. In Section 3, we introduce the background of Kinetic Drives. Section 4 gives the motivation of this paper. In Section 5, we propose our design and architecture. The numerical result of storage overhead is given in Section 6. The performance evaluation is shown in Section 7. Finally, we conclude our paper and present the future work in Section 8.

2. Related Work

NoSQL databases [1,2] provide many Big Data applications with better performance, efficiency, and flexibility. Recently, many types of NoSQL databases have been developed, such as key-value stores [3], column stores [19–21], document stores [22–25], and graph databases [26–28].

There are many large-scale key-value store systems, such as Amazon Dynamo [4], Facebook Cassandra [5], and LinkedIn Voldemort [6]. In these KVS systems, data are stored in traditional block-based hard disk drives. Hence, it is necessary to have dedicated storage servers and/or layers to manage data. However, our global key-value store system in this paper is different from these existing systems. It is based on Kinetic Drives specifically designed for key-value pair data without dedicated storage servers/layers.

Object-based storage devices (OSDs) [10,11] manage data as objects rather than traditional data blocks. With more powerful CPU and memory, active disks [12–14] have been designed to process data on disks to achieve better performance than traditional hard disk drives. Developed by Seagate [7], the Kinetic Drive is an example and special case of OSDs and active disks. Some documents about Kinetic Drives were provided in [8,9]. Recently, the research work in [29] compared the performance between Kinetic Drives and traditional hard disk drives. In [16], the authors investigated the data allocation issues for Kinetic Drives in a cluster in one location and proposed multiple indexing schemes to manage data. The authors in [18] considered Kinetic Drives' bandwidth constraints and proposed the solution to allocate key-value pairs accordingly. Both research works in [16,18] only focused on a cluster of Kinetic Drives in one data center location, so that the solution did not consider a global-scale KVS system. In this paper, however, we focus on a global key-value store system with multiple regions and offer our design and architecture.

Many peer-to-peer (P2P) systems [30–34] adopt the idea of key-value pairs. Although data in P2P systems are stored in key-value pair format, the goal of P2P is to offer file sharing, which is different from our paper. In this paper, our global key-value store system is to provide services for users to store and access data.

There are some other research works [35–39] about various perspectives of key-value store systems. However, they did not consider Kinetic Drives as the storage devices. In this paper, we take advantage of Kinetic Drives and use them to build a global key-value store system, focusing on its design and architecture.

3. Background

3.1. Preliminaries of Kinetic Drives

Seagate recently announced the invention of Kinetic Drives [8,9]. Figure 1 shows the difference between traditional and Kinetic storage stacks. We can see that traditional hard disk drives are connected via an Serial Attached SCSI (SAS) or AT Attachment (ATA) interface, while Kinetic Drives are accessed via Ethernet connections. Currently, there are two Ethernet ports in a Kinetic Drive, and its storage capacity is 4 TB. Kinetic Drives support a *key* size up to 4 KB and a *value* size up to 1 MB for key-value pairs. With Ethernet connections and supported by TCP/IP, data stored in Kinetic Drives can be quickly accessed by users via the IP addresses of the drives using the following built-in APIs.

- Get(key): Given a *key*, gets the key-value pair.
- GetPrevious(key): Given a *key*, gets the previous key-value pair.
- GetNext(key): Given a *key*, gets the next key-value pair.

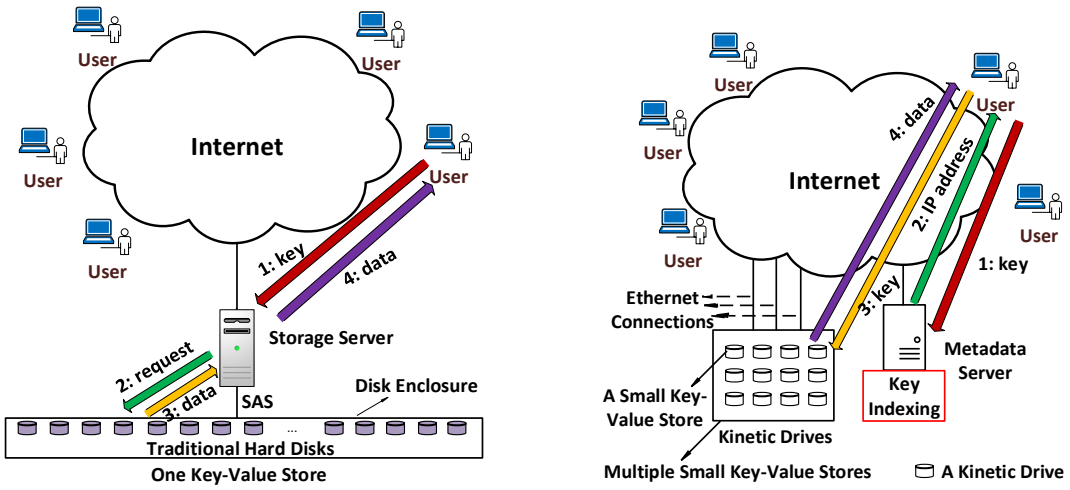
- GetKeyRange(key1, key2): Given *key1* and *key2*, gets the key-value pairs in the range between them.
- Put(key, value): Given a *key* and a *value*, stores the key-value pair.
- Delete(key): Given a *key*, deletes the key-value pair.

With its own CPU, RAM, and built-in LevelDB system [15], the Kinetic Drive manages data by sorting and storing key-value pairs in an order based on *keys*. With a given *key*, a Kinetic Drive can independently operate itself to perform key-value operations and directly returns the result to the user. Hence, we can see that a Kinetic Drive becomes a part of the storage resource pool for users to access data, when it is connected to the Internet.

In addition, there is a unique feature of Kinetic Drives—P2P operation. Direct data migration via P2P operation is available between two Kinetic Drives because they are accessed via IP addresses. Therefore, a P2P data transfer can be initiated from one Kinetic Drive to another, given the IP address of the destination drive. This direct P2P data transfer does not need any storage server as the intermediary.

3.2. Comparison between Traditional and Kinetic KVS Systems

The comparison between traditional and Kinetic KVS systems is shown in Figure 2 in detail. In traditional KVS, a storage server directly manages data stored in multiple disk drives. In Figure 2a, a *key* arrives first at the storage server, which finds the specific drive containing the key-value pair. Then, the storage server fetches those data and sends them to the user. Hence, the storage server is the intermediary for the actual data *value* to go through.



(**a**) Traditional Key-Value Store System (**b**) New Kinetic Key-Value Store System

Figure 2. Key-value store systems: traditional vs. Kinetic [16].

It is different for the Kinetic KVS system. To manage a set of Kinetic Drives and find which drive stores the specific key-value pair, a metadata server is needed. This metadata server only maintains a mapping between *keys* and the IP addresses of the Kinetic Drives, as shown in Figure 2b. We can see that when a *key* arrives at the metadata server, it finds the IP address of the Kinetic Drive storing that key-value pair and returns the IP to the user. Then, the user directly contacts the specific Kinetic Drive to access the key-value pair. Hence, the actual data *value* does not need to go through the metadata server to arrive at the user.

3.3. Advantages and Impact of Kinetic Drives

The above architectural difference shown in Figure 2 is crucial for reducing the amount of data going through the metadata server. Although *keys* still have to arrive at the metadata server, the actual data (*values*) do not. In key-value store systems, the *value* size usually is much larger than the size of the *key*. Hence, with the huge amount of key-value pairs in the whole system, the total data traffic going through the metadata server is significantly reduced.

Furthermore, the traditional KVS system consists of the storage server(s) and multiple hard disks drives, as shown in Figure 2a. Those hard disk drives only store data, without the capability to process the key-value operations by themselves. However, in Figure 2b, this new Kinetic KVS system actually has multiple small key-value stores plus a metadata server. Each Kinetic Drive is a small key-value store and is able to process key-value operations by itself independently. Hence, users can interact with Kinetic Drives in parallel so that the throughput is improved.

In addition, due to P2P operations, it is easier to migrate data among Kinetic Drives to further improve scalability. The metadata server does not even need to participate in the data transfer between two Kinetic Drives. In a large-scale deployment, this feature becomes especially important because of parallel data transfer among those drives. Therefore, the system can greatly mitigate the bottleneck effect of the server.

In [29], experimental results showed different write throughput performed by a traditional KVS system and Kinetic Drives when the *value* sizes were changed, as shown in Figure 3 as an example. We can see that when the *value* size is small, the traditional LevelDB server indeed has greater throughput. This is because the CPU of the dedicated storage server is more powerful than the CPU of the Kinetic Drive, so that the server processes data quicker. When the *value* size becomes larger, the amount of data also increases. Due to the internal LevelDB system, data access operations take place inside the Kinetic Drives. Therefore, for Kinetic Drives, those large amounts of data do not need to be frequently accessed by the storage server, so that the I/O traffic can be reduced. However, for the traditional LevelDB server, it still has to access data from/to disk drives, thus generating many I/O operations, which reduce the throughput. Hence, the throughput is greatly increased for Kinetic Drives with a larger *value* size.

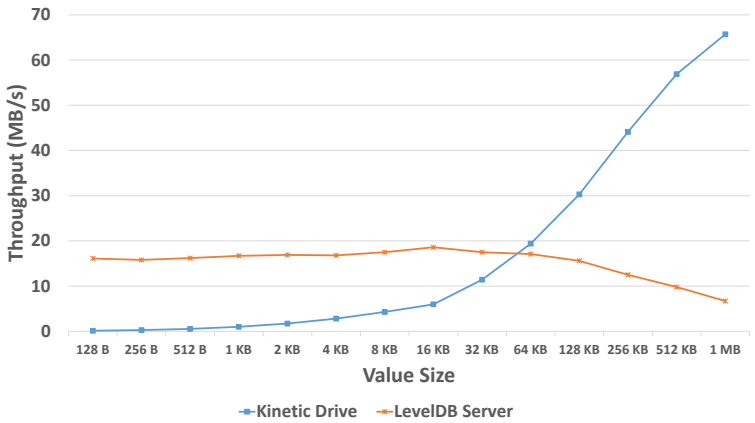


Figure 3. Comparison of sequential write throughput between the Kinetic Drive and the LevelDB server (*Key* size = 16 B, one million key-value pairs) [29].

3.4. Big Data and Kinetic Drives

Data volume is one of the important dimensions of the Big Data concept. The amount of data currently is huge and keeps growing at a rapid rate. In the Big Data environment, variety is another dimension. There are huge amounts of data sources. For example, in the Internet of Things environment, all kinds of "things" are generating data, such as vehicles, appliances, people, mobile devices,

and industrial sensors. These different types of data sources have been generating data in various and irregular formats (e.g., videos, images, texts, numbers, and web pages).

To deal with various and irregular data formats, NoSQL databases usually offer more flexible and better performance than traditional SQL databases. As one of the NoSQL databases, the key-value store system provides a simple, but efficient data storage and access scheme. Key-value store systems have been widely used in many Big Data applications, such as Amazon Dynamo [4], Facebook Cassandra [5], and LinkedIn Voldemort [6]. When those applications generate a data record such as a customer's information or a user's profile, it can be stored in a key-value pair format in those decentralized systems.

Kinetic Drives are specifically designed for key-value store systems to store key-value pair data. As mentioned previously, we can find that Kinetic Drives significantly change the data access mechanism for user-to-drive and drive-to-drive interactions. The performance of the entire large-scale KVS system can be improved by Kinetic Drives due to better parallelism and scalability.

Kinetic Drives can be widely deployed to replace traditional hard disk drives for key-value store systems. As shown in Figure 1a,b, Kinetic Drives reduce the complexity of data access and storage. With proper metadata management schemes, those large-scale key-value store systems such as Amazon Dynamo [4], Facebook Cassandra [5], and LinkedIn Voldemort [6] can use Kinetic Drives to improve the performance and reduce the overhead of the system. Furthermore, with Kinetic Drives, the Seagate Open Storage Platform [40] can even work with storage in OpenStack (an open standard cloud computing platform) [41], which is widely deployed in the cloud. Kinetic Drives can be the storage devices for key-value pair data, and our design in this paper offers scalable solutions for a global-scale key-value store system.

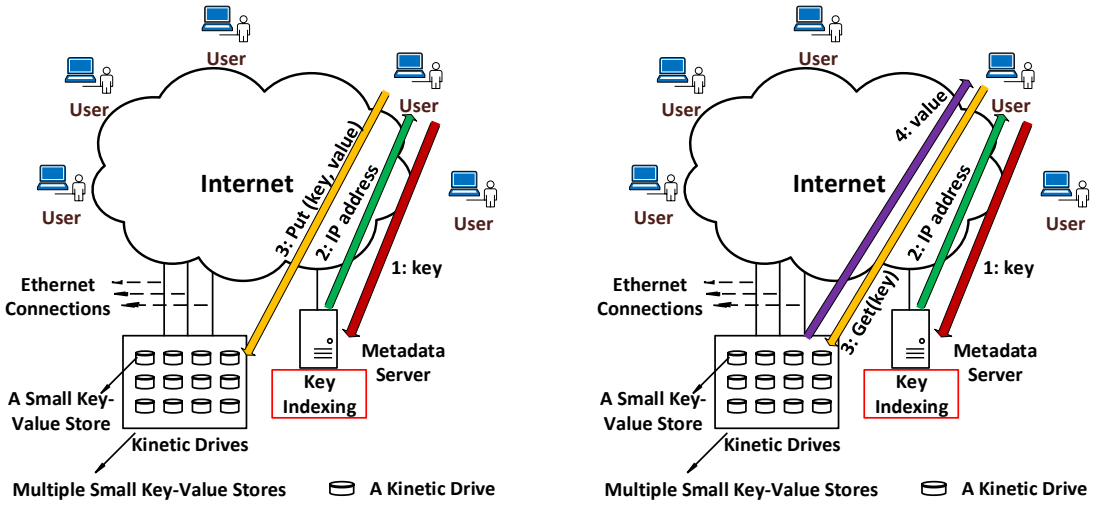
4. Motivation

Figure 4 shows data storing and retrieval by a user with only a cluster of Kinetic Drives. Based on the discussion above in this paper, we can see that a metadata server is needed for a cluster to find the destination drive.

Currently, in the Big Data environment, a huge amount of data sources worldwide is distributed in different regions and keeps generating key-value pairs. In order to store data and provide access, a global system with a large pool of Kinetic Drives is needed. Hence, similar to Figure 4, metadata servers are required to manage those Kinetic Drives. Given a *key*, the system should quickly find which Kinetic Drive in which region has stored the key-value pair. Furthermore, if this given *key* is for a new key-value pair, the location of it to be stored should be decided as well. It is important to design a global KVS system for users in many regions to access data.

There is a naive approach that there is no mapping table between the *keys* and IP addresses of drives. In this case, given a *key*, all drives' IP addresses are returned to the user by the metadata servers. However, this naive approach is unrealistic, because the user has to contact all the Kinetic Drives to search their data. This "exhaustive search" is not possible, considering the unacceptable cost and inefficiency of the amount of drives. Hence, the system needs an indexing scheme to map *keys* to IP addresses of drives.

On the other hand, it is also impractical to build a huge indexing table to map every key-value pair to the IP address of each Kinetic Drive. The *key* size a Kinetic Drive supports is up to 4 KB. If all the possible key-value pairs were mapped in the indexing table, there would be $2^{4\ \mathrm{KB}}$ ($= 2^{4096 * 8}$) records, which is so large that they could not be stored in metadata servers, along with the huge search cost. This "exhaustive mapping" should be avoided as well, since the size of the indexing table(s) should be limited. We need to explore the tradeoff between these two extremes "exhaustive search" and "exhaustive mapping" for the indexing scheme design.



(**a**) User Storing the Data (**b**) User Retrieving the Data

Figure 4. Data access with a cluster of Kinetic Drives [42].

Practically, some crucial factors should be considered for a large-scale system, such as data backup, recovery, and coordination among metadata servers. For example, it is important to backup data from a drive to others in the case of data corruption and drive failure. Furthermore, with the Big Data scenario, more than one metadata server is needed. The coordination among them is necessary.

5. Design and Architecture

In this section, we present our design and architecture in detail.

5.1. Overview

The goal of our work is to present a global key-value store system. We assume the entire system is divided into multiple regions worldwide. We believe the concept of regions is important, similar to regions (e.g., U.S. East, Asia Pacific) in Amazon Web Services (AWS) [43]. In each region, there is a group of Kinetic Drives that offers key-value pair storage and access. There are also many users dispersed globally in different locations, accessing key-value pairs stored in Kinetic Drives. Figure 5 shows the overall picture of the system with N regions.

In order to manage this group of Kinetic Drives for each region, a metadata server is needed. This metadata server is also the first contact for users in this region accessing data. We assume all the metadata servers worldwide know the IP addresses of each other.

In the real deployment, practically, there should be multiple synchronized metadata servers in each region for reliability and fault tolerance. In this paper, to better explain our design, we use a metadata server to represent these multiple ones in each region. In other words, we assume this one metadata server takes care of data management in its region. The data synchronization issues among metadata servers in a region are classical problems in distributed systems, and they are not unique for Kinetic Drives. In this paper, we focus on the coordination among metadata servers in different regions.

For redundancy and fault tolerance considerations, each drive in a region is assumed to have two backup drives in two other regions. In other words, each key-value pair has three copies distributed in three regions. In the rest of this paper, we use the term "*original data*" to represent the primary copy of the key-value pair. These *original data* are initially stored by users. The term "*original drive*" means the drive initially storing these *original data* as the primary copies. Furthermore, the terms "*Backup 1*"

and "*Backup 2*" are used to indicate the first and second additional backup copies in two other regions. These backup copies are replicated by the KVS system based on the *original data*.

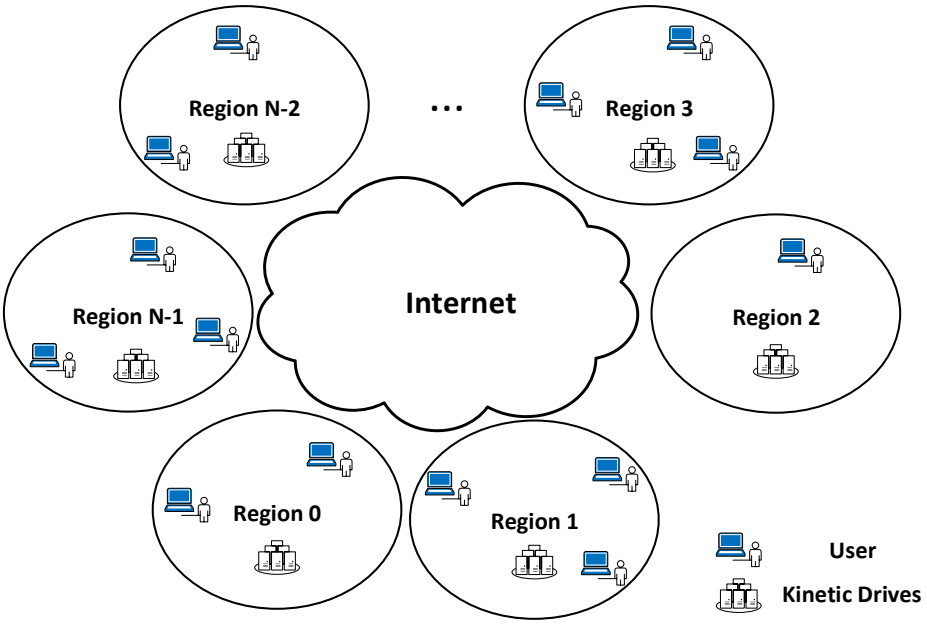


Figure 5. A novel global key-value store system based on Kinetic Drives.

5.2. Indexing Scheme for Original Data

Several *key* indexing schemes for a cluster of Kinetic Drives in one location were discussed in [16]. This paper, in contrast, focuses on a global KVS system with multiple regions. In this paper, we generally follow one approach in [16] to assign key-value pairs of the *original data* to drives. Generally speaking, it maps the key-value pairs to drives based on *key* ranges. Essentially, a *key* consists of a series of bits 0 and 1. Furthermore, we apply the same idea to map key-value pairs to different regions. The impacts of other indexing schemes in [16] are beyond this paper's scope and will be considered as our future work.

In this paper, the *keys* are assumed to be randomly generated (e.g., a common way to generate a *key* is to apply a hash function on the *value*), so that they are more or less uniformly distributed. We use the prefix of a *key* to decide a key-value pair's region. We assume there are N regions so that the first $\lceil log_2 N \rceil$ bits are used to make the decision. For example, if there are eight regions in total, then the region is decided by the first three bits of the *key*. The *original data (primary copy)* of the key-value pair whose *key* starts with "000" is stored in Region 0. Therefore, the mapping between prefixes of *keys* and regions works in the following way.

$$\begin{aligned} \underline{000}\ldots &\longrightarrow \text{Region } 0 \\ \underline{001}\ldots &\longrightarrow \text{Region } 1 \\ \ldots\ldots &\longrightarrow \ldots\ldots \\ \underline{111}\ldots &\longrightarrow \text{Region } 7 \end{aligned}$$

We apply the same method to locate which Kinetic Drive stores the key-value pair within a region. In one region, a part of the all the drives is used for storing *original data (primary copies)*, and the rest of the drives are for backup and other possible usages. The metadata server in each region maintains a table to map *key* ranges to drives. If there are M drives for storing original data in a region (of course, the number M is variable among different regions), the additional $\lceil log_2 M \rceil$ bits after the first $\lceil log_2 N \rceil$ bits are used for locating the specific drive. In Table 1, we show an example of Region 4 (eight regions in total), assuming 128 drives are used for storing *original data*. In that case, we know that all the

key-value pairs whose *keys* start with "100" are stored in this region. The fourth to 10th bits decide the specific drive. Hence, the key-value pair whose *key* starts with "100 0000011" (indicating the *key* range from "100 0000011 0..." to "100 0000011 1...") is stored in the third drive (IP: 135.37.118.6) in Region 4 (drive numbers start from 0).

Table 1. An example of the key indexing table for *original data* (Region 4).

Key Range	IP Address of the Drive
100 0000000 0... to 100 0000000 1...	135.37.118.3
100 0000001 0... to 100 0000001 1...	135.37.118.4
100 0000010 0... to 100 0000010 1...	135.37.118.5
100 0000011 0... to 100 0000011 1...	135.37.118.6
...	...

5.3. Initial Stage

When key-value pairs are generated, we use the above indexing scheme to store *original data* in the drives at the initial stage. Figure 6 shows our system with multiple regions. At the very beginning, all drives in all regions are empty. When a user in Region 1 generates a key-value pair, it first contacts the metadata server MS1 of the user's region by sending the *key*. Based on the prefix of the *key*, the metadata server MS1 looks at the *key* and finds where (i.e., which region) this key-value pair should be stored (Region 2 in this case) as the *original data (primary copy)*.

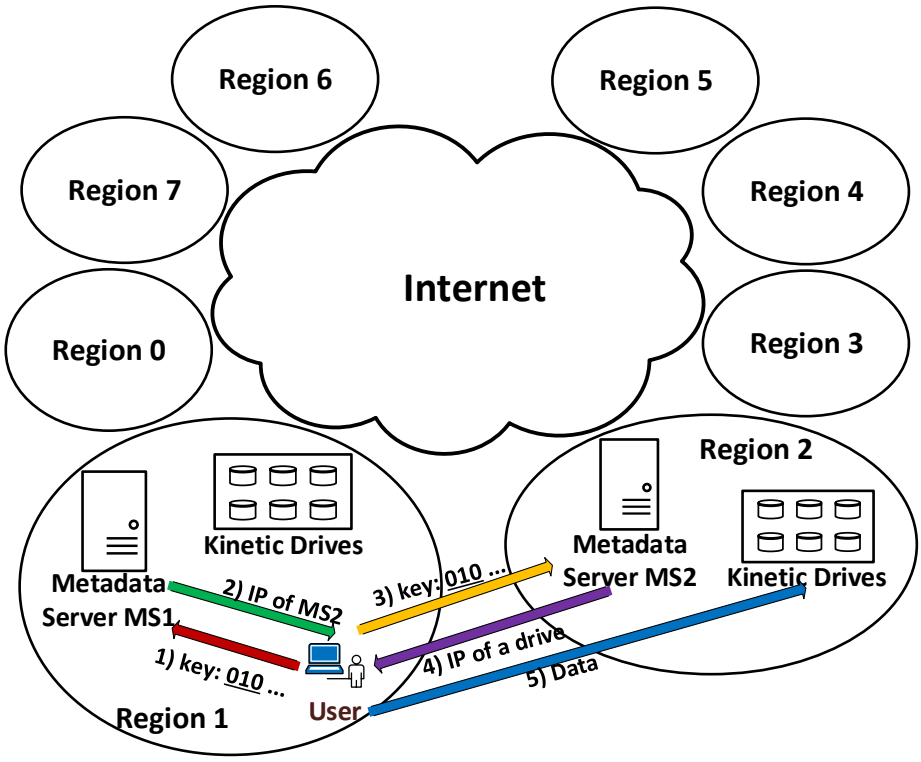


Figure 6. An example of our key-value store system. MS, metadata server.

The IP address of metadata server MS2 of Region 2 is returned to the user by the metadata server MS1. Next, the user sends the *key* to that metadata server MS2, which finds the specific drive that should store the key-value pair, based on additional $\lceil log_2 M \rceil$ bits after the first $\lceil log_2 N \rceil$ bits, as mentioned earlier. Then, metadata server MS2 returns the IP address of that particular Kinetic Drive to the user. Finally, the user directly contacts the drive via its IP address to store the key-value pair.

We can see that during this initial stage, only *original data (primary copies)* are stored without additional backup copies.

5.4. Preparation and Rules for Backup

For the practical deployment, it is likely that a drive can be backed up (locally) to the drive(s) in the same region for future quick data recovery. In that case, it is trivial that the metadata server just adds entries in the mapping table to reflect the backup. In this paper, we focus on location-aware data access so that we intentionally show how to backup data to drives in different regions. Although local backups in the same region can be helpful, sometimes, this inter-region backup can even provide more fault tolerance and resilience for the system.

For the purpose of backing up the *original data* to two different regions, the metadata server in each region keeps track of its drives' access frequencies by users in all regions during the initial stage. For example, if a user from Region 2 accesses Drive 3 in Region 4 once, the metadata server (MS4) in Region 4 increases Drive 3's access frequency from Region 2 by one. This information of data access can be easily maintained, because a metadata server has to be contacted first by a user. Hence, after the initial stage, metadata servers have clear knowledge about the access frequencies of all their drives accessed by all regions, which is helpful for backup. In this paper, we assume the initial stage is a relatively long enough period to gather data access information. The access frequencies afterwards are stable, so that they generally follow the statistics in the initial stage.

At some moment, a drive's initial stage ends, when this drive is full or getting too full (e.g., reaching a threshold, such as 80% full). Then, backup and drive splitting happen. In this paper, the concepts of backup and drive splitting are different. Backup means that data in this drive are copied and transferred to two additional drives in two other regions. Drive splitting means that part of the data needs to be transferred to another drive (in the same region) to acquire more capacity for future storage.

We have the following rules for backup:

- To reduce the overhead, instead of backing up key-value pairs one by one, the system backs up the entire drive in one shot when needed.
- The first backup copy of the drive (i.e., *Backup 1*) is stored in another region by which the access frequency of this drive is the highest, after the initial stage. For example, we assume Drive 3 in Region 4 needs backup, and this drive is accessed mostly by users in Region 2. Hence, the first backup copy (*Backup 1*) of Drive 3 in Region 4 is a drive in Region 2. This rule considers the access frequency, so that Region 2's users who frequently access data in Drive 3 of Region 4 do not have to interact with Region 4 for future access (instead, they can just access data from the backup drive in Region 2), so the distance and delay are reduced.
- The second backup copy of the drive (i.e., *Backup 2*) is stored in another region, which is the farthest from the region of the *original drive* and the region chosen by the above rule (using the highest access frequency). For instance, following the above example, Region 5 is the farthest region from Regions 2 and 4. Then, the second backup copy (*Backup 2*) of Drive 3 in Region 4 is stored in a drive in Region 5. This rule is applied so that users physically far from Regions 2 and 4 can just access data in their closer region (i.e., Region 5) to reduce the distance and delay as well.

In this paper, we show our criteria to choose two backup drives for location-aware access. In the real deployment, these criteria could be changed for different requirements. We can find that backup copies are not only for fault tolerance, but also can help reduce distance and delay for data access.

5.5. Backup and Splitting

The system backs up a drive to two other regions, when this drive is full or getting too full for *original data (primary copies)*. Based on the information of access frequencies and locations mentioned in

the backup preparation and rules, the metadata server of this drive knows the destinations (which two regions) of two backup copies. Hence, it contacts two other metadata servers of those two regions.

After that, each of these two other metadata servers finds an empty drive in its region and returns its IP address to the metadata server of the *original drive* that needs backup. Then, this *original drive* gets those two drives' IP addresses and initiates P2P data transfer operations to them to backup (copy) the data. The data transfer from a drive to another drive does not need a metadata server as the intermediary. As long as the *original drive* knows the IP address of the destination drive from the metadata server, P2P data transfer can be started and finished.

An example of backup and splitting is shown in Figure 7. We can see that Drive 3 in Region 4 needs to be backed up. Based on the backup rules, the metadata server MS4 of Region 4 knows that the destinations for backup should be two empty drives in Region 2 and Region 5, respectively. Then, MS4 contacts the metadata server MS2 of Region 2 and the metadata server MS5 of Region 5. Next, MS2 and MS5 find within their regions that drive D21 and drive D18 are empty drives for backup data respectively, so that the IP addresses of drive D21 and drive D18 are returned to MS4. After that, MS4 shares these IP addresses with its drive D3, which initiates P2P data transfer operations to copy data to drive D21 in Region 2 and drive D18 in Region 5 for backups.

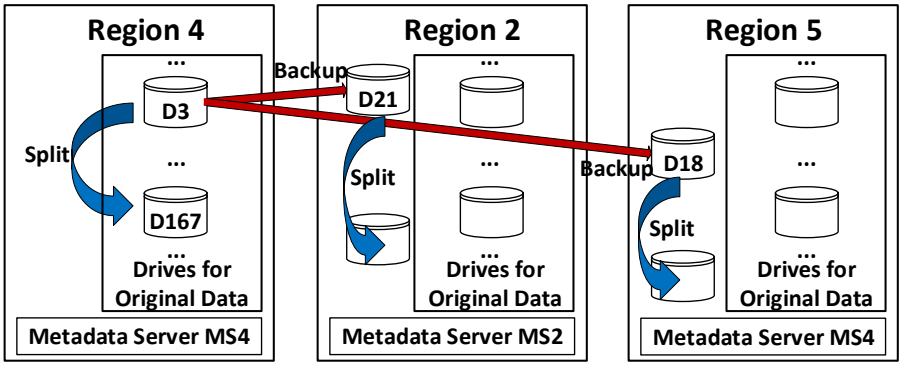


Figure 7. An example of backup and splitting.

There are three identical copies of data stored in drives in three regions after backup. We know those three drives are full or getting too full. To make more storage space, we need to split the data in them and transfer part of the data to other drives. Here, we adopt the approach in [16]. Since each drive covers key-value pairs in a certain *key* range as shown in Table 1, it can split its data into two parts with smaller and greater *keys* and transfer one part to another empty drive in the same region. Then, the metadata server updates the indexing table.

For example, now drive D3 in Region 4 is full or getting too full, so that it needs to be split. As shown in Table 1, this drive stores key-value pairs whose *key* range is from "100 0000011 0..." to "100 0000011 1..." currently. The metadata server can find an empty drive (e.g., drive D167) as the destination for splitting. Then, the second half of the data with greater *keys* in drive D3 is transferred to drive D167. The metadata server MS4 also splits the indexing mapping for drive D3 and updates that as follows.

$$\underline{100}\ \underline{0000011}\ \underline{00}\ldots \text{to } \underline{100}\ \underline{0000011}\ \underline{01}\ldots \longrightarrow \text{drive D3's IP}$$
$$\underline{100}\ \underline{0000011}\ \underline{10}\ldots \text{to } \underline{100}\ \underline{0000011}\ \underline{11}\ldots \longrightarrow \text{drive D167's IP}$$

From Figure 7, we can see that the data in drive D3 of Region 4 have been backed up to drive D21 of Region 2 and drive D18 of Region 5. Hence, those two drives (D21 of Region 2 and D18 of Region 5) need to be split as well following the same way as D3 of Region 4. The indexing tables in metadata servers MS2 and MS5 are also updated.

From the technical perspective, it is feasible to separate key-value pairs into two parts with smaller and greater *keys* for data splitting, because Kinetic Drives support the APIs "GetPrevious(key)" and "GetNext(key)". Practically, the metadata server can record the smallest and greatest *keys* of data in a drive. Hence, using these APIs, the Kinetic Drive can easily separate data.

5.6. Future Data Access

There are three drive copies in the system, after a drive has been backed up to two other regions. When a user tries to access a key-value pair, it is best for the user to get these data from the region closest to his/her location. In order to support that, each metadata server for all regions maintains a table, recording the mapping between *key* ranges and the IP addresses of all three drive copies, as shown in Table 2, which is an advanced version based on Table 1.

Table 2. Mapping table maintained in each metadata server.

Key Range	Original Drive's IP	1st Backup Drive's IP	2nd Backup Drive's IP
Key Range 1			
Key Range 2			
...			
Key Range *M*			
Additional Backup Drives' Key Ranges			
...			

In Table 2, *M* is the number of drives for storing the *original data (primary copies)*. Each *key* range is covered by three drives, including *original drive* in this metadata server's region and two additional backup drives in two other regions. The IP addresses of these three drives are stored in three columns respectively. For the example in Figure 7, the *key* range of drive D3 in Region 4 and the IP addresses of drives D3 (Region 4), D18 (Region 5), and D21 (Region 2) are stored (drive D3 is the *original drive* in Region 4) in metadata server MS4's table. In addition to these *M* rows, Table 2 also stores the mapping between *key* ranges and IP addresses of those drives as additional backup copies for other regions. For example, in metadata server MS2's table, the *key* range of drive D21 in Region 2 and the IP addresses of drives D3, D18, and D21 are also stored, because drive D21 is one backup copy of drive D3 in Region 4.

When a user reads/retrieves a key-value pair, he/she first sends the *key* to the metadata server in its region. After the metadata server receives the *key*, it checks the first $\lceil log_2 N \rceil$ bits to see in which region this key-value pair is supposedly stored, based on our indexing scheme described in Section 5.2.

- If the *key* falls into the same region as the user and is covered by the *key* ranges of *original drives* in this region, then the metadata server directly looks up its mapping table, finds out which drive covers the *key*, and returns the drive's IP address to the user. Finally, the user directly contacts this drive for data retrieval. For example, in Figure 7, if a user in Region 4 reads a key-value pair whose *key* is covered by the *key* range of the Drive 3 in Region 4, this scenario is applied.
- If the *key* is not covered by the *original drives' key* ranges of the metadata server, but it is covered by the *key* range of an additional drive as the backup copy of another region, a similar process happens. For example, in Figure 7, a user in Region 2 initiates a read request for data whose *key* is covered by drive D21's *key* range of Region 2. This *key* is not covered by the *original drives* in Region 2, because drive D21 is the additional backup copy of drive D3 in Region 4. The metadata server MS2 still returns the IP address of drive D21 of Region 2 to the user, since drive D21 is the closest one to the user (i.e., in the same region).
- If the *key* is neither covered by the *original drives' key* ranges of the metadata server, nor the *key* ranges of the additional drives as backup copies of other regions, the metadata server then

involves another metadata server in a different region for data retrieval. For example, in Figure 7, a user in Region 2 retrieves data whose *key* is covered by drive D155 in Region 5. However, in Region 2, both *original drives* and additional drives as backup copies of other regions do not cover this *key*. In that case, the metadata server MS2 checks the *key* and finds that it is covered by *original drives* in Region 5. Hence, it returns the IP address of metadata server MS5 to the user. After the user contacts the metadata server MS5, then MS5 looks up its table as shown in Table 2 and finds that the *key* range of drive D155 covers this *key*. Instead of immediately returning drive D155's IP address to the user, MS5 compares the distances from the user to three drive copies' locations respectively, figures out which drive copy is the closest one to the user (it could be drive D155's additional backup drive in other regions such as Region 1), and returns its IP address to the user.

We can see that by taking advantage of three drive copies, the user receives the IP address of the drive closest to its location.

When a user writes/updates a key-value pair into a drive, a similar process is applied. The difference is that, instead of receiving only the IP address of the closest drive, the user gets the IP addresses of three drive copies covering these data's *key*, from the metadata server. Then, it contacts these three drives and writes/updates data into them, so that all three drives get updated. For example, in Figure 7, if a user writes/updates a key-value pair into drive D3 in Region 4, the metadata server returns the IP addresses of drive D3 (Region 4), drive D21 (Region 2), and drive D18 (Region 5) to the user. Then, the user updates data in drive D21 (Region 2) and drive D18 (Region 5).

5.7. Data Recovery

All data are stored in three drives from three different regions. If one of these three drives fails, the system can replace the bad drive and recover the data from two other drives. The metadata server just finds the closer drive, initiates a P2P data transfer to recover data, updates its mapping table, and notifies two other metadata servers in different regions.

For example, in Figure 7, if drive D3 in Region 4 fails, the metadata server MS4 figures out that either drive D21 in Region 2 or drive D18 in Region 5 is the closer one to drive D3. Then, MS4 finds an empty drive (e.g., D137) in its region and initiates a P2P data transfer operation from drive D21 or D18 to drive D137. Finally, MS4 replaces drive D3's IP address with D137's in its mapping table and notifies metadata servers MS2 and MS5 to update their mapping tables to reflect the change as well.

Following a similar idea of local backup discussed in Section 5.4, practically, for each region, data can be quickly recovered from drives within the same region, if some drives are reserved as local backups for the *original data*. However, in this paper, we just show that our design is also feasible for data recovery between different regions, especially if inter-region data recovery is necessary.

5.8. Adding Drives

As time goes on with newly generated data, it is possible that more drives need to be added. When a set of new Kinetic Drives is added in a region, some of them can be used to store the *original data* of this region, while the rest of them become other regions' backup drives. In both cases, when new drives are available, the metadata server can simply update Table 2 for new key-value pairs by filling in the IP addresses of new drives. Hence, our approach discussed in this section can work well in this scenario.

5.9. Discussion

Part of our design can be applied to other key-value store systems without using Kinetic Drives. For example, maintaining multiple copies per data is a usual way to handle fault tolerance and reliability. Furthermore, location-aware design is a common scheme to speed up data access.

However, most of our design is tailored to Kinetic Drives. As mentioned previously, Kinetic Drives are specifically designed for storing key-value pairs and providing data access. Given a *key*, Kinetic Drives can run and perform searches for data by themselves. With this unique feature, data management becomes easier, so that traditional storage servers can be eliminated. Only simple metadata servers are needed for managing those Kinetic Drives. Hence, the indexing table design mapping *keys* to drives is unique for Kinetic Drives. Furthermore, many other issues such as data access, data backup and splitting, and data recovery become more efficient by taking advantage of the P2P data transfer offered by Kinetic Drives. With P2P data transfer, data can be directly sent and received by Kinetic Drives without going through the storage servers.

6. Numerical Result: Storage Overhead

In this section, we show that the storage overhead for metadata servers to manage those Kinetic Drives is affordable.

For example, we assume there are 32 regions worldwide. In each region, there are 2048 Kinetic Drives including drives for *original data* and additional backup copies for other regions. In that case, the first five bits in a *key* are used to decide the region, and the additional 11 bits are used to locate drives within a region.

We have 32 metadata servers in total. For each metadata server, there are 2048 rows and four columns (*key* range plus three IPs) in its mapping table, as shown in Table 2, assuming all drives are used. The *key* range takes 5 bits + 11 bits = 16 bits = 2 bytes to store. Furthermore, to support the drive splitting mentioned in Section 5.5, the metadata server maintains the smallest and largest *keys* for each drive. Each *key* is up to 4 KB, so that each entry in the first column for each row takes 4 KB * 2 + 2 B $\approx$ 8 KB. Assuming IPv6 is used for the drives, each of the second to fourth columns takes 128 bits = 16 B. Hence, each row in the mapping table takes fewer than 9 KB (if a shorter *key* is used, it can be further reduced). The total storage overhead for each metadata server is less than 9 KB * 2048 = 18,432 KB = 18 MB, which is affordable (To be precise, we also need to include the access frequencies of drives accessed by all regions mentioned in Section 5.4. However, the storage overhead of these numbers is relatively very small compared to the current result, so that it can be negligible and does not affect our conclusion.), We can see that the dominating storage overhead for each row in the mapping table is the cost to store the smallest and greatest *keys* for drive splitting, and others can be negligible.

Based on the above calculation, we can find that the storage overhead for each metadata server is approximately up to $9M$ KB (9 KB * M), where M is the number of Kinetic Drives for the region. Table 3 summarizes the total storage capacity of a region and the storage overhead of its metadata server for different numbers of drives. For example, only 72 MB of storage overhead are needed for a cluster of 8192 Kinetic Drives to provide a 32,768 TB storage capacity.

Table 3. Storage Capacity and Overhead of a Region (4 TB per drive).

Number of Kinetic Drives	Total Storage Capacity (in TB)	Storage Overhead up to (in MB)
512	2048	4.5
1024	4096	9
2048	8192	18
4096	16,384	36
8192	32,768	72

We conclude that the storage overhead for each metadata server is very affordable and roughly linearly proportional to the number of drives in its region. Hence, a good scalability is achieved.

7. Performance Evaluation

While this paper focuses on the high-level design for the entire global key-value store system based on Kinetic Drives, in this section, we conduct simulations to show the performance of the data backup approach and rules discussed in Section 5.4.

7.1. Simulation Setup

We simulated a global key-value store system with 16 and 32 regions respectively. In each region, there is a cluster of Kinetic Drives accessed by users. The locations of those clusters and users are randomly generated and distributed in a normalized 1000 * 1000 map.

For each Kinetic Drive in each region, we assume it is accessed by all the users following a Poisson distribution with a mean of 500 times.

Network latency is an important performance metric of a large-scale networked system. Propagation delay is a typical component of the network latency. It is the amount of time taken by the first bit to travel from the sender to the receiver over the physical medium of the network link. Propagation delay depends on the distance between sender and receiver and the propagation speed, as shown in the following equation.

$$Propagation\ Delay = Distance / Propagation\ Speed$$

We can see that it is proportional to the distance between the source and destination in the network, and a shorter distance can reduce the propagation delay. Since clusters of Kinetic Drives and users are dispersed among different locations, the propagation delay directly reflects the distance between a user and the target Kinetic Drive, to show the cost of a data access. Moreover, with a shorter distance, the number of intermediate networking devices (e.g., routers) along the path may be decreased as well, possibly reducing processing and queuing delays, so that the entire network latency can be further shortened.

This paper discusses the high-level architecture design of a global key-value store system, especially focusing on the backup rules and coordination among metadata servers, Kinetic Drives, and users. Hence, in order to evaluate this design, we use the average propagation delay per data access request for all drives in all regions by all the users as the metric. As mentioned above, this propagation delay reflects the distance between users and the target Kinetic Drives, which also has influence on the entire network latency.

We compare our result with the random approach, which just randomly chooses two regions to find two available Kinetic Drives for the *original data*'s backups, without considering the access frequencies and distances. To fairly show the advantage of our approach, however, in the random approach, when a user tries to access a Kinetic Drive, we still use the same way as our approach to choose a Kinetic Drive in the region with the shortest distance to the user as the target drive to calculate the result. In other words, for the random approach, the randomness is only applied for the selection of backup drives.

7.2. Results

We first conducted simulations with 1000 users. We varied the number of Kinetic Drives per location, 512, 1024 and 2048, as shown in Figures 8 and 9.

From Figures 8 and 9, we can see that our backup approach outperforms the random one, in terms of the normalized average propagation delay for data access requests. In other words, the average propagation delay in our approach is shorter than the random selection. This is because we consider the data access frequencies when the system selects the first backup drive, so that many frequently accessed data can be accessed near the users. Furthermore, our backup approach considers the distances so that some users can access data from regions that are closer.

Comparing the results between Figures 8 and 9, we can find that the average propagation delays per request are generally stable in the random approach when the number of regions is increased from 16 to 32. The results for 16 regions are smaller than those for 32 regions in our approach. This is because with more regions, the randomness of cluster and user distribution plays a more important role, so that the performance improvement in our approach becomes less obvious.

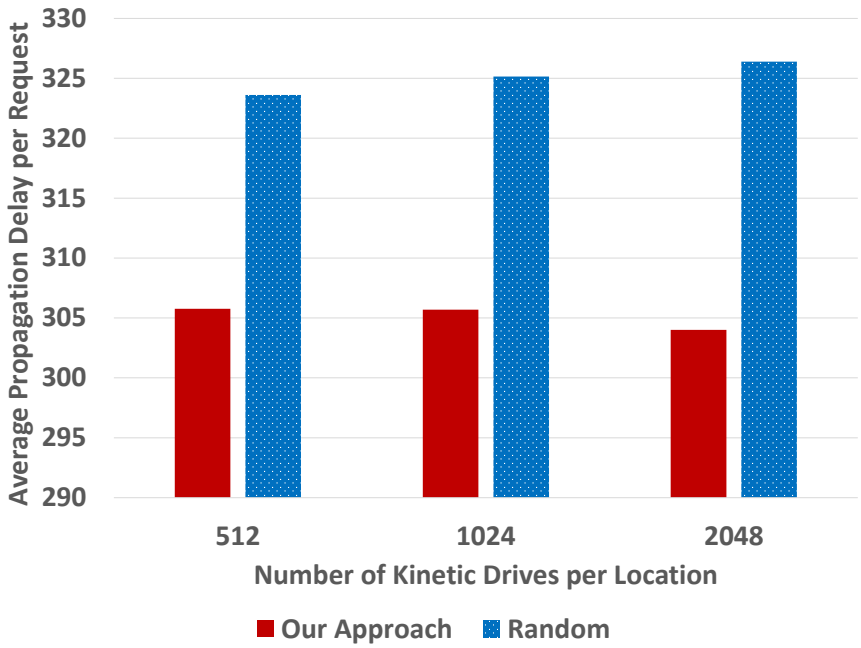


Figure 8. Average propagation delay per request (normalized): 16 regions, 1000 users.

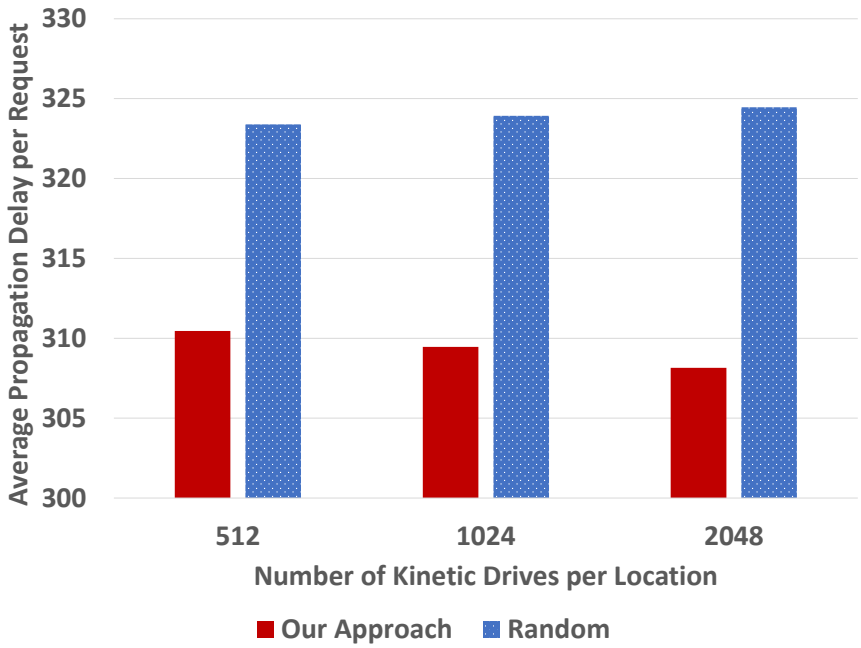


Figure 9. Average propagation delay per request (normalized): 32 regions, 1000 users.

Next, we conducted simulations with 1024 Kinetic Drives per location, varying the number of users, 500, 1000, and 2000, as shown in Figures 10 and 11.

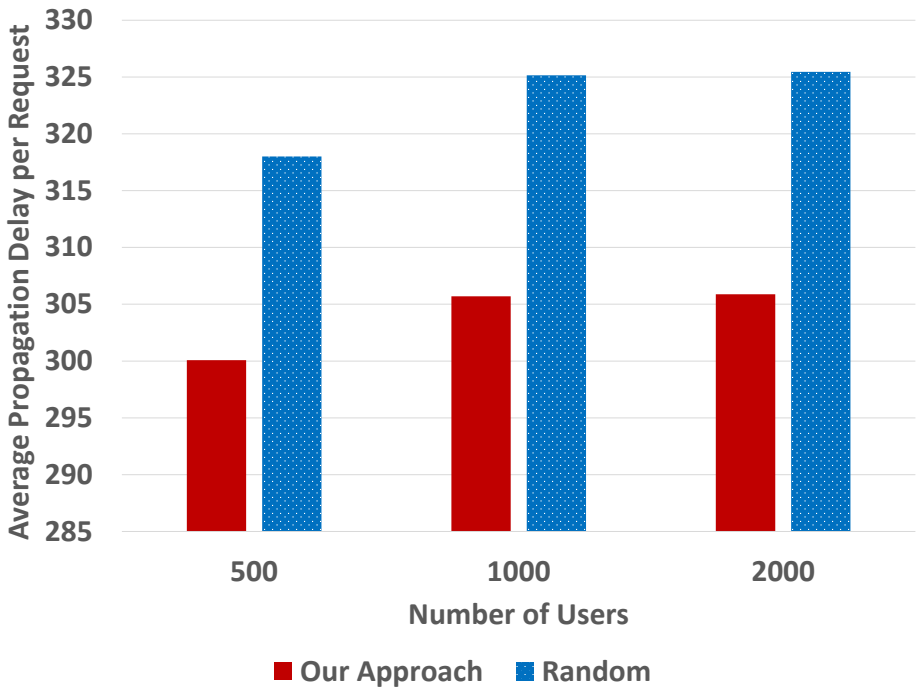


Figure 10. Average propagation delay per request (normalized): 16 regions, 1024 drives per location.

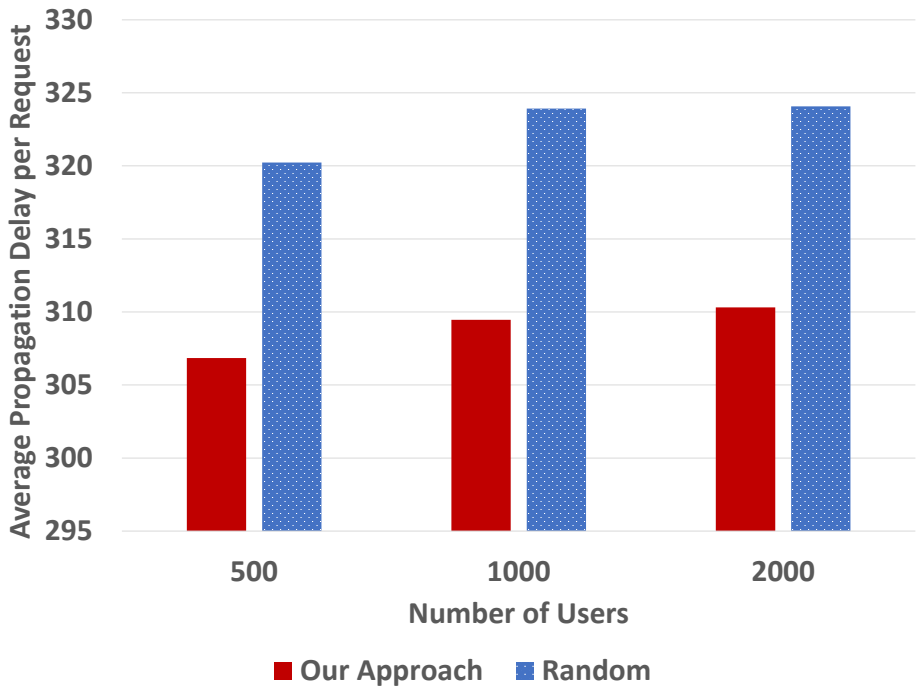


Figure 11. Average propagation delay per request (normalized): 32 regions, 1024 drives per location.

From Figures 10 and 11, we can see that the results are consistent with Figures 8 and 9. The normalized average propagation delay for data access requests in our backup approach is shorter than the random one.

Similar to the results between Figures 8 and 9, we can find that the average propagation delays in Figures 10 and 11 are generally stable in the random approach. The performance improvement in our approach is smaller when the number of regions is increased from 16 to 32, for the same reason mentioned previously.

To sum up, with the consideration of data access frequencies and distances, the average propagation delay for data access requests can be reduced. Based on the discussion mentioned previously, for the practical deployment, processing/queuing delays are potentially decreased as well because of the shorter distance.

8. Conclusions

In many Big Data applications, as the one of NoSQL databases, key-value store systems provide simple and flexible solutions. Kinetic Drives were recently developed by Seagate especially for efficient key-value pair operations. Users can directly access Kinetic Drives via their IP addressees so that data storage and management complexity can be reduced.

In this paper, we propose a novel global key-value store system based on Kinetic Drives. We investigate data management issues for a large number of Kinetic Drives in different regions. Our proposed approach for data access, *key* indexing, data backup, and recovery is scalable with a small storage overhead. The performance evaluation shows that our location-aware design with the data backup approach reduces the average propagation delay for data access requests.

As future work, we plan to consider different *key* distributions and investigate their impacts on the performance. We will explore various approaches to handle different types of workloads, especially imbalanced scenarios, and propose solutions accordingly.

Author Contributions: Conceptualization, X.C.; formal analysis, X.C. and C.L.; investigation, X.C. and C.L.; methodology, X.C. and C.L.; resources, X.C. and C.L.; supervision, X.C.; validation, X.C. and C.L.; writing, original draft preparation, X.C. and C.L. All authors read and agreed to the published version of the manuscript.

Funding: This article is supported by the Open Access Publishing Support Fund at Grand Valley State University.

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

Abbreviations

The following abbreviations are used in this manuscript:

KVS	Key-Value store
OSDs	Object-based storage devices
P2P	Peer-to-peer

References

1. NOSQL Databases. Available online: http://nosql-database.org/ (accessed on 27 September 2020).
2. Strauch, C.; Sites, U.L.S.; Kriha, W. *NoSQL Databases*; Lecture Notes; Stuttgart Media University: Stuttgart, Germany, 2011.
3. Seeger, M.; Ultra-Large-Sites, S. *Key-Value Stores: A Practical Overview*; Computer Science and Media: Stuttgart, Germany, 2009.
4. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* **2007**, *41*, 205–220. [CrossRef]
5. Lakshman, A.; Malik, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40. [CrossRef]
6. Sumbaly, R.; Kreps, J.; Gao, L.; Feinberg, A.; Soman, C.; Shah, S. Serving large-scale batch computed data with project voldemort. In Proceedings of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, 14–17 February 2012.
7. Seagate. Available online: http://www.seagate.com/ (accessed on 27 September 2020).
8. The Seagate Kinetic Open Storage Vision. Available online: http://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/ (accessed on 27 September 2020).
9. Kinetic HDD. Available online: http://www.seagate.com/enterprise-storage/hard-disk-drives/kinetic-hdd/ (accessed on 27 September 2020).

10. Mesnier, M.; Ganger, G.R.; Riedel, E. Object-based storage. *IEEE Commun. Mag.* **2003**, *41*, 84–90. [CrossRef]
11. Factor, M.; Meth, K.; Naor, D.; Rodeh, O.; Satran, J. Object storage: The future building block for storage systems. In Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology, Sardinia, Italy, 20–24 June 2005; pp. 119–123.
12. Acharya, A.; Uysal, M.; Saltz, J. Active Disks: Programming Model, Algorithms and Evaluation. *ACM SIGOPS Oper. Syst. Rev.* **1998**, *33*, 81–91. [CrossRef]
13. Lim, H.; Kapoor, V.; Wighe, C.; Du, D.H.C. Active disk file system: A distributed, scalable file system. In Proceedings of the 2001 Eighteenth IEEE Symposium on Mass Storage Systems and Technologies, San Diego, CA, USA, 17–20 April 2001.
14. Riedel, E.; Faloutsos, C.; Nagle, D. *Active Disk Architecture for Databases*; Technical Report; Carnegie Mellon University: Pittsburgh, PA, USA, 2000.
15. LevelDB. Available online: https://github.com/google/leveldb (accessed on 27 September 2020).
16. Cao, X.; Minglani, M.; Du, D.H.C. Data allocation of large-scale key-value store system using kinetic drives. In Proceedings of the 2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService), San Francisco, CA, USA, 6–9 April 2017; pp. 60–69.
17. Cao, X.; Li, C. Internet of drives: A global key-value store system using kinetic drives. In Proceedings of the International Conference on Internet Computing and Internet of Things (ICOMP'19), Las Vegas, NV, USA, 29 July–1 August 2019; pp. 68–74.
18. Eldakiky, H.; Du, D.H. Key-value pairs allocation strategy for kinetic drives. In Proceedings of the 2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService), Bamberg, Germany, 26–29 March 2018; pp. 17–24.
19. HBase. Available online: https://hbase.apache.org/ (accessed on 27 September 2020).
20. Column (Data Store). Available online: https://en.wikipedia.org/wiki/Column_(data_store) (accessed on 27 September 2020).
21. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R.E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* **2008**, *26*, 4. [CrossRef]
22. CouchDB. Available online: http://couchdb.apache.org/ (accessed on 27 September 2020).
23. MongoDB. Available online: https://www.mongodb.com/ (accessed on 27 September 2020).
24. Document Stores. Available online: https://db-engines.com/en/article/Document+Stores (accessed on 27 September 2020).
25. Document-Oriented Database. Available online: https://en.wikipedia.org/wiki/Document-oriented_database (accessed on 27 September 2020).
26. Graph Database. Available online: https://en.wikipedia.org/wiki/Graph_database (accessed on 27 September 2020).
27. ArangoDB. Available online: https://www.arangodb.com/ (accessed on 27 September 2020).
28. Neo4j. Available online: https://neo4j.com/product/ (accessed on 27 September 2020).
29. Minglani, M.; Diehl, J.; Cao, X.; Li, B.; Park, D.; Lilja, D.J.; Du, D.H. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In Proceedings of the 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, 15–17 December 2017; pp. 501–510.
30. Zhao, B.Y.; Huang, L.; Stribling, J.; Rhea, S.C.; Joseph, A.D.; Kubiatowicz, J.D. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE J.Sel. Areas Commun.* **2006**, *22*, 41–53. [CrossRef]
31. Rowstron, A.I.T.; Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg 2001, Heidelberg, Germany, 12–16 November 2001; pp. 329–350.
32. Clarke, I.; Sandberg, O.; Wiley, B.; Hong, T.W. Freenet: A distributed anonymous information storage and retrieval system. In Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability 2001, Berkeley, CA, USA, 25–26 July 2001; pp. 46–66.

33. Stoica, I.; Morris, R.; Karger, D.; Kaashoek, M.F.; Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, San Diego, CA, USA, 27–31 August 2001; pp. 149–160.
34. Rhea, S.; Eaton, P.; Geels, D.; Weatherspoon, H.; Zhao, B.; Kubiatowicz, J. Pond: The oceanstore prototype. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies 2003, San Francisco, CA, USA, 31 March–2 April 2003; pp. 1–14.
35. Atikoglu, B.; Xu, Y.; Frachtenberg, E.; Jiang, S.; Paleczny, M. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.* **2012**, *40*, 53–64. [CrossRef]
36. Lim, H.; Fan, B.; Andersen, D.G.; Kaminsky, M. SILT: A memory-efficient, high-performance key-value store. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles 2011, Cascais, Portugal, 23–26 October 2011; pp. 1–13.
37. Lim, H.; Han, D.; Andersen, D.G.; Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, Seattle, WA, USA, 2–4 April 2014; pp. 429–444.
38. Marmol, L.; Sundararaman, S.; Talagala, N.; Rangaswami, R. NVMKV: A scalable, lightweight, FTL-aware key-value store. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, Santa Clara, CA, USA, 8–10 July 2015; pp. 207–219.
39. Escriva, R.; Wong, B.; Sirer, E.G. HyperDex: A distributed, searchable key-value store. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Helsinki, Finland, 13–17 August 2012; pp. 25–36.
40. OpenStack Swift Associated Projects. Available online: https://docs.openstack.org/swift/queens/associated_projects.html (accessed on 27 September 2020).
41. OpenStack. Available online: https://www.openstack.org/ (accessed on 27 September 2020).
42. Cao, X. Efficient Data Management and Processing in Big Data Applications. Ph.D. Thesis, University of Minnesota, Minneapolis, MN, USA, 2017.
43. AWS Regions and Endpoints. Available online: https://docs.aws.amazon.com/general/latest/gr/rande.html (accessed on 27 September 2020).