Data Allocation of Large-scale Key-value Store System using Kinetic Drives

Xiang Cao*, Manas Minglani+, David Hung-Chang Du*
*Department of Computer Science and Engineering, University of Minnesota, Twin Cities, USA
+Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities, USA
Email: xiang@cs.umn.edu; mingl001@umn.edu; du@cs.umn.edu

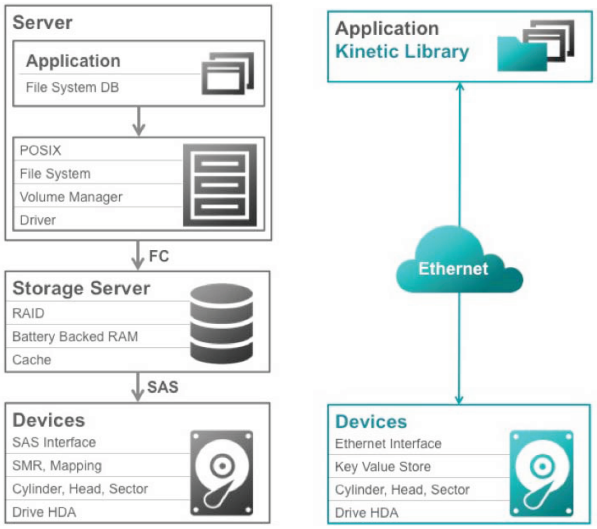
***Abstract*—Nowadays, NoSQL databases become more and more prevalent. As an important category in NoSQL databases, Key-value store is widely used in a lot of data-centric applications, providing flexible and efficient performance. Key-value pairs can be easily accessed given the keys. Recently, a new Ethernet accessed disk drive called "Kinetic Drive" was invented. This drive can be directly accessed by applications via IP address, without through a separate storage server. Given the keys from the user, Kinetic Drive can perform key-value pair operations by itself. It can reduce the management complexity, especially in large-scale deployment in the data center.**

Given a set of Kinetic Drives in a data center, it is important to manage the key-value pairs and store them in an organized way. In this paper, we investigate how to allocate data on large-scale key-value store system in the data center using Kinetic Drives. We show the tradeoff among various design factors. We invent key indexing schemes and allocate data on drives accordingly. We propose approaches to migrate data among drives. Performance evaluation shows that our approaches can handle various key distributions well respectively in different cases.

I. Introduction

Recent years, NoSQL databases [1][2] have been developed to provide more flexibility and better performance than traditional relational databases. As one important NoSQL database, key-value store [3] offers an easy but efficient data storage and management scheme. In the key-value store, a record of data consists of a *key* and a *value*. A *key* is generated as an index to store, retrieve and delete the record. The *value* is the actual data, which can be any type, e.g., graph, video, web pages, numbers. Users can access the data given the keys. Without complicated operations (e.g.,join) or foreign keys in the relational databases, key value store is more convenient and flexible. Data can be easily accessed with "put(key, value)", "get(key)" and "delete(key)" operations. Nowadays, many existing key-value store systems have been widely deployed for various applications, such as Facebook Cassandra [4], Amazon Dynamo [5] and LinkedIn Voldemort [6].

In the data storage industry, Object-based Storage Device (OSD) has been introduced [7][8]. They can manage data as objects, instead of traditional block storage. As an innovative example of OSD and Active Disks [9][10][11], a new disk drive called "Kinetic Drive" [12][13][14] was invented by Seagate [15] a few years ago. This Kinetic Drive greatly changes the data access and management scheme for the disk drives. Kinetic Drive provides a direct Ethernet connection interface between host and drive. Users can directly access the data via Ethernet connection with an IP address. With these features, data access and management with Kinetic Drives becomes easy and flexible, especially in the data center environment. Traditionally, in the data center, there is a separate storage server between applications running in the (computing) server and disk drives. The storage server is used to manage the disk drives and key-value pairs. Users' requests have to go through this separate storage layer to access the data in the drives. This data access and management paradigm lead to extra overhead and complexity. Figure 1(a) shows the traditional storage stack.



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

Figure 1: Comparison of Traditional and Kinetic Storage Stack [14]

With the newly invented Kinetic Drives, however, in the data center, this separate storage server can be removed to reduce the management cost and provide more flexibility, as shown in Figure 1(b). Kinetic Drive has a built-in CPU, RAM and LevelDB [16] that can support key-value operations. Given the keys, the Kinetic Drive can store, update, retrieve and delete the data by itself. The applications

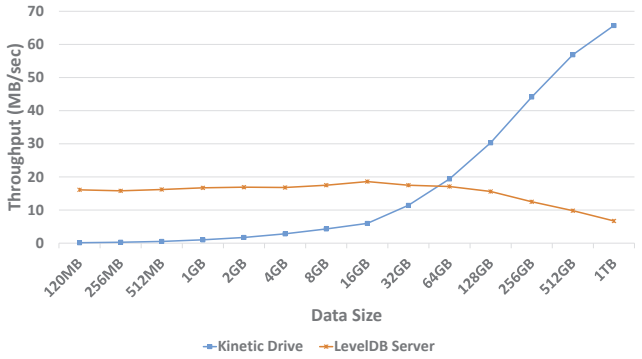


Figure 2: Comparison of Write throughput between Kinetic Drive and LevelDB server [17]

can directly issue users' key-value requests to the Kinetic Drives without the storage server's help. In other words, each Kinetic Drive can be viewed as a small "key-value store" that can perform the key-value operations by itself as long as the keys are provided. Hence, the separate storage server can be removed. In [17], we conduct some experiments to show the advantages of Kinetic Drives compared with traditional architecture as shown in Figure 2 and Table I. Also, with direct Ethernet connection and plug-and-play feature, Kinetic Drives can be more scalable than traditional drives. In Section 3, we discuss the advantages of Kinetic Drives in details.

Obviously, it is important to manage key-value pairs on large-scale key-value store system in the data center. The built-in LevelDB in a Kinetic Drive is used to manage the key-value pairs within that drive. Each Kinetic Drive uses different conceptual levels to organize key-value pairs in that drive with LevelDB. However, it is also critical to manage a large number of Kinetic Drives in a data center in a macro level. When users' key-value requests are received, the corresponding locations of the data should be found out, i.e., which Kinetic Drive stores the key-value pairs for users' particular data. Therefore, an efficient indexing scheme is needed to map the key-value pairs to the drives. This indexing information is stored as a table in a metadata server.

Given the keys from the users, the metadata server should quickly locate the correct Kinetic Drives and forward the users' requests to them. In other words, the metadata server cannot exhaustively search too many drives for key-value pairs, instead, only as few drives as possible should be involved given a key. Also, the indexing table cannot record the mappings between every key-value pair and the corresponding location due to the huge amount of data. A very large indexing table costs too much storage space in the metadata server. In addition, data migration amount among drives (when some drives are full) and percentage of active disks in the data center are important considerations as well. There is a tradeoff among these above design factors. Hence, a careful design of the indexing table and data allocation scheme is necessary.

In this paper, we propose data allocation schemes for large-scale key-value store system in the data center using Kinetic Drives. We show the tradeoff of various design factors. We design the indexing schemes to map key-value pairs and allocate data to disk drives. We consider different key distributions and propose data migration approaches. With limited size of the indexing table, users' requests can be quickly redirected to the corresponding disk drives.

Our paper has the following contributions.

- We show the tradeoff among design factors, such as data migration amount, average percentage of active disks, number of involved disks and number of key ranges per drive.
- We design indexing tables for the metadata server in the data center to quickly find out the correct disk drives given the users' keys.
- We propose approaches to allocate and migrate key-value pairs on Kinetic Drives for different key distributions, considering the tradeoff among design factors.
- Performance evaluation shows our indexing schemes and data allocation approaches can handle various key distributions well respectively in different cases.

The rest of the paper is organized as follows. In Section 2, we show the related work. We present the background and motivation in section 3. We discuss our problem in section 4 and propose our solution in section 5. Performance evaluation is shown in section 6 and we conclude our paper in section 7.

II. Related Work

Some key-value store systems have been developed in recent years, such as Facebook Cassandra [4], Amazon Dynamo [5] and LinkedIn Voldemort [6]. They provide large-scale key-value storage for various applications. Different from our design, these systems operate on traditional disk drives which require separate storage server(s) or layer(s) to manage the data. In addition, they ignore the key distribution in the key-value pairs and assume the keys are usually generated by a hash function providing a typically uniform distribution among the keys. However, in our problem, in order to efficiently support key range search and keep the semantic meaning of the key, we generally assume keys do not simply follow the uniform distribution. Hence, our design consider the data allocation based on various key distributions other than the simple uniform one.

Many peer-to-peer (P2P) systems [18][19][20][21][22] provide key-value store for file sharing. Files can be transferred among distributed users across the Internet without central server(s). In the P2P systems, data are typically stored in key-value pairs. They can be looked up and routed with other peers given the keys. Compared with our problem,

P2P systems are focused on file sharing in a distributed environment. However, our design concentrates on the key-value store in a data center to provide storage service for users.

Object-based Storage Device (OSD) [7][8] has been introduced to provide a novel way to manage data as objects. Active Disks [9][10][11] are also innovate devices to process data. With more powerful CPU and larger memory, Active Disks can perform more functionalities beyond traditional disk drives. As a special case of OSD and Active Disks, Seagate recently announced the invention of Kinetic Drives and provided some documents about the details [12][13][14]. In [17], we evaluate the performance of Kinetic Drives and compare them with traditional hard disks. To the best of our knowledge, our work in this paper is the first one to address the data allocation issue in a large number of Kinetic Drives.

There are some other work about key-value store [23][24][25][26][27]. They propose their research work in different aspects of key-value store with various scenarios. Different from these existing work, our paper considers the features of Kinetic Drives and design the data allocation scheme based on the unique advantages of this new kind of disk drives.

III. Background and Motivation

A. Preliminaries of Kinetic Drive

Kinetic Drives were recently invented by Seagate [12][13][14]. Compared with traditional disk drives, Kinetic Drives can be accessed by host via Ethernet connections instead of SAS or PCI-e buses. Each drive provided by Seagate in our performance testings [17] has a storage capacity of 4TB and 2 Ethernet connections of 1Gb/s. Users can directly access the Kinetic Drive via IP address with the Ethernet cable.

In addition to the unique feature of direct Ethernet connection, the Kinetic Drive supports key-value operations with the built-in LevelDB [16]. Given the key, the Kinetic Drive can access the key-value pairs by itself. With LevelDB, it can read, write, update an delete the data within the drive. Users only need to send the key-value pairs to the drives via IP addresses. The drives we test can support the key size up to 4KB and the value size up to 1MB. The Kinetic Drives support the following key-value operations with APIs.

- Put(key, value): Users can store the key-value pair with the given key.
- Get(key): Users can retrieve the key-value pair with the given key.
- GetKeyRange(key1, key2): Users can retrieve the key-value pairs in the key range between key1 and key2.
- Delete(key): Users can delete the key-value pair with the given key.
- GetNext(key): Users can retrieve the next key-value pair based on the given key.
- GetPrevious(key): Users can retrieve the previous key-value pair based on the given key.

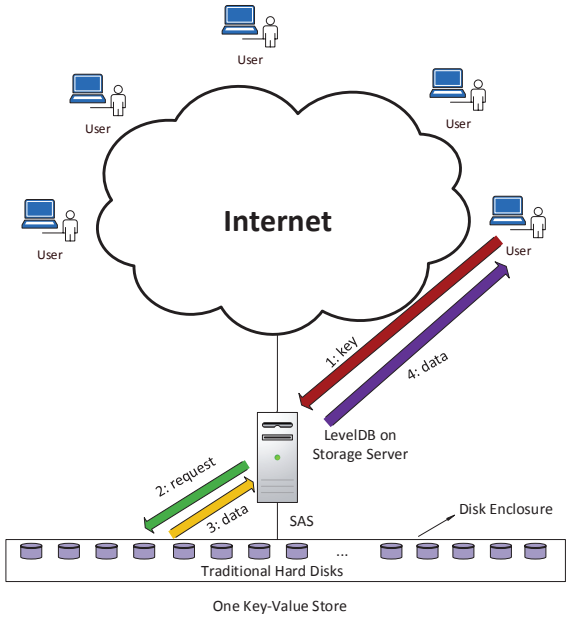
B. Comparison with traditional solution

With these features and properties, Kinetic Drives are very useful for the key-value store. In the key-value store system with traditional disk drives, a separate storage server (or layer) is needed for data management as shown in Figure 1(a). Users' requests from the applications are issued by the (computing) server to the storage server(s). The storage server(s) then connect to the disk drives. This process has to go through the storage server which increases the complexity. The storage server(s) become a separate storage layer to manage the disk drives. The key-value operations are run by the storage server(s) as well, because the traditional disk drives are not able to autonomously perform the key-value operations. In addition, the scalability is limited. The storage server(s) can be a bottleneck for the data center, because they have their limit to manage disk drives in large scale.

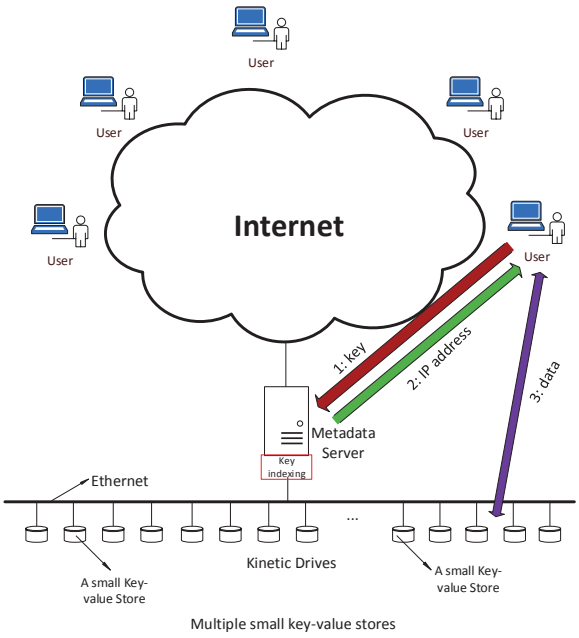
With the Kinetic Drives, the data management becomes easy and flexible. As shown in Figure 1(b), the new Kinetic storage stack is simple. Users' requests from the applications can be directly issued to the disk drives in the data center, without the storage server(s) as the intermediate layer. All we need is a metadata server, which simply stores the key indexing information and forwards the requests to correct Kinetic Drives. Given the IP addresses of the Kinetic Drives, disk drives can be directly accessed. This new stack can remove the storage server(s), which greatly reduces the complexity of data management and increases the scalability. We can easily add more Kinetic Drives by simply connecting Ethernet cables to them.

In large-scale deployment in the data center, the difference between the architectures with traditional hard disks and Kinetic Drives becomes more evident. In key-value store system, LevelDB is widely used and we use it as an example. As shown in Figure 3(a), the traditional architecture is deployed with LevelDB or other key-value store systems on a storage server with non-Kinetic hard disks. This storage server runs LevelDB on it and manages those hard disks. The storage server receives the users' keys, searches the drives, and makes a request to the drives to fetch the data. Each hard disk is only a simple storage device which cannot run key-value pair operations without the storage server (layer). Hence, we can see that this traditional architecture shown in Figure 3(a) is a single large key-value store system which involves storage server and hard disks.

However, with Kinetic Drives, it becomes different. As shown in Figure 3(b), in this new Kinetic key-value store system architecture, each Kinetic Drive can be viewed as a small "key-value store" that can perform the key-value pair operations by itself. A Kinetic Drive has its own CPU, RAM, built-in LevelDB system and Ethernet port with IP address.



(a) Traditional key-value store system



(b) New Kinetic key-value store system

Figure 3: Comparison between traditional and Kinetic key-value store systems

512MB	LevelDB Server	Kinetic Drives
Throughput (MB/s)	14.6	22.45

Table I: Data Transfer Throughput between Two Drives

Those components make a Kinetic Drive an independent small "key-value store". After receiving users' requests, the metadata server can simply forward those requests such as "Get(key)" and "Put(key, value)" to the correct Kinetic Drives via their IP addresses. Then these Kinetic Drives that receive those requests can perform the key-value pair operations by themselves. In other words, the Kinetic Drives replace the work of traditional storage server to run the key-value operations autonomously. As long as the keys are provided, the Kinetic Drives can store, retrieve, update and delete data within their drives, and return the results. Hence, we can see that the new Kinetic architecture in Figure 3(b) are multiple small key-value stores working together with the metadata server.

C. Performance Comparison

To show the advantage of Kinetic Drives, we conduct some experiments to compare the performance between traditional architecture of LevelDB on storage server and this new Kinetic architecture [17]. Two sets of experiments are conducted. One is for the write throughput and the other is for data transfer between drives.

In order to compare the write throughput between traditional and Kinetic architectures, we use one Kinetic Drive and connect it to a server which sends key-value pairs. We also install LevelDB on a storage server and attach a traditional hard disk with the same amount of storage capacity.

As discussed earlier, Figure 2 shows the performance results of sequential write throughput. During the experiments, we keep the number of key-value pairs the same, and increase the value size. We can see that when the value size is small, the traditional LevelDB server works better. This is because the server has to send IP packets encapsulating key-value pairs to the Kinetic Drive. Since the number of key-value pairs is kept the same, even if for a very small key-value pair, it still takes an entire IP packet. Hence, the network overhead impacts the throughput when the data amount is small.

As the value size grows, the data amount also increases. We can see that in that case, Kinetic Drive performs better. This is because the Kinetic Drive itself can run LevelDB. We know that in LevelDB, data need to be frequently updated with sorting and merging. These operations can take place in Kinetic Drive. Once the data are stored in the Kinetic Drive sent by the server, they do not need to be fetched by the server for LevelDB operations or stored back to the drive. This greatly brings a huge advantage. Therefore, we can also see that the throughput grows as the value size increases for

Kinetic Drive.

However, traditional hard disk does not have built-in LevelDB or support operations as Kinetic Drives does. For traditional hard disk, the LevelDB server has to frequently fetch the data from the disk, do some LevelDB operations and send them back to the disk. Hence, the disk I/O becomes a bottleneck so that the throughput drops when the data value size gets larger.

We also compare the data transfer throughput in traditional and Kinetic solutions as shown in Table I []. For Kinetic Drives, one Kinetic Drive issues P2P command to transfer 512MB data in key-value pairs to another drive. For traditional architecture, the LevelDB server reads 512MB data in key-value pairs from one hard drive and write them in another drive. As seen in the Table I, we can find that the data transfer throughput for Kinetic Drives is greater than traditional solution. For Kinetic Drives, the data can be directly transferred between the drives without going through the server. However, for traditional solution, the storage server has to be the intermediate place for data transfer between drives.

D. Scalability

From the above discussion, we can find out that Kinetic Drives have their unique advantages. The results in Figure 2 and Table I only involve a few drives. In large-scale deployment for the data center environment, the advantages of Kinetic Drives become more evident because they can scale well.

Since each Kinetic Drive can support key-value operations by itself, it is easy to add more drives in the data center without too much overhead on the metadata server. The metadata server only needs to manage the data without doing too much computing. With the plug-and-play feature and built-in LevelDB, the key-value store system with Kinetic Drives can be designed in larger scale. Many Kinetic Drives can compute, store users' data in parallel, which can further improve the throughput for the entire system. However, in the traditional architecture, the storage server can be the bottleneck for the system. The limitation on the storage server can impact the scalability, because every computing operation is on the storage server.

Besides, for the data transfer among drives, we can also find out the advantage of Kinetic Drives. Table I only shows data transfer between two drives (and we already see the improvement). In the large-scale environment, the data transfer among drives can happen in parallel without the server's help. However, in the traditional architecture, parallel data migration becomes difficult because of the limitation of the storage server. The data transfer throughput for the entire system with many Kinetic Drives can be further improved compared with traditional solution with non-kinetic drives. Hence, data migration among Kinetic Drives is feasible.

In addition to the above advantages, another important feature is that users can directly communicate with Kinetic Drives via their IP addresses. In our key-value store system design, given the key from the user, the metadata server returns the IP address of the correct Kinetic Drive to the user. Then the user can directly retrieve and store data from/to the Kinetic Drive by "Get(key)" and "Put(key, value)" operations via the IP address. Hence, the *value* data in the key-value pair does not need to go through the metadata server. Since value size is usually larger than the key size, this direct IP address access with Kinetic Drive can greatly decrease the traffic amount to the metadata server, which can further reduce the burden and make the system more scalable. In the traditional solution, every user's requests have to go through the storage server, which can be a bottleneck for the entire system.

E. Motivation

With huge amount of data stored in the data center, many Kinetic Drives are needed. In other words, one Kinetic Drive with TBs of storage capacity is obviously not sufficient. From our previous discussion, we can see that in the data center, the metadata server sends the users' key-value requests to the correct Kinetic Drives. For example, if a user wants to retrieve the key-value pair with the "key = 38293924", the metadata server needs to know where this key-value pair stores (i.e., which Kinetic Drive has it) first, because there are so many Kinetic Drives in total. Hence, it is important for the metadata server to manage these Kinetic Drives and have a clear view of the locations of key-value pairs. Given a key, the metadata server should quickly find out which Kinetic Drive has this key-value pair if it is a "Get(key)" operation. For the "Put(key, value)" operation, the metadata server also needs to know where this key-value pair should be stored.

Obviously, the simplest way is that the metadata server send users' requests to all the Kinetic Drives for data retrieval. Then all the drives will search for the data. However, this naive "exhaustive search" for all drives is absolutely inefficient and has huge costs. Many Kinetic Drives have to do a lot of unnecessary searches for the data, because at the end, only one or a few drives have the data, or even no drive stores them at all. Huge amount of resources are wasted in this naive approach. Hence, we need to design an indexing table for the metadata server to map the key-value pairs to the drives, so that only as few drives as possible are involved for data search.

On the other hand, the indexing table should not be too large. An extreme approach is that the indexing table includes the mappings for every possible key-value pair to drives. For example, the key-value pair with "key = 0000...00" is stored in drive 0, and another key-value pair with "key = 1111...11" is stored in drive 999. In this way, the indexing table would become extremely large. Since the

Kinetic Drive supports the key size up to 4KB, there would be $2^{4KB}(= 2^{4096*8})$ records for all the possible key-value pairs in the indexing table, which is not practical or cannot be stored in the metadata server. Hence, when we design the indexing table, we should limit its size and make sure it does not bring too much storage overhead for the metadata server. Therefore, a careful design for the efficient indexing table is necessary.

IV. Problem

A. Our Scenario

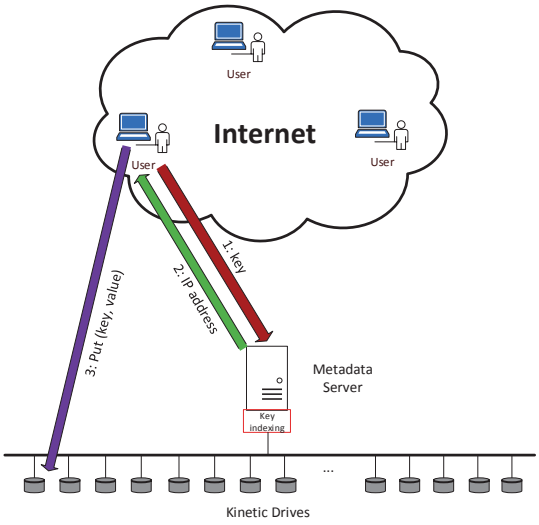
In this paper, we consider the following scenario. In the data center environment supporting key-value store system, there are metadata server(s), data storage (Kinetic Drives) and outside users as shown in Figure 3. The metadata server(s) are connected to the Internet and have the key indexing table. The data storage consists of a large amount of Kinetic Drives. The outside users retrieve and store data in key-value pairs from/to the data center.

In this key-value store system, users can store (”Put(key, value)”), retrieve (”Get(key)”), range query (”GetKeyRange(key1, key2)”) and delete (”delete(key)”) data. Also, the data can be updated by storing the new key-value pair and deleting the old one.

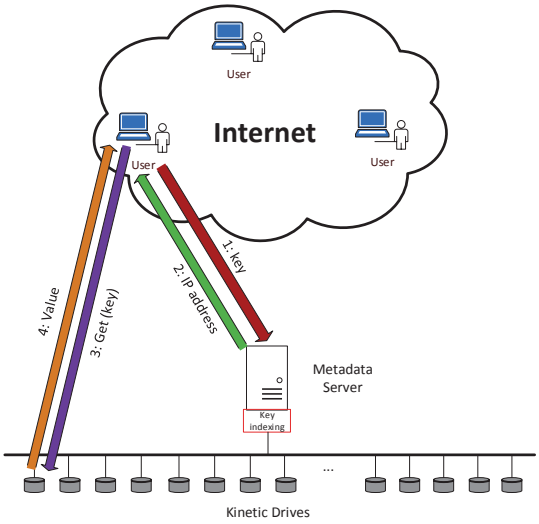
When a user wants to store a key-value pair, the following process happens as shown in Figure 3(a). The user provides the key first (How to generate the key depends on the user, which is beyond this paper’s scope.) Then the user sends the key to the metadata server(s) via the Internet. (In the real system, there should be multiple metadata servers. In this paper, we simply assume these metadata servers can coordinate and the key indexing table can be synchronized to be updated.) The metadata server(s) look up the key indexing table to find out which Kinetic Drive should store this key-value pair, and return this Kinetic Drive’s IP address to the user. After that, the user directly connects to that Kinetic Drive via its IP address and issues ”Put(key, value)” operation to store the key-value pair.

If a user retrieves a key-value pair, it is similar to the above process as shown in Figure 3(b). The user sends the key to the metadata server to get the IP address of the Kinetic Drive that has the data. Then it issues ”Get(key)” operation. If a range query happens, the user may need to ask more than one Kinetic Drives to get the results. The data deletion works similarly in this way.

Since the key indexing table records the mappings between key-value pairs in ranges and the drives (i.e., which range of keys are stored in which drives. The details are explained in the next section.), the metadata server can do some quick preliminary search for data retrieve requests, in the case that there are no such key-value pairs for the given key in the current covered key ranges. If this happens, it means the data is definitely not in any drive. Then the metadata server can just return ”Not Found” to the user so that the user does not need to ask the Kinetic Drives again. Hence, the further unnecessary query for the drives can be avoided to reduce the waste of resources.



(a) User storing the data



(b) User retrieving the data

Figure 4: Key-value store system with Kinetic Drives

B. Research Issues and Design Challenges

There are several research issues in our design for the key indexing table and data allocation on Kinetic Drives as follows.

1) **What does the key indexing table look like?** In the key indexing table, we should map the key-value pairs to the Kinetic Drives. **What should be in the indexing**

table? The design needs to answer these questions.

2) **In the real applications, key-value pairs are continuously coming to the data center. In other words, key-value pairs are dynamically generated.** Therefore, the key distribution will change accordingly. In most cases, it is not realistic to assume some predetermined distributions. This dynamic change of key distribution should be handled. Further adjustment of the indexing table is needed so that data have to migrate among the drives.
3) Although data migration among the drives is unavoidable, it brings cost. It takes time to move the data and has impact on drives' performance. **Hence, data migration amount is an important performance metric for the system. In our design, we should reduce that.**
4) In the key indexing table, the number of key ranges for drives is another important design metric. Ideally, each drive is responsible for only one key range, which is simple but efficient for data management, especially for key range search. However, because of the data migration, some drives may be associated with multiple key ranges in the key indexing table. In our design, we should consider reducing the number of key ranges for drives.

Considering the real scenario, we also have the following design challenges.

1) **Since key-value pairs are dynamically coming to the system, generally, keys are not uniformly distributed.** For example, keys starting with "000" may be more than those starting with "011". Also, users are distributed worldwide. Each user follows its own scheme to generate the keys, so that it is generally unrealistic to assume keys follow some known formats or distributions. Hence, we cannot easily assign the key-value pairs based on some static criteria of key patterns.
2) **When a user requests data retrieval, the metadata server should not return too many Kinetic Drives' IP addresses to the user.** In other words, the number of involved drives for a "Get(key)" operation should be limited. An extreme case is that every drive searches for the data. This exhaustive search is absolutely not acceptable. If too many drives participate in unnecessary searches for key-value pairs, huge amount of resources are wasted.
3) The key indexing table should not be too large as well. As discussed in the motivation, it is not realistic to record the mappings between every possible key-value pair to the drive, although this way is very accurate. **Hence, the size of the key indexing table should be limited and does not bring too much storage overhead for the metadata server.**
4) In addition to the above challenges, percentage of active disks is another important design factor. When the system receives data, it is better to keep data in as few drives as possible to full utilize the disk resources. In other words, we should consider to lower the percentage of active disks in the system).

C. Design Tradeoff and Goal

In summary, we can see that there is a tradeoff among four different design factors. (1) data migration amount; (2) percentage of active disks; (3) the number of searched disks for a request; (4) the number of key ranges per drive. In our design, we should consider this tradeoff.

Our design goal is as follows. Given a set of N Kinetic Drives in the data center with the metadata server(s), users use the data center as the key-value store system. Data retrieval ("Get(key)"), range query ("Get(key1, key2)"), store ("Put(key, value)") and delete ("delete(key)") operations are supported. Our goal is to design a key indexing scheme to store key-value pairs among N Kinetic Drives, so that the following requirement should be satisfied.

- It can handle dynamic generated key-value pairs stored in the system.
- Limited number of drives should be involved for a given key.
- The indexing table should be in small scale so that it can be stored in the metadata server(s).
- It should avoid too much data migration among drives.
- Drives should cover as few key ranges as possible.
- The system should lower the percentage of active disks.

V. Our Design

A. Assumptions

In this paper, we have the following reasonable assumptions.

1) We assume the key-value pairs generated by users are received by the system dynamically. In other words, we generally do not know what the upcoming key-value pairs are, so that future key distribution is unknown.
2) We can simply assume all the Kinetic Drives are uniform and have the same storage capacity.
3) To simply the problem, we assume there is only one synchronized key indexing table in the metadata server. In reality, there may be multiple tables distributed in different metadata servers. However, in our problem, the synchronization of these tables is not an issue.
4) We assume the total Kinetic Drives can store all the key-value pairs received. If current drives do not have enough storage space, more drives can be added and our approach can also apply.

Key range	IP address of the drive
all 0s to 001...	138.32.211.4
010... to 011...	138.32.211.5
100... to 101...	138.32.211.6
...	...

Table II: An example of key indexing table

B. Key Indexing Table

The essential design is the key indexing table. This table maps the key-value pairs to the drives. As discussed in the previous section, given the key in the key-value pair, the metadata server quickly looks up the key indexing table and find out where this key-value pair is (or will be stored), then the IP address of that drive will be returned to the user.

The key indexing table should be designed in a simple and efficient way. This table should not require too much storage space. Also, the search for drives should be performed quickly. In our design, we use the key range to map the key-value pairs to the drives, explained as follows.

A key is essentially a sequence of multiple bits, consisting of 0 and 1. In our approach, We allocate the key-value pairs to the drives based on the key ranges.

There are two different ways to design the key indexing table, non-overlapped (disjoint) key ranges or overlapped (joint) key ranges in drives. Non-overlapped key ranges mean that each drive store key-value pairs with different non-overlapped key ranges. In other words, given a key, there is only one drive that can potentially have that key-value pair (if we ignore multiple data copies issue here). On the other hand, overlapped key ranges mean that different drives can cover the key-value pairs with overlapped key ranges. Hence, give a key, multiple drives need to be searched to find that key-value pair.

In our design, we generally consider the non-overlapped case. In this way, the metadata server only returns one Kinetic Drive's IP address to the user given a key. The user therefore, only needs to contact one drive. If the overlapped case is adopted, however, multiple drives can be searched because it is possible that the given key falls in the overlapped key range among different drives. Hence, the valuable disk resources and bandwidth would be wasted. However, the overlapped method can delay and reduce the data migration amount when one drive is full. We will discuss this later in this paper and also design an approach allowing overlapped key ranges to show the tradeoff.

Here is an example of the key indexing table shown in Table 1, assuming the maximum number of bits in the key is 10. From this example, we can see that the drive with the IP address "138.32.211.4" is used to store the key-value pairs with key range from all 0s to 001...(keys starting with 001). Hence, if a key falls into this range, the key-value pair will be stored and retrieved in that drive. In the next paragraphs, we will see that a drive can be used to store key-value pairs in multiple key ranges.

C. Initial Assignment and Further Adjustment

At the beginning, all drives are empty. As the key-value pairs come into the system, we need to assign those data to the drives. Hence, certain mappings between the key-value pairs to the drives need to be created to allocate the data accordingly, even before the system receives any key-value pairs. We call this process as "initial assignment" because the initial mappings are created.

As time goes on, more and more key-value pairs will be stored in the drives. Since the key-value pairs are dynamically received and the key distribution is generally unknown, the initial assignment will not be applicable for future data in later stage. Some drives may have more data while others may have less data according to the initial assignment. At certain time, some drives will become full and cannot store any new data. For example, based on Table 1, if the data in the key range (all 0s to 001...) become more and more as time goes on, the drive "138.32.211.4" will be full at some moment. Then this drive cannot store the data in this range any more. Hence, we need to further adjust the mappings and update the key indexing table.

Intuitively, when a drive is full, we need to migrate some data to another drive, freeing some storage space for new upcoming key-value pairs. (Generally, we do not simply redirect future data in this full drive's key range to another drive. This would cause multiple drives have overlapped key ranges. We have a separate approach allowing overlapped key ranges.) In that case, the key indexing needs to be split and updated. For example, if the drive "138.32.211.4" is full, the index "all 0s to 001..." will be split into two parts, e.g., "all 0s to 0010..." and "0011...". One of these two new key index range will point to a new drive and key-value pairs within this range will be migrated to that new drive accordingly. Table 2 shows the new possible key indexing table after the adjustment. The data within the key range "0011..." are now stored in drive "138.32.211.7". Hence, we can see that it is necessary to further update the key indexing table and migrate data.

The initial assignment and further adjustment are highly related. We should consider the further adjustment when we design the initial assignment scheme. A good initial assignment can reduce the data migration amount for the further adjustment later and the number of key ranges for each drive. In the follow subsection, we discuss the importance of the initial assignment and show our approaches. Also, we consider the percentage of active disks as well.

D. A "OneToAll" Approach

To better explain our approaches, we denote all the drives with the number 0, 1, 2, 3, ... In this "OneToAll" approach, all the key-value pairs are assigned to only drive 0 initially. In that case, the drive 0 stores the data in any key range

Key range	IP address of the drive
all 0s to 0010...	138.32.211.4
0011...	138.32.211.7
010... to 011...	138.32.211.5
100... to 101...	138.32.211.6
...	...

Table III: An example of the new key indexing table after further adjustment

(i.e., all 0s to all 1s), as shown below. Now there is only one key index entry in the table.

$$\text{all 0s to all 1s} \longrightarrow \text{drive 0}$$

As time goes on, the drive 0 will be full. Then we move half of the data from drive 0 to drive 1 and split the key index into two drives. Those half of the data are decided based on the ascending order of the keys. In other words, we separate the data into two parts with the same amount. One part of key-value pairs are those with smaller keys, whereas the other part of data are with greater keys. (Without losing generality, we keep the first half of data in drive 0 and move the other half to drive 1).

For Kinetic Drives, it is technically feasible to separate the data into two parts with smaller and greater keys. The Kinetic Drives support an API "GetPrevious(key)" to get the previous key-value pair based on the current given key. Hence, with the known ending key, the drive can easily find half of the data with greater keys using this API.

After that, the key index will be split into two entries, such as (This key range cutoff is only an example. The actual separation depends on the the key distribution and the value size.)

$$0... \longrightarrow \text{drive 0}$$
$$1... \longrightarrow \text{drive 1}$$

In this case, a new key-value pair with its key starting from "0" will be stored in drive 0. Hence, data will be put into two drives. With more and more data come in, at some moment, either drive 0 or drive 1 will be full. For example, if drive 1 is full, we split the key index for drive 1, move half of the data from drive 1 to drive 2, and update the table. Then the key indexing table may look like as follows.

$$0... \longrightarrow \text{drive 0}$$
$$10... \longrightarrow \text{drive 1}$$
$$11... \longrightarrow \text{drive 2}$$

After that, a new key-value pair will be stored in one of these 3 drives based on its key. We follow the same way for the subsequent upcoming key-value pairs. From the discussion, we can see that the essential idea of this approach is that, any upcoming new key-value pair will be stored in one of the current occupied drive. If one drive is full, then half of the data is moved to a new empty drive, and the key indexing is split and updated. Hence, the system stores the data starting with only **one** drive and gradually extend to **all** drives.

For this "OneToAll" approach, since data are initially assigned to only one drive, this approach leads to a lot of unnecessary data movement in some circumstances. If the key distribution of the entire key-value pairs is roughly uniform, this approach is not a good choice. Here, when we say "uniform" distribution, we mean that there are roughly the same amount of key-value pair data in each prefix of the key. For example, in each prefix "0000000", "0000001", "0000010", ... , "1111111", the data amount are roughly the same. In this scenario, this "OneToAll" approach still migrates data from one drive to another, without taking advantage of this uniform distribution.

However, on the other hand, this "OneToAll" approach yields a small percentage of active disks. The active and occupied disk drives are kept as few as possible. If the workload comes lightly and the disk bandwidth can sustain the users' requests, this approach actually is a good choice for lowering the percentage of active disks.

Another advantage of this approach is that, it can guarantee each drive covers data in only one key range. In other words, there is only one key range for each drive. This property can keep drives from storing data in more than one consecutive key range. It can greatly reduce the data management complexity. The more key ranges are covered in one drive, the more complexity it will bring. Also, it will cause the indexing table to be larger. Reducing number of key ranges in drives is especially helpful for key range search. Given a key range search request, it is better to involve as few drives as possible for searching data. If every drive stores data in only one key range as this "OneToAll" approach does, it is ideal for key range query in terms of the number of included disk drives.

In addition, the approach is also scalable for adding more disk drives. Since data are migrated from one drive to another, it still applies when more drives are added to the system if current disk drives are not enough. There is no change for this approach.

E. Two Additional Approaches

1) Approach 1: "Prefix-All" approach: From our previous discussion, we can see that data migration is inevitable. It takes network bandwidth and drives' resources. Hence, a good key indexing approach should reduce the data migration and consider the tradeoff among different factors. Our first additional approach "Prefix-All" works as follows.

Initial Assignment: Instead of filling the drives one by one with data, we assign the key-value pairs initially based on the prefixes of the keys. Given a key, we check the prefix of the key and decide which drive this key-value pair should be stored. The length of the prefix is determined by the number of drives in the data center. If there are N drives in total, we use the first $log_2 N$ bits of the key as the prefix to

decide the location of the data. As an example, we suppose there are 128 Kinetic Drives, then we use the first 7 bits of the key as the prefix. Hence, the initial key indexing table is as follows.

$$\begin{gathered}0000000... \longrightarrow \text{drive } 0\\ 0000001... \longrightarrow \text{drive } 1\\ 0000010... \longrightarrow \text{drive } 2\\ 0000011... \longrightarrow \text{drive } 3\\ ... \longrightarrow ...\\ 1111110... \longrightarrow \text{drive } 126\\ 1111111... \longrightarrow \text{drive } 127\end{gathered}$$

In the above table, "0000000..." means the key starting with "0000000". We can see that at the initial assignment stage of this approach, based on the prefixes of the keys, the data are separated into all the drives in advance. Different from the previous case, the key indexing table is predetermined at the first place, whereas the "OneToAll" approach gradually expands the key indexing table. In this approach, we just assign the key-value pair to the corresponding drive based on the first some bits.

Data Migration: Later, when a drive becomes full, we will move part of the data in that drive out to another drive. Here, we use a greedy way to pick up the destination drive. We select the drive with the maximum free storage space. Then, we fill the chosen destination drive with data of half of its free storage space, from the full drive. (We can move the data with greater keys out.) After that, we split the key index and update the table. (If we cannot find a destination drive that has at least half of its storage capacity as free storage space, it is the time to add more new empty drives.) In the above example, if drive 0 is full and drive 126 has the maximum free storage space, then the updated key indexing table may be as follows.

$$\begin{gathered}00000000... \longrightarrow \text{drive } 0\\ 0000001... \longrightarrow \text{drive } 1\\ 0000010... \longrightarrow \text{drive } 2\\ 0000011... \longrightarrow \text{drive } 3\\ ... \longrightarrow ...\\ 1111110... \text{ and } 00000001... \longrightarrow \text{drive } 126\\ 1111111... \longrightarrow \text{drive } 127\end{gathered}$$

Since this approach assign the data based on the key prefix, it can deal with the keys in uniform distribution well. With predetermined data allocation, key-value pairs can directly go to the corresponding drives without too much data migration, since data are roughly the same for each prefix. Hence, the data migration can be reduced. Also, this approach can make sure each drive has at most two different key ranges. However, the percentage of active disks in this "Prefix-All" approach can be very high. All drives can be potentially accessed and active at the beginning stage, since each drive is responsible for a certain key range.

Although this approach can also apply when more new empty drives are added, it is less flexible and scalable than "OneToAll" and the following "Prefix-Half" approach. All the drives are assigned with key ranges in advance and potentially used at the beginning. When some new drives are added, some drives may already have data in two different key ranges due to the data migration.

2) Approach 2: "Prefix-Half" approach: In the real scenario, all the key-value pairs are dynamically and continuously generated by users. Hence, the key-value store system generally has no prior knowledge about the future key distribution. The "Prefix-All" approach handles the uniform keys distribution perfectly in terms of data migration. However, in reality, the key distributions are not likely to be extremely uniform. Also, in order to consider the tradeoff between percentage of active disks and data migration amount, we have our "Prefix-Half" approach as follows.

Initial Assignment: The "OneToAll" approach uses only one drive as the initial assignment location, whereas the "Prefix-All" approach uses all the drives. In this "Prefix-Half" approach, in order to take care of different possible key distributions, we use half of the entire drives to initially store the data. In other words, half of the drives are spare drives and empty initially. They are not used to store data at the beginning.

For half of drives that are used for initial data assignment, we use the key prefix to decide the location of the data, which is similar to the "Prefix-All" approach. If there are N drives in total, we use the first $log_2 \frac{N}{2}$ bits of the key as the prefix to locate data. For example, if we have 128 drives, only 64 drives are used for data allocation at the beginning and the first 6 bits of the key are the indexes. The key-value pair whose the key starting with "000000" is stored in drive 0, as shown below.

$$\begin{gathered}000000... \longrightarrow \text{drive } 0\\ 000001... \longrightarrow \text{drive } 1\\ 000010... \longrightarrow \text{drive } 2\\ 000011... \longrightarrow \text{drive } 3\\ ... \longrightarrow ...\\ 111110... \longrightarrow \text{drive } 62\\ 111111... \longrightarrow \text{drive } 63\end{gathered}$$

Data Migration: As time goes on, more and more key-value pairs are received and stored in the drives. At some moment, one of the drive becomes full and cannot store new data anymore. When this happens, we move half of the data from this drive to an empty spare drive and split the key index into two parts. This is similar as the "OneToAll" approach shown in Section 5.4. We can keep the half of data with smaller keys in the drive and move the other half of data out.

For example, based on the previous key indexing table, if drive 0 is full. We can move half of the data with greater keys to drive 64, which is initially empty. Then the new key indexing table can be updated as follows.

0000000... $\longrightarrow$ drive 0
000001... $\longrightarrow$ drive 1
000010... $\longrightarrow$ drive 2
000011... $\longrightarrow$ drive 3
... $\longrightarrow$ ...
111110... $\longrightarrow$ drive 62
111111... $\longrightarrow$ drive 63
0000001... $\longrightarrow$ drive 64

Further Merging: From the previous discussion, we know that the initial empty spare drives are filled with data one by one. Later, at some moment, all the disk drives are occupied by data, i.e., no empty drives exist any more. When this happens and we find one of drives becomes full again, we then do the following further data merging.

Since the key distribution is not likely to be extremely uniform, some drives have more data while others have less data. In order to store upcoming key-value pairs, we further merge data among drives to create empty drives. If an empty drive can be created after some data merging, then we can use this newly created empty drive to store the upcoming key-value pairs, (i.e., we move half of the data with greater keys from the full drive to this newly created empty drive, and update the key indexing table.)

Here is how we merge data and create empty drives. We create the empty drives one by one and on demand. When all the drives are occupied with data and the current drive is full, we then create an empty drive. In order to do that, we select two drives with least amount of data whose key ranges are adjacent to each other. This adjacency means that those data in these two chosen drives can be merged as a consecutive key range (i.e., the start key of one key range is next to the end key of the other). If the data in these two drives can be merged into one drive, we do that. Then we merge the key index and update the index table as well. After that, an empty drive is created.

With this further data merging method, we not only create an empty drive, but also make sure data in two adjacent key ranges will be merged, so that after merging, data are still in one key range in the drive.

For example, based on the previous key indexing table, if drive 2 and 3 have least amount of data, we merge and migrate them into one drive, e.g., drive 3. Then drive 2 becomes an empty drive and can be used to store upcoming key-value pairs. After that, the new key indexing table is as follows.

0000000... $\longrightarrow$ drive 0
000001... $\longrightarrow$ drive 1
00001... $\longrightarrow$ drive 3
... $\longrightarrow$ ...
111110... $\longrightarrow$ drive 62
111111... $\longrightarrow$ drive 63
0000001... $\longrightarrow$ drive 64
... $\longrightarrow$ ...

In our scenario, we assume that the total Kinetic Drives have enough storage capacity to store all the key-value pairs. If later we cannot find an destination drive to store new data, we believe the system needs to add more drives, since the majority of the storage capacity has been used already. After new empty drives are added, we can then use our method again to store more data. Hence, this approach is also a scalable one.

Compared with the "Prefix-All" approach, with small amount of extra data migration caused by further merging, more drives store data with consecutive key ranges. Also, we can see that this approach can reduce the percentage of active disks. At the beginning, only half of the drives are potentially active and involved for data access. Then the system gradually expands its active disk set on demand. Hence, this approach considers the tradeoff between percentage of active disks and data migration amount. It is in the middle of "OneToAll" and "Prefix-All" approaches and provides a flexible data allocation scheme.

F. "Prefix-Half-2Drives" approach

In the previous section, we discuss the tradeoff among the number of involved disks for a request and data migration. In order to reduce the data migration amount, we can design an approach that allows disks have overlapped key ranges. When a disk A becomes full, instead of immediately split the index and move part of the data out, we can redirect future data in disk A's key range to another disk B. In that case, at this moment, data migration can be delayed or even avoided. However, disk B's key ranges are now extended and have an overlap with A.

Of course, we need to consider the performance of processing a key-value request. Given a key, the system should not search too many drives for data. If a lot drives have overlapped key ranges, the number of involved disks searched by the system for data could become huge, which greatly impacts the performance and wastes the disk resources. Hence, in our approach, we limit the number of drives searched for data up to 2. In other words, given a key, the system searches at most two drives for data.

We design this approach based on the "Prefix-Half" approach. (In the "Prefix-All" approach, there are no spare drives at the beginning.) We take advantage of the spare empty drives. Here is how it works.

Similar to the "Prefix-Half" approach, the first half of the drives are assigned with predetermined key ranges and the other half drives are spare empty drives. In addition, the first half drives are grouped in pairs. If disk A is in pair with B, we call A and B are neighbor drives. When we group the neighbor drives in pairs, we consider the key ranges covered by them. Two disks are neighbors only when they cover adjacent key ranges. Each drive only has one neighbor drive in our case, (since we limit the number of drives searched for any data up to 2.) When the drive A is full and data

in A's key range is coming, we first consider its neighbor drive B as the destination. If its neighbor drive B still has storage capacity for data, we store the new data (which was supposed to store in A) in drive B, and update the indexing table for drive B. For example, if the original indexing table is as follows, (drive 0 and 1, 2 and 3, 62 and 63 are grouped as neighbors.)

$$\begin{gathered} 000000... \longrightarrow \text{drive } 0 \\ 000001... \longrightarrow \text{drive } 1 \\ 000010... \longrightarrow \text{drive } 2 \\ 000011... \longrightarrow \text{drive } 3 \\ ... \longrightarrow ... \\ 111110... \longrightarrow \text{drive } 62 \\ 111111... \longrightarrow \text{drive } 63 \end{gathered}$$

then drive 0 becomes full. In this approach, we redirect future data in 000000... to drive 1 first, if drive 1 still has storage space. Hence, at some moment, the indexing table may look like this. We can see that drive 0 and drive 1 now have an overlapped key range 0000001..., and any data in this range will be searched in both drive 0 and 1.

$$\begin{gathered} 000000... \longrightarrow \text{drive } 0 \\ 0000001... \text{ and } 000001... \longrightarrow \text{drive } 1 \\ 000010... \longrightarrow \text{drive } 2 \\ 000011... \longrightarrow \text{drive } 3 \\ ... \longrightarrow ... \\ 111110... \longrightarrow \text{drive } 62 \\ 111111... \longrightarrow \text{drive } 63 \end{gathered}$$

As data continue comes, drive B may become full later, then we migrate data. In order to do that, we bring one of the spare empty disk drive C and reorganize data among disk A, B and C. We balance the data among these three drives and each drive stores data with 2/3 of its storage capacity. (We need to balance data amount of two full drives among three drives.) When we do that, each drive is assigned data with one consecutive key range and three drives cover the entire original key ranges of A and B one after one, (e.g., Drive A, B and C can store the first, second and last 1/3 of the key ranges respectively.) This way can make sure after the data migration, these three drives have no overlapped key range and yield a good data occupied rate of 2/3.

In the above example, after the data migration, the indexing table may look as follows.

$$\begin{gathered} 0000000... \longrightarrow \text{drive } 0 \\ 000001... \longrightarrow \text{drive } 1 \\ 000010... \longrightarrow \text{drive } 2 \\ 000011... \longrightarrow \text{drive } 3 \\ ... \longrightarrow ... \\ 111110... \longrightarrow \text{drive } 62 \\ 111111... \longrightarrow \text{drive } 63 \\ 0000001... \longrightarrow \text{drive } 64 \end{gathered}$$

With data redirection allowing overlapped key ranges, we can see that data migration can be delayed or even avoided at some cases. If all the spare empty drives are occupied, new drives need to be added in the system.

G. A Special Case: How to handle roughly known key distributions ("Prefix-Half-known" approach)

In our previous approaches, we assume that the system has no prior knowledge of the key distributions. In some cases, however, we may know some rough key distributions in advance. In other words, we may have a prior knowledge about some general key distributions. This information is only rough and not very accurate, but can reflect the general distributions of keys. Suppose there are 256 Kinetic Drives in the data center, the granularity of this known key distribution is not precise enough in every single drive (i.e., we do not know the percentages of data in every one of the 256 key ranges). Instead, we may only know approximately the percentages of data in each of 16 key ranges. (Each of these 16 key ranges are obviously bigger than each of 256 key ranges.) For example, the system may only roughly know the percentages of data amount in key range of 0000..., 0001..., 0010..., ..., 1111... respectively. However, in the key range 0000..., it still does not have a knowledge of percentages of data distribution in its sub-ranges. In other words, this prior knowledge is only to the level of 16 bigger key ranges generally.

Here, we design an approach that can take advantage of this general and rough known knowledge of key distributions. We propose a solution based on the "Prefix-Half" approach with changes. It works as follows. Suppose we have N Kinetic Drives and M key ranges ($N >> M$). We know that each range $R_i (i = 1, 2, ..., M)$ has a percentage P_i of total data. In the "Prefix-Half" approach, we equally divide the entire key range with the number of half of the drives ($N/2$) and assign each drive a predetermined key range. However, in this approach, since we know the percentages P_i for each R_i, we use the following way.

We still keep half of the drives as empty spare drives and divide the rest of them into M parts. The number of drives in each part $U_i (i = 1, 2, ..., M)$ is proportional to the percentage P_i. Each part of drives U_i is used to store data in the corresponding key range R_i initially. Within U_i, we further equally divide the key range R_i into $|U_i|$ sub-ranges, and each drive in U_i is responsible for storing data in each sub-range initially.

The rest of this approach is similar to the "Prefix-Half" approach. When a drive is full, data in that drive are split into two parts and one part is moved out to an empty spare drive. When all the drives are filled with data, we further merge data among the drives and update the indexing table as shown in Section 5.5.

From the above discussion, we can see that, with some prior rough knowledge of key distribution, drives can initially store data with different key coverage ranges based on the

various percentages. Hence, a more accurate data allocation approach is proposed.

VI. Performance Evaluation

A. Experiment Setup

To evaluate the performance of our approaches, we simulate a data center environment with 256 Kinetic Drives. Each drive has the storage capacity of 2TB. We generate key-value pairs in different key distributions as shown in the follow paragraphs. Each key-value pair is about 1MB size. The total key-value pairs generated and stored in the data center is about half of the entire storage space of the data center.

B. Performance Comparison

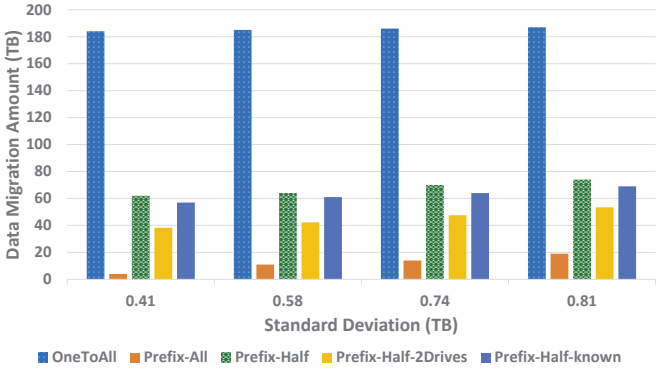
We use the total data migration amount and the average percentage of active disks (i.e., disks occupied with data), as the performance metrics. We compare our five approaches. For each key distribution, we keep the same amount of key-value pairs.

1) Lightly Unbalanced Distribution: From our previous discussion, we can see that the "Prefix-All" approach can handle uniform key distribution well. Ideally, if the keys are uniformly distributed for all the key ranges, there are zero data migration. Key-value pairs can directly be stored in their predetermined drives. Also, in this ideal uniform case, every drive stores data only in one consecutive key range, which is perfect.

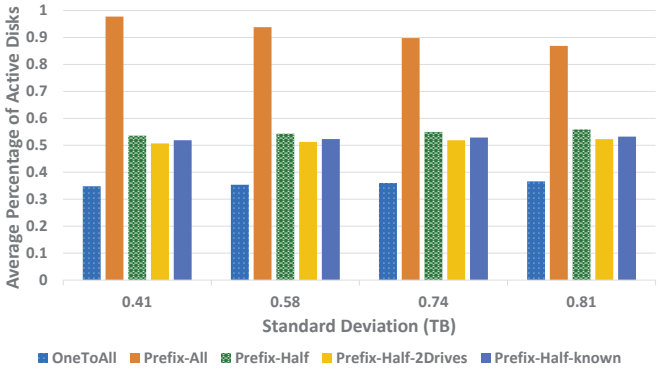
However, in reality, this perfect uniform key distribution does not happen. There are more data in some key ranges, whereas there are less data in other key ranges. The difference may be not very large. In order to reflect this lightly unbalanced key distribution among different key ranges, we generate the data with their keys in normal distribution. Among these 256 key ranges, some of them have more data while others have less data. Overall, the data amount among all the key ranges roughly follows a rough normal distribution with its average of 1GB. We generate the traces with different standard deviations to compare the performance.

This type of traces can simulate and reflect some cases that there are different data amount in different key ranges. It is a typical case that some key ranges are popular while others are not. With the normal distribution, the lightly unbalanced data amount can be captured, but it does not fully reflect some highly skewed key distribution. In other words, we use the normal distribution to reflect the various popularities and unbalance in some moderate degree. Although there are some difference of data amount among the key ranges, the overall variance does not highly differ.

Figure 5 shows the results. We vary the standard deviation of data among key ranges in 0.41TB, 0.58TB, 0.74TB and 0.81TB respectively. In Figure 5(a), we can see that the "Prefix-All" approach outperforms the others in terms of



(a) Data migration amount



(b) Average percentage of active disks

Figure 5: Comparison among different approaches - lightly unbalanced distribution

data migration. Although data are in normal distribution, they are not highly skewed. With lightly unbalanced data amount in different key ranges, the "Prefix-All" approach can deal with it well. Based on the prefix of the keys, most of the data do not need to be migrated after they have been initially stored in drives.

Meanwhile, we can see that the "OneToAll" approach has the largest data migration amount, because it does not take advantage of this key distribution. It just migrate data to drives one after another. The data migration of "Prefix-Half" is obviously between that of "OneToAll" and "Prefix-All". Also, compared with "Prefix-Half", the "Prefix-Half-2Drives" and "Prefix-Half-known" approaches can reduce the data migration. The "Prefix-Half-2Drives" can delay and even decrease the data migration because neighbor drives can store some data without being migrated. With better knowledge of key distribution, the "Prefix-Half-known" can lead to smaller data migration amount. When the standard deviation increases, the data migration amount also becomes larger for all approaches. This is because if data are more unbalanced among key ranges, the data migration happens more frequently.

For the percentage of active disks in figure 5(b), we can

find that the ”OneToAll” approach has the smallest value. This is because data are first stored in one drive, then gradually migrated to other drives one by one. The ”Prefix-All” approach yields the highest average percentage of active disks, because even at the early stage, a lot of disks receive data in their key ranges. The ”Prefix-Half”, ”Prefix-Half-2Drives” and ”Prefix-Half-known” approaches have average percentage of active disks in the middle.

From the above discussion, we can find that if the data are lightly unbalanced among key ranges, the ”Prefix-All” approach has small data migration, but large average percentage of active disks. There is a tradeoff between these two design factors. Also, if we allow overlapped key ranges in two different drives (which ”Prefix-Half-2Drives” does), the data migration amount can be reduced.

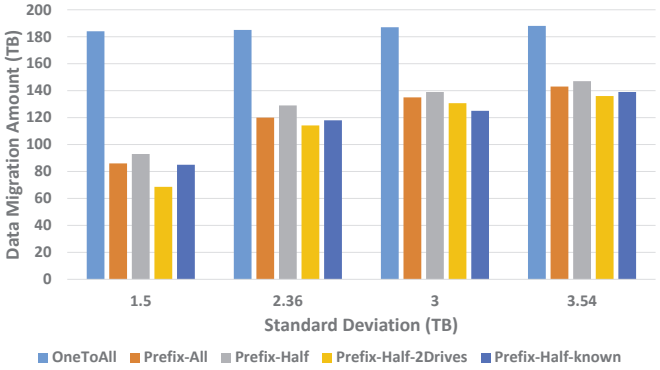
2) Highly Skewed Distribution: We generate key-value pairs with keys in highly skewed distribution. In some scenarios, there are a lot of data in certain key ranges, while in some other ranges, there are little or even no data. Depending on the key generation method, some key ranges may never be used, whereas some other key ranges are very popular and many applications generate high volume of key-value pairs within those ranges.

We vary the number of key-value pairs among different key ranges exponentially. (In other words, the number of data among different key ranges change exponentially. We use that to reflect this highly unbalanced case.) We generate multiple traces with different standard deviations 1.5TB, 2.36TB, 3TB and 3.54TB. Compared with previous lightly unbalanced key distribution, the data here are more skewed.

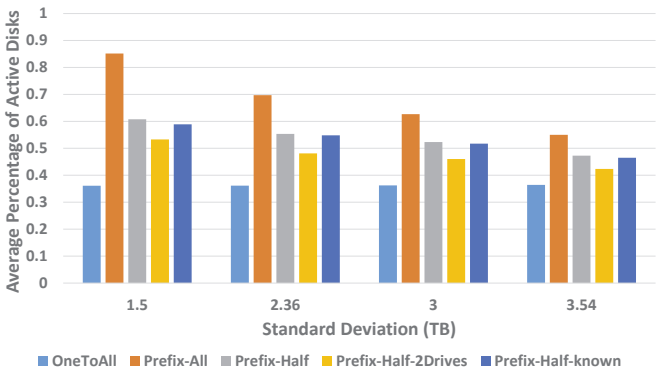
Figure 6 shows the results. Still, the ”OneToAll” approach has the largest data migration amount. The ”Prefix-Half” approach has more data migration amount than ”Prefex-All” approach, but the difference is much smaller than that in lightly unbalanced key distribution shown in Figure 5(a). When data are highly skewed, both of these two approaches have to migrate a large amount of data. When data becomes more and more unbalanced, data migration amount in all approaches are larger. Also, we can see that ”Prefix-Half-2Drives” and ”Prefix-Half-known” approaches can reduce the data migration amount.

For the average percentage of active disks shown in figure 6(b), we have the following observations. As data become more unbalanced, the average percentage of active disks decreases. This is because more drives have no data in their predetermined key ranges when the key distribution are more unbalanced. Also, we can see that the ”Prefix-All” approach has the largest average percentage of active disks while ”OneToAll” yields the smallest value.

For the highly skewed key distribution, we still can see the tradeoff between data migration amount and average percentage of active disks. Generally, the ”Prefix-Half” is a good choice. With limited extra data migration, it can reduce the average percentage of active disks and provides a more flexible and scalable storage scheme.



(a) Data migration amount



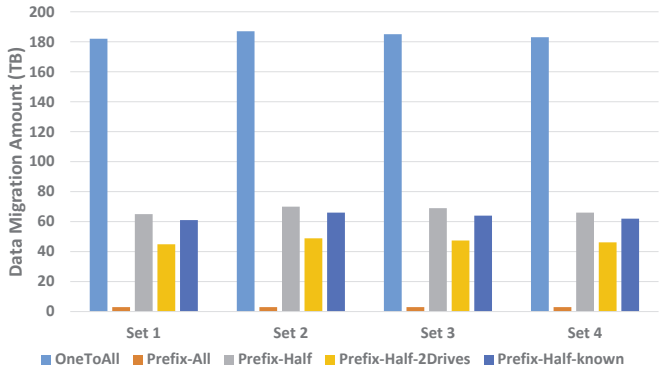
(b) Average percentage of active disks

Figure 6: Comparison among different approaches - highly skewed distribution

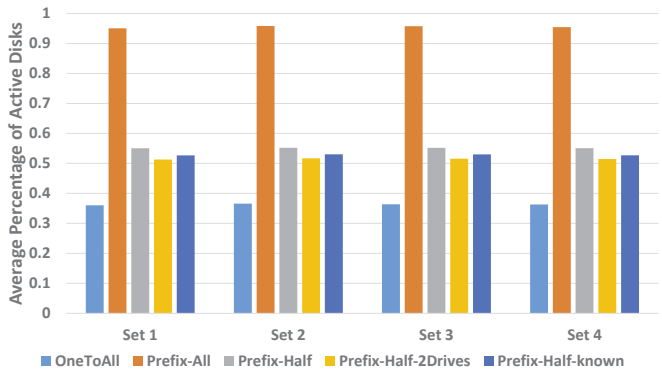
3) Random Distribution: In addition to the above distributions, we also generate keys following random distribution. Among all the key ranges, the data amount are totally random. (The total key-value pairs are still kept the same as previous distributions.)

This random distribution can reflect the random key generation method. For example, the random hashing function is widely used to generate the keys. By simply hashing the data, a key can be created. Due to the hashing function, a predetermined key distribution cannot be guaranteed. Hence, we use random distribution to reflect this scenario.

We conduct multiple experiments and select four sets of results as shown in Figure 7. In Figure 7(a), we can see that for data migration, the ”OneToAll” approach is still the worst, while the ”Prefix-All” approach yields the best result. The other three ”Prefix-Half” related approaches are in the middle. For the average percentage of active disks shown in figure 7(b), it becomes opposite. The ”OneToAll” approach is the lowest, while ”Prefix-All” approach has the highest value. The ”Prefix-Half-2Drives” and ”Prefix-Half-known” approaches can also have some improvement compared with ”Prefix-Half”.



(a) Data migration amount



(b) Average percentage of active disks

Figure 7: Comparison among different approaches - random distribution

To sum up, we can generally see the tradeoff between data migration amount and average percentage of active disks. Also, with overlapped key ranges in two drives, the data migration amount can be reduced. With better knowledge of key distribution, the performance can be improved as well. According to the practical factors, limitations and environment, we can choose different approaches for use.

VII. Conclusion

As one of the popular NoSQL databases, key-value store provides simple, flexible and efficient storage paradigm. Recently, a new storage device called "Kinetic Drive" was invented. Kinetic Drives can perform key-value operations by themselves given the keys. This new innovation greatly changes the storage stack.

In this paper, we investigate the data allocation of large-scale key-value store system in a data center using Kinetic Drives. We show the tradeoff among different design factors. We design the key indexing schemes and propose approaches for data allocation and migration among Kinetic Drives. Performance evaluation shows the results of different approaches for various key distributions.

References

[1] Nosql databases, http://nosql-database.org/.

[2] C. Strauch, U.-L. S. Sites, and W. Kriha, "Nosql databases," *Lecture Notes, Stuttgart Media University*, 2011.

[3] M. Seeger and S. Ultra-Large-Sites, "Key-value stores: a practical overview," *Computer Science and Media, Stuttgart*, 2009.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[6] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208479

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

[8] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *2005 IEEE International Symposium on Mass Storage Systems and Technology*. IEEE, 2005, pp. 119–123.

[9] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," *SIGPLAN Not.*, vol. 33, no. 11, pp. 81–91, Oct. 1998. [Online]. Available: http://doi.acm.org/10.1145/291006.291026

[10] H. Lim, V. Kapoor, C. Wighe, and D. H.-C. Du, "Active disk file system: A distributed, scalable file system," in *Mass Storage Systems and Technologies, 2001. MSS'01. Eighteenth IEEE Symposium on*. IEEE, 2001, pp. 101–101.

[11] E. Riedel, C. Faloutsos, and D. Nagle, "Active disk architecture for databases," Tech. Rep., 2000.

[12] Kinetic open storage documentation wiki. https://developers.seagate.com/display/KV/Kinetic+Open+Storage+Documentation+Wiki.

[13] Kinetic hdd, http://www.seagate.com/enterprise-storage/hard-disk-drives/kinetic-hdd/.

[14] The seagate kinetic open storage vision, http://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/.

[15] http://www.seagate.com/.

[16] Leveldb, leveldb.org.

[17] M. Minglani, J. Diehl, X. Cao, B. Li, D. Du, and D. Lilja, "Kinetic action: Application level performance analysis of kinetic drives," Submitted to FAST 2017, Tech. Rep.

[18] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE J.Sel. A. Commun.*, vol. 22, no. 1, pp. 41–53, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2003.818784

[19] A. I. T. Rowstron and P. Druschel, "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*, ser. Middleware '01. London, UK, UK: Springer-Verlag, 2001, pp. 329–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=646591.697650

[20] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 46–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=371931.371977

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "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*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071

[22] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The oceanstore prototype," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, ser. FAST '03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1090694.1090696

[23] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2318857.2254766

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043558

[25] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "Nvmkv: A scalable, lightweight, ftl-aware key-value store," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 207–219. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813767.2813783

[26] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 429–444. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616488

[27] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342360