

**Efficient Data Management and Processing in Big Data
Applications**

**A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY**

Xiang Cao

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY**

David Hung-Chang Du

May, 2017

© Xiang Cao 2017
ALL RIGHTS RESERVED

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in the PhD study. First, I sincerely thank my advisor Professor David Hung-Chang Du for his continuous support during these years. Professor Du is a great mentor who has patiently enlightened and encouraged me all the time. I have learned a lot from his guidance along the way. His advice always made me have a better understanding of my research. I could not succeed in completing this thesis without his generous help. I truly believe my experience working with Professor Du will be tremendously beneficial for my future career.

I would like to thank Professor Tian He, Professor Rui Kuang and Professor David Lilja for serving as my thesis committee members, and their valuable feedback and comments.

I also thank Professor Yingfei Dong from University of Hawaii for his guidance and collaboration in my early stage of the PhD study. I thank current and former members of our research group, including but not limited to, Zhichao Cao, Jim Diehl, Hebatalla Eldakiky, Ziqi Fan, Xiongzi Ge, Alireza Haghdoost, Weiping He, Xiaoxiao Jiang, Bingzhe Li, Manas Minglani, Kewal Panchputre, Hao Wen, Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and Meng Zou. Thanks for their collaboration and discussion. I learned so much from them. I want to express my best wishes and thanks for many friends I have met in University of Minnesota during these years. Thanks for their help and support.

I would like to acknowledge the National Science Foundation (NSF) and Center for Research in Intelligent Storage (CRIS), for providing the sources of funding. I thank the Department of Computer Science and Engineering for the support of teaching assistantships, and its staff for their help.

Finally, I want to express my sincere gratitude to my family and relatives for their support. Especially, I thank my wife and parents for their love, understanding and encouragement during my PhD study.

Dedication

To my family and friends who have helped, supported and encouraged me along the way.

Abstract

In today’s Big Data applications, huge amount of data are being generated. With the rapid growth of data amount, data management and processing become essential. It is important to design efficient approaches to manage and process data. In this thesis, data management and processing are investigated for Big Data applications.

Key-value store (KVS) is widely used in many Big Data applications by providing flexible and efficient performance. Recently, a new Ethernet accessed disk drive for key-value pairs called “Kinetic Drive” was developed by Seagate. It can reduce the management complexity, especially in large-scale deployment.

It is important to manage the key-value pairs and store them in Kinetic Drives in an organized way. In this thesis, we present data allocation schemes on a large-scale key-value store system using Kinetic Drives. We investigate *key* indexing schemes and allocate data on drives accordingly. We propose efficient approaches to migrate data among drives.

Also, it is necessary to manage huge amount of key-value pairs to provide attributes search for users. In this thesis, we design a large-scale searchable key-value store system based on Kinetic Drives. We investigate an indexing scheme to map data to the drives. We propose a *key* generation approach to reflect metadata information of the actual data and support users’ attributes search requests.

Nowadays, MapReduce has become a very popular framework to process data in many applications. Data shuffling usually accounts for a large portion of the entire running time of MapReduce jobs. In recent years, scale-up computing architecture for MapReduce jobs has been developed. With multi-processor, multi-core design connected via NUMALink and large shared memories, NUMA architecture provides a powerful scale-up computing capability.

In this thesis, we focus on the optimization of data shuffling phase in MapReduce framework in NUMA machine. We concentrate on the various bandwidth capacities of NUMALink(s) among different memory locations to fully utilize the network. We investigate the NUMALink topology and propose a topology-aware reducer placement

algorithm to speed up the data shuffling phase. We extend our approach to a larger computing environment with multiple NUMA machines.

Contents

| | |
|-------------------------------------------------------------------------|------------|
| Acknowledgements | i |
| Dedication | iii |
| Abstract | iv |
| List of Tables | ix |
| List of Figures | x |
| 1 Introduction | 1 |
| 2 Background and Related Work | 5 |
| 2.1 Background | 5 |
| 2.1.1 Preliminaries of Kinetic Drives | 5 |
| 2.1.2 Advantages of Kinetic Drives | 7 |
| 2.1.3 NUMA Architecture | 10 |
| 2.1.4 SGI UV 2000 | 11 |
| 2.2 Related Work | 12 |
| 2.2.1 NoSQL databases, Key-value Store and Kinetic Drives | 12 |
| 2.2.2 Data Processing in MapReduce and NUMA Machine | 15 |
| 3 Data Management of Key-Value Store System using Kinetic Drives | 17 |
| 3.1 Introduction | 17 |
| 3.2 Motivation | 19 |
| 3.3 Our Problem | 20 |

| | | |
|----------|---------------------------------------------------------------|-----------|
| 3.3.1 | Our Scenario | 20 |
| 3.3.2 | Research Issues and Challenges | 22 |
| 3.3.3 | Design Tradeoff and Goal | 24 |
| 3.4 | Our Design | 24 |
| 3.4.1 | Assumptions | 24 |
| 3.4.2 | Key Indexing Table | 25 |
| 3.4.3 | Initial Assignment and Further Adjustment | 26 |
| 3.4.4 | A “OneToAll” Approach | 28 |
| 3.4.5 | Two Additional Approaches | 30 |
| 3.4.6 | “Prefix-Half-2Drives” Approach | 35 |
| 3.4.7 | A Special Case: How to handle roughly known key distributions | 37 |
| 3.5 | Performance Evaluation | 38 |
| 3.5.1 | Experiment Setup | 38 |
| 3.5.2 | Performance Comparison | 39 |
| 3.6 | Conclusion | 44 |
| 4 | Kinetic Drives based Searchable Key-Value Store System | 46 |
| 4.1 | Introduction | 46 |
| 4.2 | Motivation | 48 |
| 4.3 | Our Problem | 49 |
| 4.3.1 | Research Issues and Challenges | 49 |
| 4.3.2 | Our Goal | 50 |
| 4.4 | Our Design | 50 |
| 4.4.1 | Framework | 50 |
| 4.4.2 | Key Generation | 52 |
| 4.4.3 | Indexing | 55 |
| 4.4.4 | Users Data Retrieval Request | 59 |
| 4.4.5 | Data Searches in the Drive | 62 |
| 4.4.6 | Discussion | 65 |
| 4.5 | Performance Evaluation | 66 |
| 4.5.1 | Experiment Setup | 66 |
| 4.5.2 | Number of Drives Searched | 67 |

| | | |
|----------|---------------------------------------------------------------------|-----------|
| 4.5.3 | Data Searches in the Drives | 69 |
| 4.6 | Conclusion | 70 |
| 5 | Data Processing in MapReduce with Scale-up NUMA Architecture | 71 |
| 5.1 | Introduction | 71 |
| 5.2 | Motivation | 74 |
| 5.3 | Problem Statement and Proposed Solution | 76 |
| 5.3.1 | Problem Statement | 76 |
| 5.3.2 | Proposed Solution | 77 |
| 5.4 | Extension to Multiple NUMA Machines | 78 |
| 5.4.1 | Problem | 78 |
| 5.4.2 | Our Solution | 79 |
| 5.5 | Performance Evaluation | 81 |
| 5.5.1 | Experiment Setup | 81 |
| 5.5.2 | Performance Comparison within one NUMA machine | 81 |
| 5.5.3 | Performance Comparison with multiple NUMA machines | 83 |
| 5.6 | Conclusion | 84 |
| 6 | Conclusion and Future Work | 85 |
| 6.1 | Conclusion | 85 |
| 6.2 | Future Work | 86 |
| | References | 88 |

List of Tables

| | | |
|-----|-------------------------------------------------------------------------|----|
| 3.1 | An example of key indexing table | 26 |
| 3.2 | An example of the new key indexing table after further adjustment . . . | 28 |
| 4.1 | Applications of Attributes Search Requests | 66 |
| 4.2 | Number of bits assigned for attributes | 67 |
| 5.1 | Data transfer bandwidth between memories in different locations | 75 |
| 5.2 | Simulation setup I (one single machine) | 81 |
| 5.3 | Simulation setup II (multiple machines) | 83 |

List of Figures

| | | |
|-----|-------------------------------------------------------------------------|----|
| 2.1 | Traditional vs. Kinetic Storage Stack [1] | 6 |
| 2.2 | Comparison between traditional and Kinetic key-value store systems . . | 7 |
| 2.3 | Comparison of Write throughput (Kinetic Drive vs. LevelDB server) [2] | 9 |
| 2.4 | SGI UV 2000 IRU System Components Example [3] | 11 |
| 2.5 | Simplified view of SGI UV 2000 IRU | 12 |
| 2.6 | Interconnection topology of blades | 13 |
| 3.1 | Key-value store system with Kinetic Drives | 21 |
| 3.2 | Comparison among different approaches - lightly unbalanced distribution | 40 |
| 3.3 | Comparison among different approaches - highly skewed distribution . . | 42 |
| 3.4 | Comparison among different approaches - random distribution | 44 |
| 4.1 | Framework of Kinetic Drives based Searchable Key-Value Store System | 52 |
| 4.2 | Comparison of average number of drives searched | 68 |
| 4.3 | Comparison of average data amount searched | 69 |
| 5.1 | Normalized data shuffling time among different schemes | 82 |
| 5.2 | Comparison of normalized data shuffling time (multiple machines) . . . | 84 |

Chapter 1

Introduction

We are in the Big Data [4][5] era and currently living in the digital world. In today's Big Data applications, huge amount of data are being generated. According to EMC Digital Universe with Research & Analysis by IDC [6], data is growing 40% yearly towards the next 10 years. With the rapid growth of data amount, it is very important to provide a platform to manage and process data. In other words, data management and processing become essential in Big Data research. Efficient data management and processing schemes should be investigated for Big Data applications, which is the goal of this thesis.

Among the huge amount of data, unstructured data become a significant portion [6]. In recent years, NoSQL databases [7][8] have been developed to provide more flexibility and better performance than the traditional relational databases, especially in Big Data applications. As an important NoSQL database, key-value store (KVS) [9] offers easy but efficient data storage and data management. In KVS, a record of data consists of a *key* and a *value*. A *key* is generated as an index to store, retrieve and delete the *value*. The *value* is the actual data that can be of any type, e.g., graph, video, web pages, numbers, etc. Users can access the *value* by giving its corresponding *key*, without going through complicated operations (e.g., join) or foreign *keys* as in the relational databases. Nowadays, many existing key-value store systems have been widely deployed for various applications, such as Redis [10], Amazon Dynamo [11] and LinkedIn Voldemort [12].

In data storage industry, Object-based Storage Device (OSD) has been introduced [13][14]. They can manage data as objects instead of the traditional block-based storage.

As an innovative example of OSD and active disks [15][16][17], a new disk drive called “Kinetic Drive” [1][18][19] was developed by Seagate [20] recently. An active disk means a disk with certain computing capability. Each Kinetic Drive has a built-in CPU, RAM and implemented LevelDB [21]. It can perform key-value pair operations independently. This greatly changes the data access and management in the drive. Traditionally, the key-value implementation is on a storage server that manages a set of block-based disk drives. The application has to go through a storage server that connects to a set of traditional disk drives via SAS connections. With Kinetic Drive, however, the application can directly access the key-value pairs in the drives via Ethernet connections, which becomes easy and flexible.

For the first part of the thesis, we consider the data management of key-value store system using Kinetic Drives. In Big Data applications, huge amount of data are generated. Obviously, many Kinetic Drives are needed to store the data. With continuously generated key-value pairs, data allocation becomes an issue, (i.e., Given a key-value pair, which drive should store that?). It is necessary to have a metadata server to manage those drives and to provide an indexing scheme for data allocation. Hence, it is important to map a large amount of key-value pairs to drives with an indexing table on the metadata server. It is not possible to map every key-value pair to a drive location since this will create an extremely large indexing table. Mapping *key* ranges to drives is feasible to reduce the size of the indexing table. We propose data allocation schemes for a large-scale key-value store system using Kinetic Drives. We show the tradeoff of various design factors, design efficient indexing schemes, and allocate key-value pairs to Kinetic Drives. We consider different *key* distributions and propose data migration approaches. With a small-size indexing table, users can get IP addresses of corresponding drives quickly.

In many applications, the actual data is often generated with its metadata. For example, when an X-ray image is produced, some metadata (e.g., image size, generator ID, time, date) are also available. These metadata are important attributes for the actual X-ray image. Users later may retrieve the image by providing part or all the attributes to search. Hence, it is critical to store the actual data (e.g., image) with its metadata (attributes) together. Although some other NoSQL databases [22][23][24][25][26] (e.g., document store, column store) support data attributes access similar to relational

databases, they often bring complicated operations and overhead. As a light-weight NoSQL database, key-value store, however, can bring more efficient performance. Traditionally, key-value stores usually do not support data attributes search very well. They simply store the actual data as a *value* and use a *key* to retrieve it, which can not fully capture the metadata information. In this thesis, we believe it is convenient and feasible to group the actual data and its associated metadata (attributes) together and store them as the *value* in a key-value pair. It is very important to design a large-scale searchable key-value store system that can reflect metadata and support attributes search for users. For the second part of the thesis, we focus on this design. We propose an indexing scheme to map key-value pairs to Kinetic Drives. Attributes search requests from users are also considered in our design. We investigate a *key* generation approach to capture the metadata information and support users' search requests.

To process data, MapReduce [27][28] has become more and more prevalent in many applications (e.g., data analytics, high performance computing) in the last ten years. It provides a framework to allow processing huge amount of data in parallel. In a typical MapReduce application, *map*, *shuffle* and *reduce* are three major phases. As an important operation in MapReduce, the *shuffle* phase usually takes a very long time to finish depending on many factors (e.g., network bandwidth, reducer placement) [29]. Hence, reducing shuffling time can greatly decrease the entire job running time in MapReduce applications.

Traditionally, MapReduce is assumed to be executed on scale-out inexpensive commodity machines. In recent years, some research work about MapReduce on scale-up architecture have been investigated. In scale-up architecture, a single machine is equipped with more powerful CPUs and larger memories. In [30], a 16-node scale-out cluster was compared with a scale-up server running MapReduce jobs. The experiment results showed that compared to a 16-node scale-out cluster, a scale-up server provided better performance per dollar. Research work in [31][32] also had similar results.

For the third part of the thesis, we particularly focus on the data shuffling in the MapReduce framework. We use the SGI UV 2000 machine [33] as an example to investigate data shuffling in scale-up NUMA architecture. Our goal is to reduce the data shuffling time in the MapReduce computing framework with the consideration of variations of different data transfer bandwidth based on different locations using NUMALink(s).

The rest of this thesis is organized as follows. Chapter 2 provides the background and related work. In Chapter 3, data management of key-value store system using Kinetic Drives is presented. Chapter 4 discusses the Kinetic Drives based searchable key-value store system. In Chapter 5, data processing in MapReduce with scale-up NUMA Architecture is shown. Finally, the conclusion of this thesis and future work are in the Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we discuss the background of the Kinetic Drives and NUMA architecture. Some related work are also presented.

2.1 Background

2.1.1 Preliminaries of Kinetic Drives

Kinetic Drives were recently invented by Seagate [1][18][19]. Compared with traditional disk drives, Kinetic Drives can be accessed by users via Ethernet connections instead of the typical SAS or ATA interface. Each drive provided by Seagate in our performance testings [2] has a storage capacity of 4TB and 2 Ethernet connections of 1Gb/s. Users can directly access a Kinetic Drive via its IP address through the Ethernet. The drives we tested 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): Storing the key-value pair.
- Get(key): Retrieving the key-value pair.
- GetKeyRange(key1, key2): Retrieving the key-value pairs in the *key* range between *key1* and *key2*.
- Delete(key): Deleting the key-value pair.

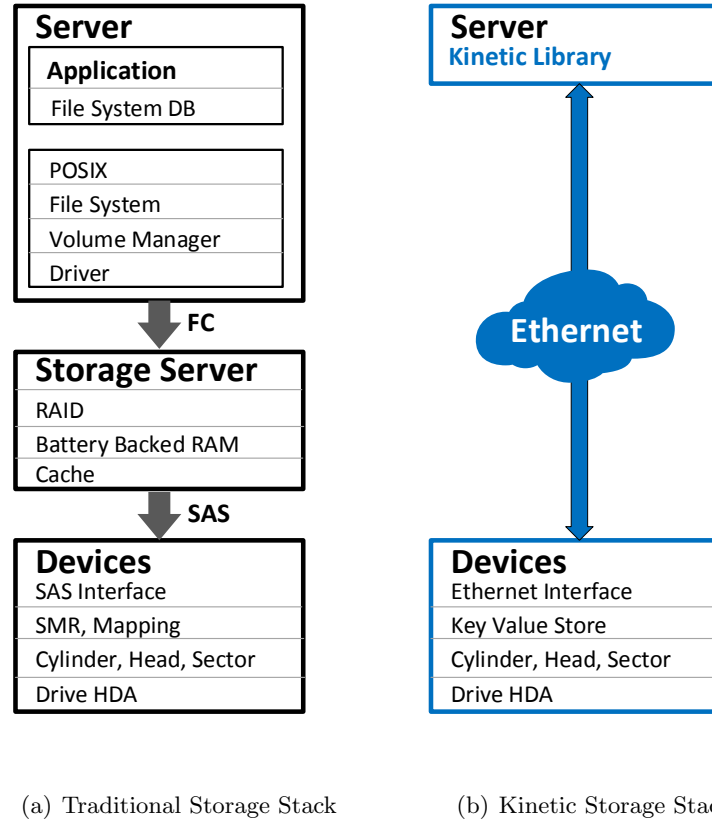
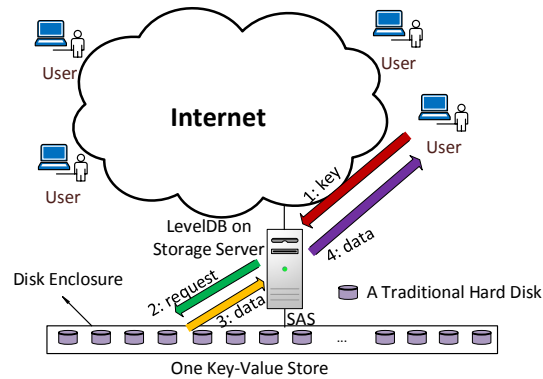


Figure 2.1: Traditional vs. Kinetic Storage Stack [1]

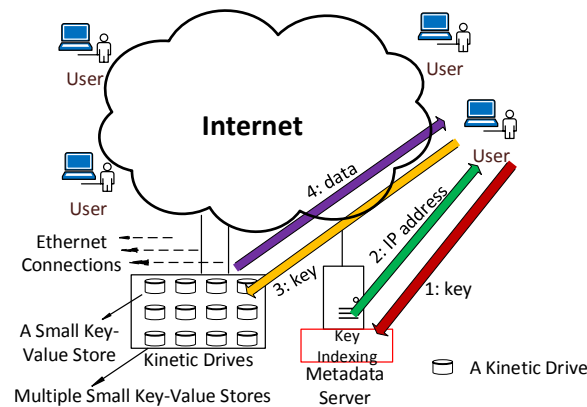
- `GetNext(key)`: Retrieving the next key-value pair based on the given *key*.
- `GetPrevious(key)`: Retrieving the previous key-value pair based on the given *key*.

Each Kinetic Drive has a built-in CPU, RAM and implemented LevelDB [21]. It can perform key-value pair operations independently. This greatly changes the data access and management in the drive. Traditionally, the key-value implementation is on a storage server that manages a set of block-based disk drives. As shown in Figure 2.1(a) [1], the application has to go through a storage server that connects to a set of traditional disk drives via SAS connections. With Kinetic Drive, however, the application can directly access the key-value pairs in the drives via Ethernet connections as shown in Figure 2.1(b) [1], which becomes easy and flexible.

2.1.2 Advantages of Kinetic Drives



(a) Traditional key-value store system



(b) New Kinetic key-value store system

Figure 2.2: Comparison between traditional and Kinetic key-value store systems

In key-value store systems, LevelDB is widely used and we use it as an example to show the difference between traditional drives and new Kinetic Drives. In the traditional architecture, LevelDB is deployed on a storage server with a set of traditional hard disks. The LevelDB stores data in key-value format in multiple levels. Key-value pairs are sorted and partitioned in each level. Data in multiple levels need to be accessed

from hard disks to the storage server, then transferred to users. When more key-value pairs are inserted, data in a level has to migrate to the level below since each level can only hold certain amount of data. Hence, this traditional architecture shown in Figure 2.2(a) is a single large key-value store system which involves a storage server and some hard disks.

However, with Kinetic Drives, it becomes different. As shown in Figure 2.2(b), in this new Kinetic architecture, each Kinetic Drive can be viewed as a small “key-value store” that can perform the key-value operations by itself. After receiving a user’s request, the metadata server can simply return the IP address of the destination Kinetic Drive to the user. Then the user issues a request to the Kinetic Drive via its IP address. The Kinetic Drive then performs the key-value operation without the server’s help. The metadata server is not directly connected to the Kinetic Drives to control them. Hence, the new Kinetic architecture in Figure 2.2(b) is multiple small key-value stores working together with a metadata server.

To show the advantage of Kinetic Drives, we conducted experiments to compare the traditional architecture of LevelDB on storage server with this new Kinetic architecture [2].

For Kinetic Drive testing, we used one Kinetic Drive and connected it to a machine which sent key-value pairs to the drive. The machine is a Dell R420 running Ubuntu Server 14.04 LTS 64-bit (kernel 3.13) on two quad-core Intel Xeon E5-2407 CPUs @ 2.20 GHz and 12GB RAM. The Kinetic Drive is model ST4000NK001 with 4 TB capacity, 5900 RPM, 1Gb/s Ethernet connection, and has a 64 MB disk cache plus a 512 MB onboard RAM to run the modified version of Linux and LevelDB [2]. We also installed LevelDB on a machine with the same specification as mentioned above and attached a traditional hard disk. The disk is SATA drive, Seagate model ST4000VN000. It has specifications similar to the Kinetic Drive, i.e., 4 TB, 5900 RPM, and 64 MB disk cache [2].

Results in Figure 2.3 are summarized from [2]. It shows the sequential write throughput. Each *key* is 16B and we changed the *value* size from 128B to 1MB. We inserted one million key-value pairs. We can see that when the *value* size is small, the traditional LevelDB server works better. This is because the storage server has a more powerful CPU than that in Kinetic Drive. It can process the data quicker.

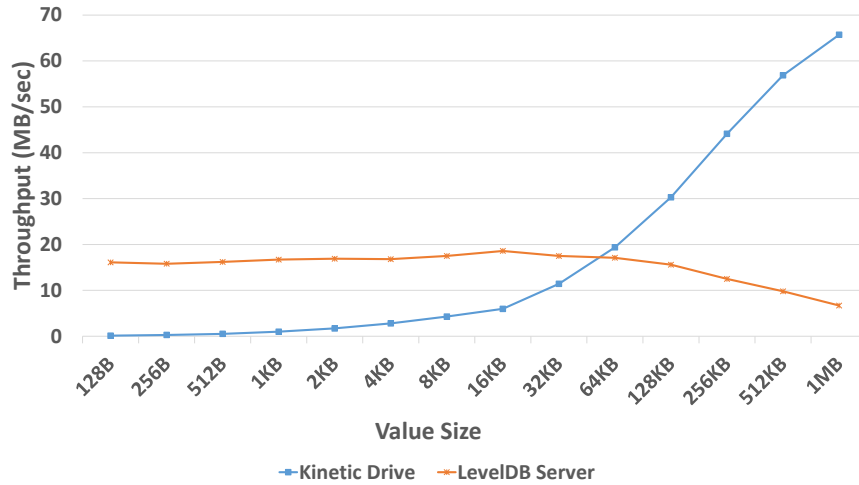


Figure 2.3: Comparison of Write throughput (Kinetic Drive vs. LevelDB server) [2]

As the *value* size grows, the data amount also increases. In that case, Kinetic Drive performs better. This is because the Kinetic Drive itself can run LevelDB. In LevelDB, data in each level need to be frequently updated with sorting and merging, and migrated to next level. These operations can take place in Kinetic Drive. Once the data are stored in the Kinetic Drives, they do not need to be fetched by the server for LevelDB operations or stored back to the drives. This greatly reduces a huge amount of I/O. For the traditional architecture, the LevelDB server has to frequently fetch the data from the disks, performs some LevelDB operations and sends them back to the disks which leads to a lot of I/O traffic. Hence, the Kinetic Drive performs better than the traditional approach when the data amount grows.

The results shown in Figure 2.3 [2] only involve one drive. In a large-scale deployment, 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.

In the traditional architecture, the storage server can be the system bottleneck. The limitation on the storage server can impact the scalability because every computing operation is run by the storage server.

The data migration among Kinetic Drives is easy and does not need the storage server's help. The Kinetic Drive can initiate a P2P operation to transfer data to another drive. Data can be directly migrated to another drive without storage server as the intermediary. In the large-scale environment, the data transfer among drives can be even in parallel. However, in the traditional architecture, parallel data migration becomes difficult because of the limitation of the storage server.

Another advantage of Kinetic Drives 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 Kinetic Drive that may hold the key-value pair to the user. Then the user can directly retrieve and store data from/to the Kinetic Drive via its IP address. Hence, the *value* 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 access with Kinetic Drive can greatly decrease the traffic amount going through the metadata server, which can further make the system more scalable. In the traditional solution, every user's requests have to go through the storage server which can be the bottleneck.

2.1.3 NUMA Architecture

NUMA (Non-Uniform Memory Access) [34] architecture plays a significant role in improving performance of computing system. It is based on the multi-processor system with shared memory. Although the specifications of different NUMA machines vary, the common features of NUMA architecture can be described as follows. NUMA machine is typically for multi-processor, multi-core computing environment. It consists of multiple blades in one single machine. Each blade has some multi-core processor sockets. Each socket (processor) is provided with a local memory. All the memories are shared and available by the sockets in the same and different blades. NUMALink(s) is used to interconnect the sockets with memories in certain topologies.

Each socket (processor) can access the data in its local memory. Data in remote memories can be transferred to local memory via NUMALink(s). Remote memories

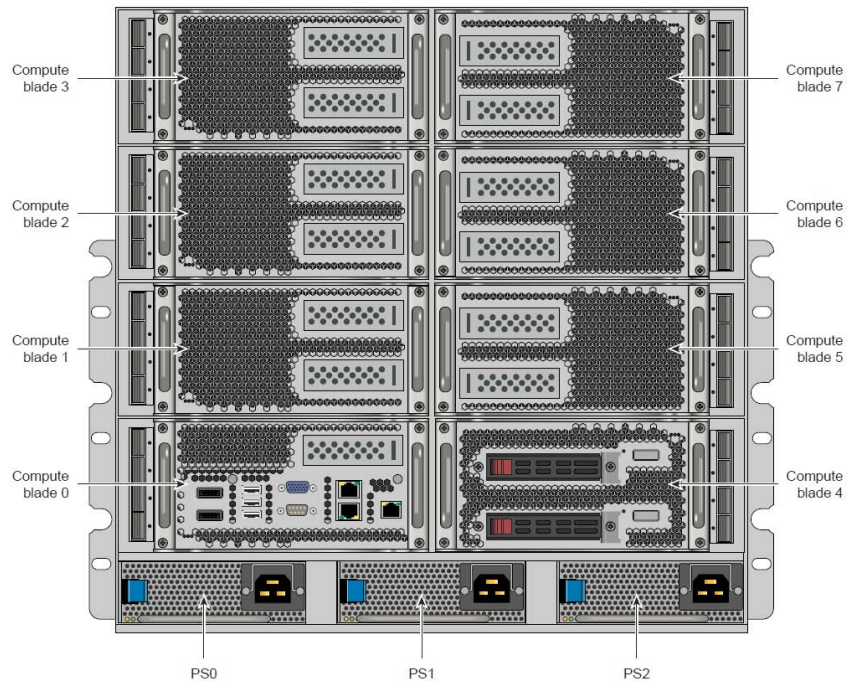


Figure 2.4: SGI UV 2000 IRU System Components Example [3]

mean the memories which belong to different sockets potentially in different blades. The data transfer bandwidth vary depending on the data location. Data in the memory from the same blade but different socket can be transferred to local memory quicker than that from different blades.

2.1.4 SGI UV 2000

The SGI UV 2000 [33] is the sixth generation of shared memory architecture from SGI. It is a scalable shared memory system with the interconnection of SGI NUMalink(s). It uses a compact blade design to support multi-processor, multi-core architecture running a single copy of standard Linux system. It provides a scale-up machine architecture for high performance computing and data intensive tasks.

The SGI UV 2000 machine shown in Figure 2.4 [3] as an example in this thesis has 8 blades. Each blade has two processor sockets as shown in Figure 2.5 in a simplified view of Individual Rack Unit (IRU). Each socket has a 8-core processor. Therefore, the

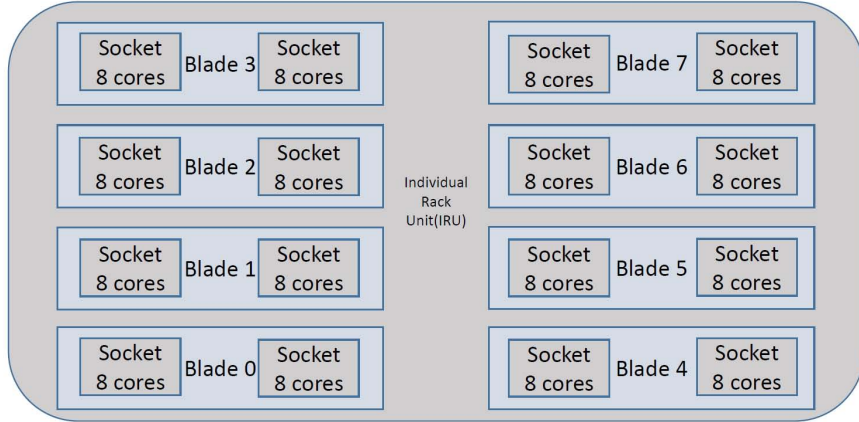


Figure 2.5: Simplified view of SGI UV 2000 IRU

entire SGI UV 2000 machine supports 16 processor sockets. Each socket has a local memory and these local memories of all the sockets are shared and accessed via SGI NUMALink(s). The entire machine can scale up to 64TB memory in total. In addition, different blades can also be connected by Ethernet. In SGI UV 2000, 1Gb/s and 10Gb/s ethernet connections are supported.

Figure 2.6 shows the interconnection topology of these 8 blades. In Figure 2.6, the circles represent the blades from 0 to 8. The blue straight line represent the direct one-hop NUMALink connections. We can see that some blades are connected with each other via one-hop NUMALink, while some other blades have two-hop NUMALink(s) to reach each other. Hence, there is a maximum of two NUMALink hops among all the blades. Data can be transferred from other sockets' memories remotely to local memory via NUMALink(s).

2.2 Related Work

2.2.1 NoSQL databases, Key-value Store and Kinetic Drives

Recently, NoSQL databases such as [7][8][22][23][24][25][26][35][36] have been developed to provide better performance and flexibility for many Big Data applications than traditional relational databases, especially for unstructured data. Generally, there are many types of NoSQL databases, such as document stores [23][24][37][38], column stores

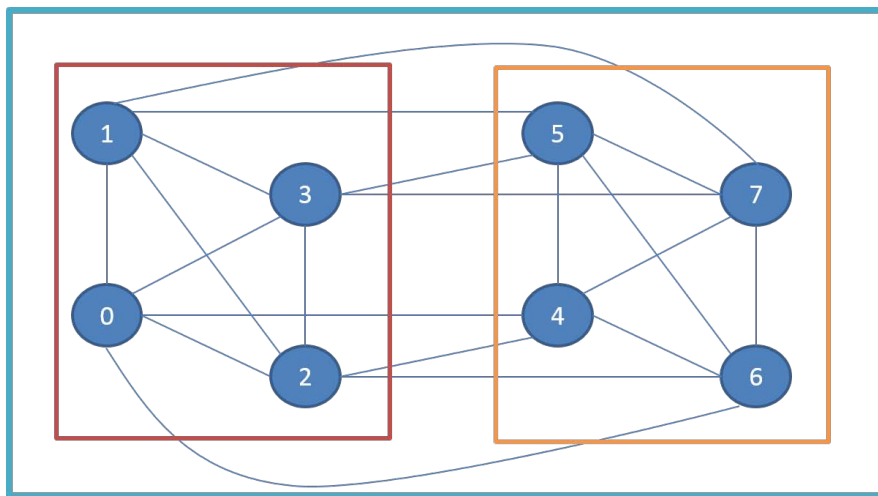


Figure 2.6: Interconnection topology of blades

[25][26][39], graph databases [40][41][42] and key-value stores [10][11][12][43][44][45].

Among these NoSQL databases, Google BigTable [25] and HBase [26] are very popular and widely used. They organize data in columns and provide flexible data storage. They are designed to scale across a large number of machines. Although they scale well for huge amount of data, they assume and use traditional storage servers to manage those data and disk drives. However, in our work of this thesis, we propose our data management of key-value store system based on Kinetic Drives, and design a Kinetic Drives based searchable key-value store system. As we mentioned in the background, Kinetic Drives can run key-value operations by themselves so that they have in-storage processing capability. This is very different from many existing work.

As a popular type of NoSQL database, key-value store system has been developed in recent years, such as Redis [10], Amazon Dynamo [11] and LinkedIn Voldemort [12]. They provided a large-scale key-value storage for various applications. Different from our design, these systems operated on a set of traditional disk drives which required separate storage server(s) or layer(s) to manage the data. In addition, they ignored the *key* distribution in the key-value pairs and assumed the *keys* were usually generated by a hash function. However, in our first part of the thesis, in order to efficiently support *key* range search and keep the semantic meaning of the *key*, we generally assume *keys* do not simply follow a uniform distribution. Also, these key-value store systems usually did

not support metadata information search from users very well. They typically assumed users had some knowledge of the *keys*. However, in our work for the Kinetic Drives based searchable key-value store system, we mainly focus on how to deploy a key-value store system for Kinetic Drives. Especially, we investigate a *key* generation approach to support the attributes search request.

HyperDex [35][36][46] provides searchable secondary attributes for users. It can map objects into multidimensional hyperspace to find out the correct servers. However, our work of searchable key-value store system differs from HyperDex [35][36][46] in several aspects. The data model in HyperDex [35][36][46] is an object with multiple attributes. Although a primary *key* can be used to retrieve the data, these attributes together is a data object. The work in [47] has a data model following the column store, document store and graph database. In our work, the actual data is generated by a Big Data application, such as an image, a video or a audio. The attributes of the actual data are metadata information. We believe these attributes and the actual data are grouped together as the *value* part in the key-value pair. Therefore, our work has a different data model. Another difference is that, similar to other NoSQL databases, HyperDex [35][36][46] assumes disk drives are managed by servers. In our work, we build our searchable key-value store system on Kinetic Drives, which can directly store and receive key-value pairs.

Many peer-to-peer (P2P) systems [48][49][50][51][52][53] provided key-value store for file sharing. Files were transferred among distributed users across the Internet without central server(s). In the P2P systems, data were typically stored in key-value pairs. They could be looked up and routed with other peers given the *keys*. Compared with our work, P2P systems focused on file sharing in a distributed environment. However, our design concentrates on the key-value store with Kinetic Drives to provide storage service for users.

Object-based Storage Devices (OSD) [13][14] have been introduced to provide a novel way to manage data as objects. Active Disks [15][16][17] are also innovative to process data on the devices. With more powerful CPU and larger memory, Active Disks can perform more functions beyond the 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 [1][18][19]. In [2], we evaluated the performance of

Kinetic Drives and compared them with LevelDB server for traditional hard disks.

There were some other studies about key-value store [36][54][55][56][57]. These studies investigated different aspects of key-value store. Different from the existing work, this thesis considers the advantages of Kinetic Drives and design the data management and searchable scheme.

To the best of our knowledge, our work in this thesis is the first one to address the data management and searchable feature of a large-scale key-value store system using Kinetic Drives.

2.2.2 Data Processing in MapReduce and NUMA Machine

Recent research work in [30][31][32] investigated the performance comparison of running MapReduce jobs on scale-up and scale-out architectures. In [30], a scale-up server running MapReduce jobs was compared with a 16-node scale-out cluster. The experiment results indicated that a scale-up server provided better performance per dollar, compared to a 16-node scale-out cluster. Research work in [31] had similar results. It showed that MapReduce jobs running on a single scale-up machine outperformed a 16-node cluster in some cases. In [32], the authors showed that fully distributed Hadoop running MapReduce jobs was not efficient compared with multi-core shared memory machine.

There were some research work [58][59][60][61][62] about task scheduling in MapReduce. Their main ideas were to consider the data locality of mappers/reducers and propose approaches to schedule the tasks on certain nodes, which could reduce the job complete time. In [29], the authors presented a weighted *shuffle* scheduling scheme in a cluster running MapReduce jobs. Other data shuffling schemes were proposed in [63][64]. However, all this related work were based on scale-out architecture. Machines were connected in a cluster with Ethernet or TCP/IP networks.

Some work about NUMA architecture have been done in [65][66][67][68][69][70] recently. In [65], NUMA-aware algorithms were proposed to *shuffle* data in NUMA machine. However, the data shuffling in [65] was not particularly based on MapReduce applications. It assumed that threads in every socket in the machine produced some data which would be consumed by every other threads. In other words, every thread needed to read data from every other threads. In the real MapReduce applications,

however, the number of mappers and reducers could be very different (i.e., The data producers are not exactly the data consumers). If a reducer is placed on the same node as a mapper, some amount of data do not need to be shuffled. Some reducers may need to *shuffle* more data than others. In addition, the algorithm proposed in [65] did not fully consider the topology information of computing blades in the NUMA machine. It simply let each thread rotate and take turns to read data from others. However, in our work, we consider different data transfer bandwidths among different sockets dependent on the memory locations to intelligently place reducers.

In [66], the authors proposed a multi-layered approach to optimize the performance of MapReduce on large-scale shared memory system with NUMA characteristics. An alternative implementation providing a modular and flexible pipeline was proposed in [67]. However, research work in [66][67] focused on the view of scalability of the entire system, especially on *map* and *reduce* task optimizations, but did not specifically address the data shuffling on NUMA machine.

A processing framework Spark was introduced in [71] for iterative analytics jobs to better utilize the distributed memory in the cluster. However, Spark does not focus on a particular hardware. It does not try to extract the hardware topology to make applications run faster, instead it operates and focuses on a distributed environment.

Chapter 3

Data Management of Key-Value Store System using Kinetic Drives

In this chapter, we discuss the data management of key-value store system using Kinetic Drives.

3.1 Introduction

In today's Big Data environment, huge amount of data are being generated. Users all over the world are storing and retrieving data all the time. It is important to provide a large-scale data repository to support data access for multiple users. Recently, NoSQL databases [7][8] were introduced to provide more flexible data access schemes than traditional relational databases. As one of the important NoSQL databases, key-value store (KVS) can be used for easy and efficient data storage and management. In KVS, a *key* is generated as an index to access data. A *value* is the actual data which can be any type (e.g., image, video, web page, text). Users can access the key-value pair given the *key*. Nowadays, key-value store systems have been widely used in many applications [10][11][12].

In data storage industry, a novel disk drive called Kinetic Drive [1][18][19] was developed by Seagate [20] recently. As an innovate example of Object-based Storage Device (OSD) [13][14] and Active Disks [15][16][17], a Kinetic Drive has certain computing capability. It can perform the key-value pair operations by itself with its own built-in CPU, RAM and implemented LevelDB system [21]. This invention of Kinetic Drives greatly changes the data access paradigm. As shown in Figure 2.1(a), the application has to go through a storage server to access data in the traditional disk drives. However, as shown in Figure 2.1(b), with Kinetic Drives, users can directly access key-value pairs via Ethernet connections, which is easier and more flexible than traditional block storage devices.

In many Big Data applications, huge amount of data are generated. Obviously, only a few Kinetic Drives are not enough to store a large amount of data. Instead, many Kinetic Drives are needed. With continuously generated key-value pairs and a large number of Kinetic Drives, data allocation becomes an issue, (i.e., Given a key-value pair, which drive should store that?). It is necessary to have a metadata server to manage those drives and to provide an indexing scheme for data allocation. Hence, it is important to map a large amount of key-value pairs to drives with an indexing table on the metadata server. It is not possible to map every key-value pair to a drive location since this will create an extremely large indexing table. Mapping *key* ranges to drives is feasible to reduce the size of the indexing table.

In the indexing table, each drive can cover either a large single *key* range or multiple separate small *key* ranges. In order to reduce the size of indexing table and management cost, it is better to reduce the number of *key* ranges covered by each drive. Also, *key* ranges can be disjoint (non-overlapped) or joint (overlapped) among many drives. In the joint (overlapped) design, any search given a *key* needs to be carried out in multiple Kinetic Drives which cover the *key* ranges. This involves too many drives and wastes valuable drive resources for a search. When one drive is full, data needs to be migrated to other drives. It is better to reduce the data migration amount. However, on the other hand, we also want to keep the number of non-empty drives low to save disk resources. Hence, these above design factors need to be carefully considered.

In this work, we propose data allocation schemes for a large-scale key-value store system 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 can get IP addresses of corresponding drives quickly.

This work has the following contributions.

- We show the tradeoff among several design factors, such as the amount of data migration, the average percentage of non-empty disks, the number of searched disks for a *key* and the number of *key* ranges per drive.
- We design indexing tables for the metadata server to quickly find out the correct disk drives to be searched given a *key* from the user.
- We propose approaches to allocate and migrate key-value pairs on Kinetic Drives for different *key* distributions considering the tradeoff among design factors.
- Our conducted performance evaluation shows the proposed indexing schemes and data allocation approaches can respectively handle various *key* distributions well.

The rest of this work is organized as follows. Motivation are presented in Section 3.2. We discuss our problem in Section 3.3 and propose efficient solutions in Section 3.4. Performance evaluation is in Section 3.5 and we conclude this work in Section 3.6.

3.2 Motivation

With huge amount of data stored in the data center, many Kinetic Drives are needed. In other words, one Kinetic Drive with a few TBs of storage capacity is obviously not sufficient. If a user wants to retrieve the key-value pair with a *key*, 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 returns all the Kinetic Drives’ IP to users 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.

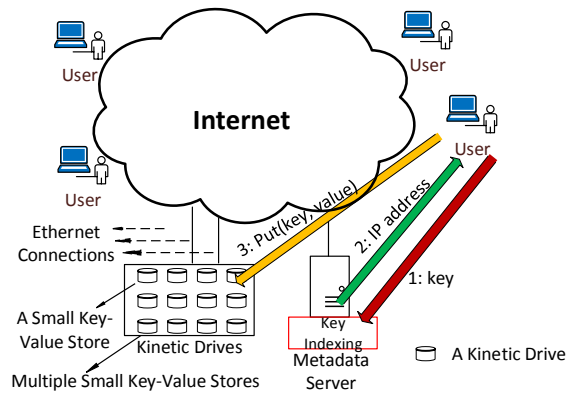
3.3 Our Problem

3.3.1 Our Scenario

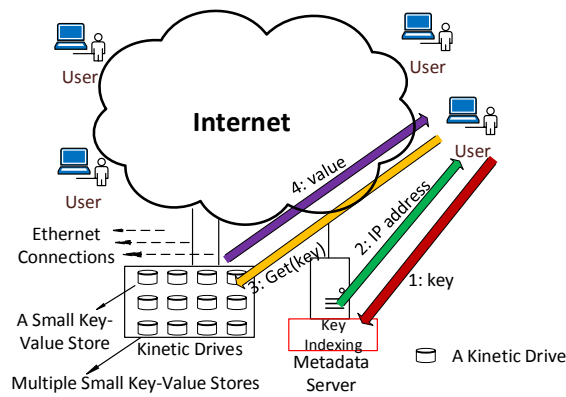
In this work, we consider the following scenario. In the data center environment supporting key-value store system, there are metadata server(s), data storage (Kinetic Drives) and outside users as shown in Figure 3.1. The metadata server(s) are connected to the Internet and have the *key* indexing table. 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 with Kinetic Drives, users can store (“Put(key, value)”), retrieve (“Get(key)”), delete (“Delete(key)”), and range query (“GetKeyRange(key1, key2)”) 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 3.1(a).



(a) User storing the data



(b) User retrieving the data

Figure 3.1: Key-value store system with Kinetic Drives

The user provides the *key* first (How to generate the *key* depends on the user, which is beyond this work'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 work, 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 3.1(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.

3.3.2 Research Issues and 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 pre-determined *key* 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 non-empty 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 non-empty disks in the system.

3.3.3 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 non-empty 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 non-empty disks.

3.4 Our Design

3.4.1 Assumptions

In this work, 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 simplify 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.

3.4.2 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. Each drive can cover a very large consecutive *key* range or multiple small separate *key* ranges. As discussed in the research issues in Section 3.3.2, it is better to reduce the number of *key* ranges per drive. If one drive covers too many separate *key* ranges, it brings too much entries in the indexing table, which further increases the storage overhead. Also, too many *key* ranges in one drive can lead to difficult management of mappings between *keys* and drives, especially for *key* range search.

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 stores key-value pairs with different non-overlapped *key* ranges.

Table 3.1: An example of key indexing table

| 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 |
| ... | ... |

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 chapter 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 3.1. 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...(key starting with 001). Hence, if a *key* falls into this range, the key-value pair will be stored and retrieved in that drive. In this example, we can see that in Table 3.1, one drive is responsible for only one consecutive *key* range.

3.4.3 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 3.1, 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 3.2 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 following part, we discuss the importance of the initial assignment and show our approaches. Also, we consider the percentage of non-empty disks as well.

Table 3.2: An example of the new key indexing table after further adjustment

| 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 |
| ... | ... |

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

all 0s to all 1s \rightarrow 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 *key*. The Kinetic Drives support an API “GetPrevious(key)” to get the previous key-value pair based on the current given *key*. In our *key* indexing table, it can simply record the smallest and largest *key* of a range for a drive. Hence, with the known ending (largest) *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... \rightarrow drive 0

1... \rightarrow 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... \rightarrow drive 0
 10... \rightarrow drive 1
 11... \rightarrow 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 extends 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 non-empty disks. The 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 non-empty 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 *key* ranges. 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.

3.4.5 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.

| | |
|------------|-----------|
| 0000000... | → drive 0 |
| 0000001... | → drive 1 |
| 0000010... | → drive 2 |
| 0000011... | → drive 3 |
| ... | → ... |

1111110... → drive 126
 1111111... → drive 127

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.

00000000... → drive 0
 0000001... → drive 1
 0000010... → drive 2
 0000011... → drive 3
 ... → ...
 1111110... and 00000001... → drive 126
 1111111... → drive 127

Since this approach assigns 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 non-empty disks in this “Prefix-All” approach can be very high. All

drives can be potentially accessed and occupied with data 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 *key* 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 non-empty 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 *key* starts with “000000” is stored in drive 0, as shown below.

```
000000... → drive 0
000001... → drive 1
000010... → drive 2
000011... → drive 3
```

... \rightarrow ...
 111110... \rightarrow drive 62
 111111... \rightarrow drive 63

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 3.4.4. 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... \rightarrow drive 0
 000001... \rightarrow drive 1
 000010... \rightarrow drive 2
 000011... \rightarrow drive 3
 ... \rightarrow ...
 111110... \rightarrow drive 62
 111111... \rightarrow drive 63
 0000001... \rightarrow 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 with 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... → drive 0
000001... → drive 1
00001... → drive 3
... → ...
111110... → drive 62
111111... → drive 63
0000001... → drive 64
... → ...

```

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 a 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 non-empty disks. At

the beginning, only half of the drives are potentially occupied with data and involved for data access. Then the system gradually expands its non-empty disk set on demand. Hence, this approach considers the tradeoff between percentage of non-empty disks and data migration amount. It is in the middle of “OneToAll” and “Prefix-All” approaches and provides a flexible data allocation scheme.

3.4.6 “Prefix-Half-2Drives” Approach

In the previous discussion, we show 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 of drives have overlapped *key* ranges, the number of involved disks searched 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*, at most two drives’s IP addresses are returned to users.

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.)

```

000000... → drive 0
000001... → drive 1
000010... → drive 2
000011... → drive 3
... → ...
111110... → drive 62
111111... → drive 63

```

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.

```

000000... → drive 0
0000001... and 000001... → drive 1
000010... → drive 2
000011... → drive 3
... → ...
111110... → drive 62
111111... → drive 63

```

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.

```

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

```

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.

3.4.7 A Special Case: How to handle roughly known key distributions

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 ($N \gg M$). We know that each range $R_i (i = 1, 2, \dots, M)$ 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 $U_i (i = 1, 2, \dots, M)$ 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 .

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.

3.5 Performance Evaluation

3.5.1 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.

3.5.2 Performance Comparison

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

Lightly Unbalanced Distribution

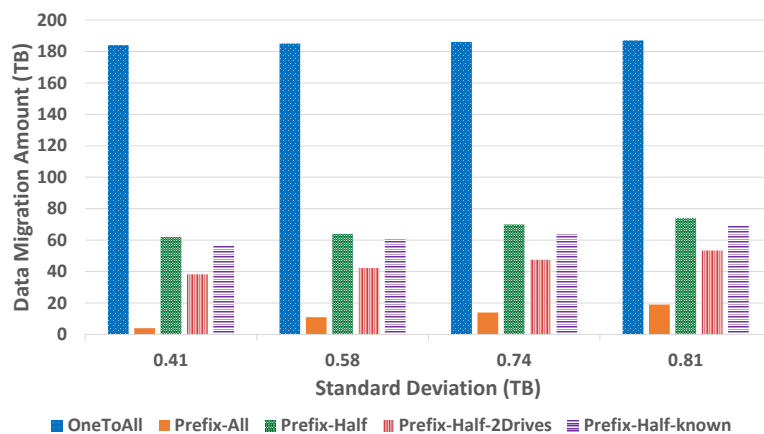
From our previous discussion, we can see that the “Prefix-All” approach can handle uniform *key* distribution well. Ideally, if the *keys* are uniformly distributed for all the *key* ranges, there are zero data migration. Key-value pairs can directly be stored in their predetermined drives. Also, in 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 1TB. We generate the traces with different standard deviations to compare the performance.

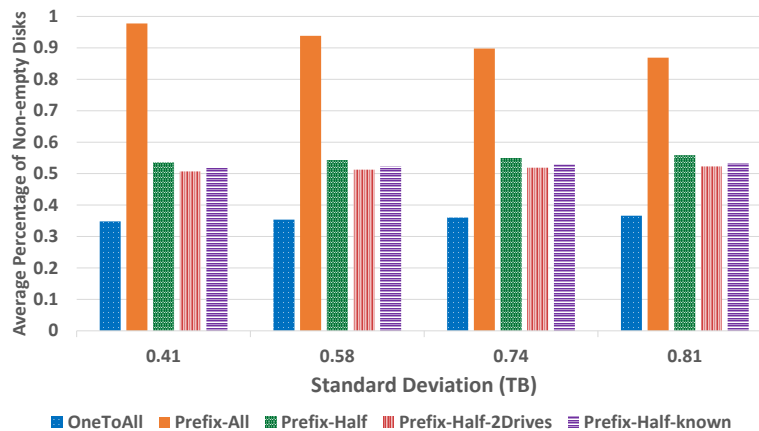
This type of traces can simulate and reflect some cases that there are different data amount in different *key* ranges. It is a typical case that some *key* ranges are popular while others are not. With the normal distribution, the lightly unbalanced data amount can be captured, but it does not fully reflect some highly skewed *key* distribution. In other words, we use the normal distribution to reflect the various popularity 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 3.2 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 3.2(a), we can see that the “Prefix-All” approach outperforms the others in terms of 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.



(a) Data migration amount



(b) Average percentage of non-empty disks

Figure 3.2: Comparison among different approaches - lightly unbalanced distribution

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 migrates data from one drive to 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.

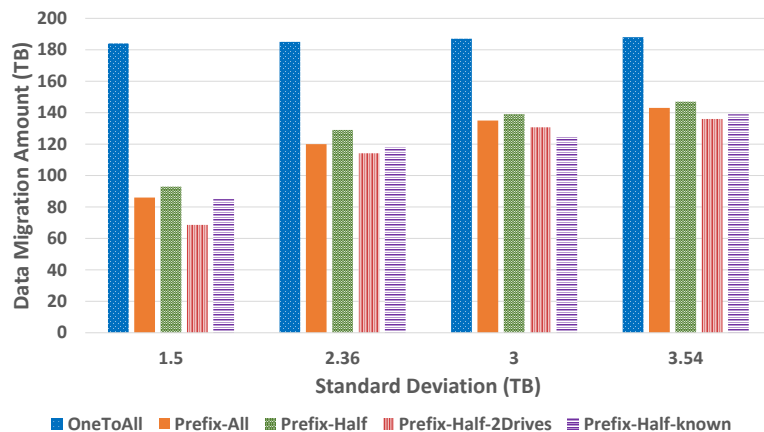
For the percentage of non-empty disks in Figure 3.2(b), we can find that the “OneToAll” approach has the smallest value. This is because data are first stored in one drive, then gradually migrated to other drives one by one. The “Prefix-All” approach yields the highest average percentage of non-empty 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 non-empty 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 non-empty 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.

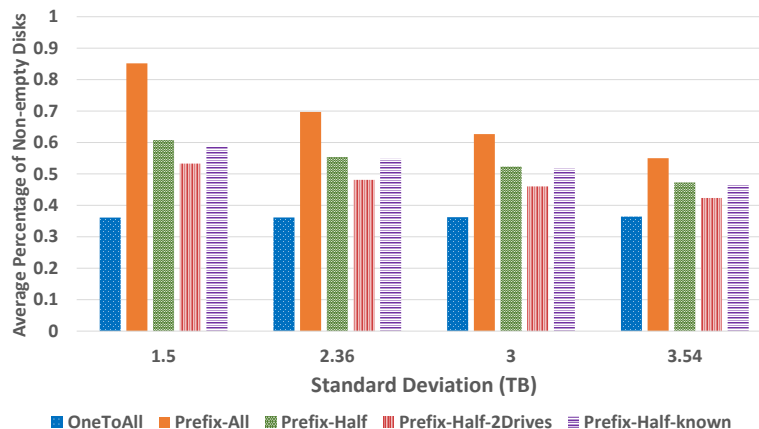
Highly Skewed Distribution

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.



(a) Data migration amount



(b) Average percentage of non-empty disks

Figure 3.3: Comparison among different approaches - highly skewed distribution

Figure 3.3 shows the results. Still, the “OneToAll” approach has the largest data migration amount. The “Prefix-Half” approach has more data migration amount than “Prefix-All” approach, but the difference is much smaller than that in lightly unbalanced *key* distribution shown in Figure 3.2(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 non-empty disks shown in Figure 3.3(b), we have the following observations. As data become more unbalanced, the average percentage of non-empty 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 non-empty 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 non-empty disks. Generally, the “Prefix-Half” is a good choice. With limited extra data migration, it can reduce the average percentage of non-empty disks and provides a more flexible and scalable storage scheme.

Random Distribution

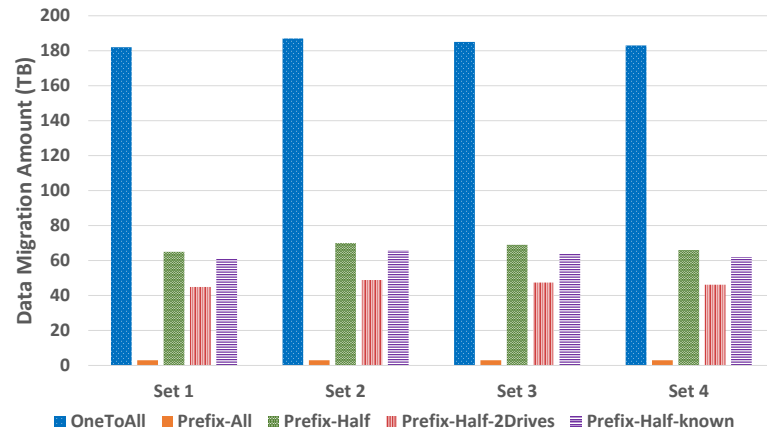
In addition to the above distribution, we also generate *keys* following random distribution. Among all the *key* ranges, the data amount are totally random. (The total key-value pairs are still kept the same as previous distributions.)

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.

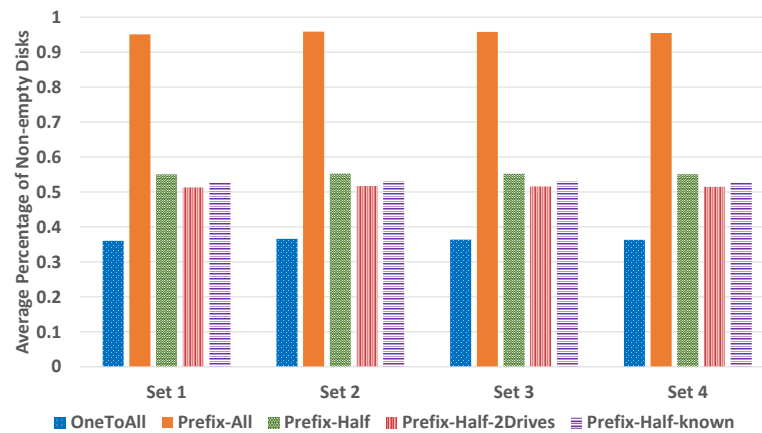
We conduct multiple experiments and select four sets of results as shown in Figure 3.4. In Figure 3.4(a), we can see that for data migration, the “OneToAll” approach is still the worst, while the “Prefix-All” approach yields the best result. The other three “Prefix-Half” related approaches are in the middle. For the average percentage of non-empty disks shown in Figure 3.4(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”.

To sum up, we can generally see the tradeoff between data migration amount and average percentage of non-empty 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.



(a) Data migration amount



(b) Average percentage of non-empty disks

Figure 3.4: Comparison among different approaches - random distribution

3.6 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 work, 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.

Chapter 4

Kinetic Drives based Searchable Key-Value Store System

In this chapter, we present Kinetic Drives based searchable key-value store system.

4.1 Introduction

In Big Data environment, data are being generated all the time, such as sales data, medical records, customer profiles. In many applications, the actual data is often generated with its metadata. For example, when an X-ray image is produced, some metadata (e.g., image size, generator ID, time, date) are also available. These metadata are important attributes for the actual X-ray image. Users later may retrieve the image by providing part or all the attributes to search. Hence, it is critical to store the actual data (e.g., image) with its metadata (attributes) together. Although some other NoSQL databases [22][23][24][25][26] (e.g., document store, column store) support data attributes access similar to relational databases, they often bring complicated operations and overhead. As a light-weight NoSQL database, key-value store, however, can bring more efficient performance. Traditionally, key-value stores usually do not support data attributes search very well. They simply store the actual data as a *value* and use a *key* to retrieve it, which can not fully capture the metadata information. In this work, we believe it is convenient and feasible to group the actual data and its associated metadata (attributes) together and store them as the *value* in a key-value pair. It is very important

to design a large-scale searchable key-value store system that can reflect metadata and support attributes search for users.

With huge amount of data, the locations of these data need to be decided. With Kinetic Drives, we consider which drive should store the key-value pair if a user provides a complete *key*. A separate metadata server is used to map the data to those Kinetic Drives. An indexing scheme is necessary for the data allocation. As discussed above, attributes search for users should also be supported in this searchable key-value store system using Kinetic Drives. Given an attributes search request from a user, (e.g., “Search(X-ray_image: image_size == 500KB, generator_ID == 001, time == 4:30PM, date == 01/01/2017)”), the system should quickly find out which drive(s) may have the actual data (i.e., image) that matches this attributes search. Then a complete *key* or a *key* range will be sent to the selected Kinetic Drives to retrieve the actual data. Hence, the metadata server should be designed for translating users’ attributes search requests to locations of the data.

When the metadata server decides a set of Kinetic Drives that may have the matched data for the attributes search request, it is efficient to reduce the number of Kinetic Drives to be selected. In other words, this set of Kinetic Drives should be as small as possible to avoid unnecessary searches for the drives. Also, the chosen Kinetic Drives do not have to search every data in them if an accurate *key* range can be received. In order to achieve these goals, a careful design of *key* generation scheme is necessary to reflect the attributes information of the actual data in the *value*.

In this work, we design a large-scale searchable key-value store system with Kinetic Drives. We propose an indexing scheme to map key-value pairs to Kinetic Drives. Attributes search requests from users are also considered in our design. We investigate a *key* generation approach to capture the metadata information and support users’ search requests.

This work has the following contributions.

- We propose an indexing scheme in the metadata server to map key-value pairs to the locations of Kinetic Drives.
- Our design considers users’ attributes search requests. Given attributes information, the metadata server can translate users’ requests and find out a limited set

of Kinetic Drives that may have the matched data.

- We also investigate a *key* generation approach that reflects metadata information of the actual data and supports users' search requests.
- Performance evaluation shows our design can reduce the number of Kinetic Drives to be searched for users' requests and decrease the data searches in the drives.

The rest of this work is organized as follows. Section 4.2 is the motivation. In Section 4.3, we present our problem. We propose our design in Section 4.4. Performance evaluation is conducted in Section 4.5. Finally, we conclude our work in Section 4.6.

4.2 Motivation

In many Big Data and IoT applications, huge amount of data are being produced. These data are stored in some places and retrieved by users later. Users often search data by providing some metadata information. For example, when an X-ray image is generated by a generator, the actual data here is the image itself. Some metadata information such as the image size, generator ID, time and date are available as well. Here we denote these metadata information as attributes of the actual data. Users can search the actual data (Here is the X-ray image.) by providing part or all the attributes. They may submit an attributes search request, such as “Search(X-ray_image: image_size==500KB, generator_ID==001, time==4:30PM, date==01/01/2017)” to retrieve the X-ray image which has these matched attributes. We can see that this type of attributes search request is popular in many applications. Hence, it is important to support this service for users.

Considering various data types in many applications, different data do not have uniform attributes. Many NoSQL databases [22][23][24][25][26] (e.g., document store, column store) are designed for data with different attributes, providing more flexible data storage and access than traditional relational databases. They also bring complicated operations and extra overhead. However, as a light-weight NoSQL database, key-value store can provide a simple but efficient data storage paradigm. Users can even access key-value pairs much easier with the invention of Kinetic Drives, which are specially designed for storing key-value pairs, as we mentioned earlier in the thesis.

In traditional key-value store systems, users can simply access a key-value pair by a *key*. However, in some scenarios, users may not know the exact *key*. Instead, they want to access data by submitting an attributes search request. Many current key-value store systems do not support this type of request well. They simply store the actual data as a *value*, using a *key* to access it. The *key* usually cannot fully reflect the metadata information (i.e., attributes) of the actual data (Hashing the *value* is a common way to generate the *key*). Therefore, it is difficult to provide the attributes search service for users.

In this work, we believe it is very important to design a large-scale searchable key-value store system using Kinetic Drives to capture the metadata information of the actual data. We consider the following scenario. There are data storage (Kinetic Drives), applications, metadata server(s), and outside users in a large-scale key-value store system. The applications generate the data and store them into the system with the help of the metadata server(s). The metadata server(s) have an indexing scheme that can receive users' requests, find out a set of Kinetic Drives (either for data storage or data retrieval), and return their IP addresses to the users. Then users can directly contact the selected drives via their IP addresses for data access. In our newly proposed key-value store system of this work, users are able to retrieve data by providing an attributes search request.

4.3 Our Problem

4.3.1 Research Issues and Challenges

In this work, there are several research issues and challenges we need to address as follows.

- In addition to data access given a *key* from a user, the proposed key-value store system should be able to reflect the metadata information of the actual data and support the attributes search requests. We need to consider this and design a novel approach to achieve that.
- Given a set of Kinetic Drives and huge amount of key-value pairs, an indexing scheme should be established to map data to the locations. Given a *key* or an

attributes search request from a user, the system needs to quickly find out the set of Kinetic Drives that may store the correct data, and return their IP addresses to the user.

- The number of selected Kinetic Drives for a *key* and an attributes search request should be kept as small as possible. In other words, the metadata server should not choose too many Kinetic Drives for a request, in order to reduce the disk resources for data search.
- If a Kinetic Drive is selected for a user’s request, this drive should try its best to avoid exhaustive search for all of its data. In other words, we should reduce the data search in the selected disks by providing a *key* range.

4.3.2 Our Goal

Based on our previous discussion, in this work, our goal is to design a large-scale key-value store system that can support users’ attributes search requests. Given a set of Kinetic Drives, an indexing scheme needs to be designed to quickly map data to their locations. The metadata server should return the IP addresses of a limited set of Kinetic Drives to users. In addition, data searches in the selected Kinetic Drives are reduced.

4.4 Our Design

In this section, we propose our solution, which includes the following parts.

4.4.1 Framework

In this part, we describe the general framework as shown in Figure 4.1 and show how our proposed key-value store system works.

1) Data Generation

There are many applications in our key-value store system. When an application generates an actual data (e.g., X-ray image), its associated metadata (e.g., image size, generator ID, time, date) are also available. Here, we denote the metadata as attributes

$(A_1, A_2, A_3, \dots, A_n)$. The actual data and the attributes are grouped together as a *value* in a key-value pair, i.e., $value = (A_1, A_2, A_3, \dots, A_n, actual_data)$. This application then generates a *key* using an approach which will be introduced later in this section. The *key* and *value* form a key-value pair.

2) Data Storage

The application can store the key-value pair it generates to the system. First, this application registers itself to the metadata server with an *application_name/ID*. Also, it provides the metadata server with a short description about how to interpret the following key-value pairs. The interpretation is related to the *key* generation approach, indexing and some explanations about the *value*. We will describe that later in details.

Then the application sends the *key* which includes the *application_name/ID* to the metadata server. After that, the metadata server analyzes the *key* based on the short description associated with the *application_name/ID*, decides the indexing, finds out the location (i.e., which drive) of this key-value pair, and returns the IP address of the destination Kinetic Drive to the application. The application finally directly contacts the drive via its IP address to store the key-value pair.

3) Data Retrieval

Later, a user may need to retrieve a key-value pair. Generally speaking, there are two types of data retrieval, either by providing a complete *key*, or submitting an attributes search request.

If the user provides the metadata server with a complete *key*, similar to the data storage above, the metadata server analyzes the *key*, decides the indexing and find out which Kinetic Drive may store it. Then an IP address of the Kinetic Drive is returned to the user. The user finally contacts the drive via its IP address with the *key* to retrieve the key-value pair.

If the user submits an attributes search request (e.g., “Search(X-ray_image: image_size==500KB, generator_ID==001, time==4:30PM, date==01/01/2017)”), then the metadata server also analyzes this request, decides the indexing and select one or more Kinetic Drive which may have the actual data whose attributes match the request. After that, the IP addresses of those selected Kinetic Drives are returned to the user

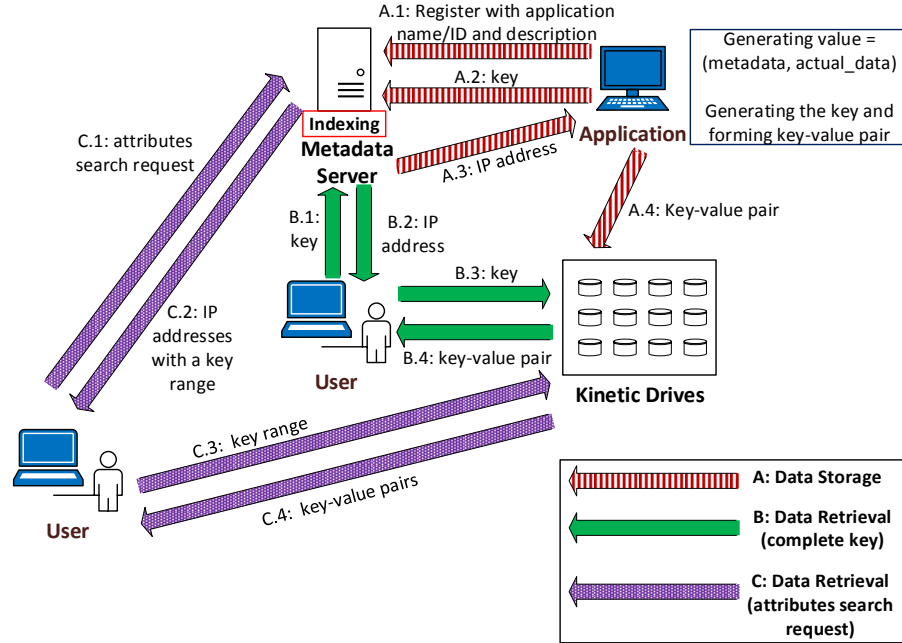


Figure 4.1: Framework of Kinetic Drives based Searchable Key-Value Store System

with a *key* range. The user finally contacts these Kinetic Drives via their IP addresses by providing a *key* range to search for the data.

Figure 4.1 shows the above framework. From our previous discussion, we can find that the metadata server generally maintains two tables. One is the table associated with the *application_name/ID*. It is used to record the information about how to analyze a *key* and an attributes search request. The other table is the indexing table, which is for deciding which drives may have the data the user is interested in.

In the rest of this work, we will discuss our solution in details as follows.

4.4.2 Key Generation

After an application produces an actual data with the metadata information, a *value* is ready. As we discussed before, here, $value = (A_1, A_2, A_3, \dots, A_n, actual_data)$, where $(A_1, A_2, A_3, \dots, A_n)$ are attributes. For example, an X-ray image with the size of 500KB,

produced by the X-ray generator 001 at 4:30PM on 01/01/2017 can be formatted as $value = (500KB, 001, 4:30PM, 01/01/2017, actual_X-ray_image)$.

Then this application generates the *key* of this *value* and forms a key-value pair. It is very important to reflect the metadata information of the actual data. A naive way is just simply letting the application send the entire data including the *value* to the metadata server, which analyzes it to understand the metadata information. However, this is unpractical because it would cause the metadata server to be a bottleneck because of huge amount of traffic. Hence, we need to consider a different approach.

In our solution, we take advantage of the *key* to capture the metadata information. Compared with the size of the *value*, the *key* size is smaller, which is easy and convenient to use. On the other hand, we believe the *key* size is large enough to include sufficient information to reflect the metadata information. (A Kinetic Drive can support a *key* up to 4KB, which is long enough to record many information.) In our design, we propose an approach to generate the *key*.

In Figure 4.1, we can see that the application generates the *key*. In our *key* generation approach, we partition the entire *key* into different parts, i.e., $key = (K_0, K_1, K_2, \dots, K_m)$. From our previous discussion about the framework, we know that the *key* should have enough information for the metadata server to analyze, both for data storage and data retrieval. Here, the most important information is the *application_name/ID*, which is used for identifying which application is sending the *key*. Therefore, the *application_name/ID* is included in the *key*. Each application is responsible for making its own *application_name/ID*, which can be a short string such as “X-ray_image”, “Sales_vehicles”, or “Information_students” to uniquely identify the application. To simplify the problem, we can just put *application_name/ID* in K_0 which is at the beginning of the *key*.

For the rest of the parts in the *key*, the application decides which information should be included. Typically, the *key* includes the attributes information of the actual data. For example, if the $value = (500KB, 001, 4:30PM, 01/01/2017, actual_X-ray_image)$, the *key* can be “X-ray_image, 500KB, 001, 4:30PM, 01/01/2017”, whose the first part is the *application_name/ID* followed by the attributes. It is important to note that the order of attributes which are included in the *key* really matters. (We will show the reason later.) In our design, the application decides this order (i.e., which attributes are in

K_1, K_2, \dots, K_m respectively.) Usually, the application puts the most popular attribute at the beginning of the *key* (i.e., K_1), stores the second most popular attribute in K_2 and so on. The least popular attribute is in K_m which is the last place. Here, the popularity means the frequencies of the attributes which appear in the users' search requests for this application. In the previous example of the X-ray image application, if the attribute "time" is searched by users most frequently, "time" should be put in K_1 . (Users do not have to provide all the attributes in their attributes search requests.) Hence, if the frequencies of attributes "time", "generator_ID", "date", "image_size" searched by users are in the descending order, then the *key* is "X-ray_image, 4:30PM, 001, 01/01/2017, 500KB". In this work, we assume each application knows this popularity of attributes searched by users.

Since the size of a *key* in Kinetic Drives is up to 4KB, usually, it can include every attribute of the actual data. However, if the application decides it prefers a shorter *key* (In our performance testings of Kinetic Drives [2], we showed that shorter *keys* could yield higher throughput.), it can include only part of the attributes (which are popular) in the *key*. Alternatively, partial contents (in terms of binary bits) of attributes can be selected in the *key* instead of complete bits. Also, a combination of these two methods can be applied as well. It is decided by the application.

In some cases, if the *key* including the attributes cannot uniquely identify the actual data, the application can generate a Global Unique Identifier (GUID) and insert it in the *key* to make it unique. For example, the application can simply hash an X-ray image, and put this hash value at the end of the *key* following the attributes.

In addition, the application needs to record the boundary information of different parts in the *key* and *value*. In other words, the metadata server should be able to understand how many bits each attribute takes, when it receives a *key*. For example, if the *application_name/ID* is "X-ray_image", the application makes the metadata server understand the first 11 symbols are *application_name/ID*. Hence, the application records the information of how to partition different parts in a short description, which is sent to the metadata server, in the registration phase.

In our previous discussion, we know that the application needs to register itself with its *application_name/ID* and a short description about how to interpret the following key-value pairs. In this short description, the application can include the following

information.

- Partition of different parts in the *key* and *value* including separation symbols.
- The frequencies (popularity) of each attribute searched by users.
- Which attributes are included in the *key*?
- The order of attributes in the *key*.

With the short description from each application, the metadata server has a clear knowledge about how to interpret the upcoming key-value pairs. A table of such information (*application_name/ID, description*) needs to be maintained in the metadata server.

4.4.3 Indexing

In the metadata server, there is an indexing table, which decides the mapping between key-value pairs and their locations. An application produces a key-value pair and sends the *key* to the metadata server for data storage. Then the metadata server analyzes the *key*, generates an indexing entry, compares it with the current indexing table, and decides which drive should store this key-value pair. The selected Kinetic Drive's IP address is returned to the application. Finally, the application directly contacts the destination Kinetic Drive via its IP address and stores this key-value pair.

In this part, we discuss the above process in details and show how the indexing works as follows.

1) Analyzing the Key

When the metadata server receives the *key*, it exams the first part of the *key* to find out the *application_name/ID* (i.e., which application sent the *key*). In the description of each application maintained in the table (*application_name/ID, description*), there is some information about how to generate the indexing entry, which will be discussed in the following part. Generally speaking, the application can include the information about how the metadata server generates the indexing entry for its *key*, put this information in the short description, and sends it to the metadata server in the registration phase. The metadata server then can decide the way to generate the indexing entry for applications.

2) Generating the Indexing Entry

Suppose there are totally N Kinetic Drives in the system. Given a *key*, the metadata server needs to decide which one of these N drives should store the key-value pair. To determine the location, an indexing table can be built. In the indexing table, each indexing entry points to a Kinetic Drive. In our proposed approach, we use a string of bits as the indexing entry. Since N drives are there, we need $\lceil \log_2 N \rceil$ bits (0 or 1) in the indexing entry to fully identify N drives. In that case, the indexing table looks like as follows.

| | | |
|------------|---|-----------|
| 0000...000 | → | drive 0 |
| 0000...001 | → | drive 1 |
| 0000...010 | → | drive 2 |
| 0000...011 | → | drive 3 |
| ... | → | ... |
| 1111...110 | → | drive N-2 |
| 1111...111 | → | drive N-1 |

Each row in this indexing table is an indexing entry. Given the *key*, once the metadata server generates the indexing entry, it can quickly find out the destination drive by looking up the indexing table. Hence, we show how the indexing entry is generated based on the *key* received from the application.

We partition the bits in the indexing entry into multiple parts P_1, P_2, \dots, P_K . Each part P_i is related to one attribute (e.g., time, date, or image_size) in the metadata information. Based on the (*application_name/ID, description*) table, the metadata server can decide which attributes in the metadata information of the application are needed to generate the indexing entry due to their popularity (search frequencies). If an attribute has a very high popularity (search frequency), we believe this is a very important attribute and should be assigned with more bits in the indexing entry. If another attribute is rarely searched by users, it is assigned with fewer or even no bits in the indexing entry. The intuition of this assignment is that users are more interested in those attributes with high search frequencies, so that important attributes with higher search frequencies should be given more bits to represent their values more accurately.

If there are many attributes of an application, it is possible that some of the attributes are not selected to fill in the indexing entry, due to the limited length of the indexing entry. (Even if we have 1024 drives, 10 bits are enough to identify them in the indexing entry). Of course, the total bits assigned to the selected attributes is equal to $\lceil \log_2 N \rceil$. Once the number of bits in each part of the indexing entry is decided (and updated in the $(application_name/ID, description)$ table by the metadata server), it decides what these bits are. According to our previous discussion about the *key* generation, we know that the *key* includes the values of the attributes of the actual data. Here, we simply copy the most significant (leftmost) bits of the selected attributes and put them together to form the indexing entry. Then the metadata server looks up the indexing table to find out what drive is pointed by this indexing entry, so that the destination drive is decided. Finally, the IP address of this drive is returned to the application, which directly stores the key-value pair into that drive.

To help understand our indexing scheme, we show an example. Suppose there are 1024 drives in total. Now the metadata server receives a *key* from the application “Sales_vehicles”. The *key* is “Sales_vehicles, 25000, Red, Sedan, 50000”, indicating this sale record is a red sedan with a 50000 mileage sold at the price of \$25000. The application wants to store this record. We assume from the description in the $(application_name/ID, description)$ table, the metadata server understands all of the 4 attributes are important and their bits should be in the indexing entry. Since there is a limit of 10 bits in the indexing entry, we partition these 10 bits in 4 parts. Based on the attributes’ popularity information in the description, the attributes “Price”, “Color”, “Type”, and “Mileage” are assigned with 4, 3, 2, 1 bits in the indexing entry respectively. (This indicates the attribute “Price” is most important and searched in highest frequency, while “Mileage” is least popular. The number of bits assigned to these four attributes can be decided proportionally according to their popularity.)

Then the metadata server just copies the most significant (leftmost) 4, 3, 2, and 1 bits of the values “25000”, “Red”, “Sedan” and “50000” respectively, and combines them together as the indexing entry. (Here, we can use the binary format to represent these attributes’ values. Hence, it is easy to copy the leftmost bits.) We assume the indexing entry formed is “1100 010 01 1”. Then the metadata server can immediately know this key-value pair should be store in the drive 787 $((1100010011)_2 = (787)_{10})$.

The IP address of the drive 787 is returned to the application. Finally, the application contacts the drive 787 directly to store this key-value pair.

3) Extensibility

When the drives keep receiving data for storage, at some moment, some drives are full and cannot store new data. In this part, we consider this case and discuss a solution.

When a drive is full, we can add a new empty drive and redirect future data into it. If the metadata server generates the indexing entry of a *key* and finds out the destination drive pointed by this indexing entry is full, the metadata server adds this new empty drive in the indexing table and return its IP address to the application.

For example, there are 1024 drives (Drive 0, 1, 2, ..., 1023) in total and now the drive 787 is full. The metadata server receives a new *key* and generates its indexing entry as “1100010011” (which points to the drive 787), then a new empty drive 1024 is added for those future data whose generated indexing entries are “1100010011”. Hence, all the future new data whose *keys* leading to the indexing entry “1100010011” will be stored in the Drive 1024, instead of the original one 787. The IP address of Drive 1024 will be returned to the application. In this way, the new indexing table is updated as follows.

```

0000000000 → drive 0
0000000001 → drive 1
0000000010 → drive 2
0000000011 → drive 3
... → ...
1100010011 → drive 787, 1024
... → ...
1111111110 → drive 1022
1111111111 → drive 1023

```

We can find that this redirection approach increases the number of drives pointed by the same indexing entry. As time goes on, more and more drives will be added. In that case, multiple drives may be involved for a user’s data retrieval request. From the practical point of view, it is better to decrease the number of drives being searched for one request from a user to save the disk resources.

Here, we propose a method to do further adjustment. We put a limit on the number of drives pointed by the same indexing entry in the indexing table and set a threshold (e.g., 3, 4, or 5). This means if the number of drives L pointed by the same indexing entry reaches the threshold, we do further adjustment in the following way. We migrate data among those L drives. Each drive stores different applications' data. We know that in each key-value pair, the *key* starts with the *application_name/ID*. Kinetic Drives store key-value pairs based on sorted *keys* using LevelDB [21], which means data in the same application are stored adjacently next to each other. As we discussed previously, data migration among Kinetic Drives is convenient. Hence, it is not difficult to migrate data among these L drives based on their *application_name/IDs*. In that case, these L drives can cover consecutive and disjoint data based on *application_name/IDs*, which can separate different applications' data into different drives.

As the previous example, as time goes on, more and more drives are added under the indexing entry “1100010011”, so that it looks as follows.

$$1100010011 \longrightarrow \text{drive } 787, 1024, 1025$$

Now, it reaches the threshold of 3 so that we do the data migration based on the *application_name/IDs* (i.e., *keys*). After data migration, each drive stores different applications' data separately in a disjoint way. Hence, the indexing entry may look like this.

$$\begin{aligned} 1100010011 &\longrightarrow \text{drive } 787, 1024, 1025 \\ \textit{application}1, 2, 3, 4 &\longrightarrow \text{drive } 787 \\ \textit{application}5, 6, 7 &\longrightarrow \text{drive } 1024 \\ \textit{application}8, 9, 10 &\longrightarrow \text{drive } 1025 \end{aligned}$$

If a user's request comes in, the metadata server can first generate its indexing entry “1100010011”. Then it can further find out the specific drive based on its *application_name/ID*.

4.4.4 Users Data Retrieval Request

In the previous part, we describe how the indexing works and data storage for a new key-value pair. In this part, we discuss our scheme about dealing with data retrieval

request from a user. Generally speaking, in our problem, there are two types of data retrieval requests. One is that a user knows a complete *key* and provides it for the metadata server. The other is that a user has no idea of the *key*, instead it submits an attributes search request, providing all or part of the attributes information.

1) A Complete Key

If the user knows the complete *key* and wants to retrieve the *value*, it can submit the *key* to the metadata server. Similar to the data storage process, the metadata server analyzes the first part of the *key* to find out the *application_name/ID* (e.g., which application the key-value pair belongs to). Then the metadata server looks up the table (*application_name/ID, description*) to understand how to generate the indexing entry based on the received *key*. After that, as we mentioned earlier, the metadata server generates the indexing entry and compares it with the indexing table to find out the destination drive which may have the key-value pair. The IP address of that drive will be returned to the user. Finally, the user can directly contact the drive to retrieve the key-value pair.

2) An Attributes Search Request

In addition to submitting a complete *key*, the user can send an attributes search request as we discussed in the previous part. This is a typical case when the user does not have an idea of the complete *key* information, but it wants to retrieve data based on some known attributes. In this part, we show how to handle this type of request.

When an attribute search request is initiated by a user, it is also with the *application_name/ID*, indicating which application's data it wants to retrieve. (All the available applications can be easily retrieved from the metadata server by the user.) A typical attributes search request has the format "Search(Application: $A_1 == V_1, A_2 == V_2, \dots, A_n == V_n$)" (e.g., it may look like "Search(X-ray_image: image_size==500KB, generator_ID==001, time==4:30PM, date==01/01/2017)"). Since the metadata server knows the *application_name/ID* from the user, it can quickly find out the useful information about how to generate the indexing entry (e.g., which attributes are selected in the *key*? How many most significant (leftmost) bits are needed in the selected attributes to generate the indexing entry?), by simply looking up the table

$(application_name/ID, description)$.

Here, we have two scenarios. One is that the user provides the complete attributes which are selected for generating the indexing entry of this application. In this case, it is similar to the data retrieval with a complete *key* from the user, as we discussed previously. The metadata server just looks up the table $(application_name/ID, description)$ and finds out how to generate the indexing entry given those complete attributes. Then it copies the most significant (leftmost) bits from these important complete attributes, combines them together, and forms an indexing entry. After that, the metadata server looks for the indexing table to find the drive which is pointed by this indexing entry, and returns the IP address to the user. Finally, the user directly contacts the drive to retrieve the data.

For example, an attributes search request “Search(Sales_vehicles: Price==25000, Color==Red, Type==Sedan, Mileage==50000)” is received by the metadata server from a user along with its interested $application_name/ID$ “Sales_vehicles”. By looking up the table $(application_name/ID, description)$, the metadata server understands that for the application “Sales_vehicles”, four attributes “Price”, “Color”, “Type” and “Mileage” are used to generate the indexing entry, which means the user’s request covers those attributes. Also, it knows that 4, 3, 2 and 1 significant (leftmost) bits of the values “25000”, “Red”, “Sedan” and “50000” are used to form the indexing entry. Then the metadata server just copies those bits, combines them together and generates the indexing entry (which we assume is “1100 010 01 1”). After that, the metadata server looks up the indexing entry 787 (Again, $((1100010011)_2 = (787)_{10})$.) to find out the IP address of the drive and returns it to the user.

The other scenario is that the user only provides part of the attributes which are used to generate the indexing entry for its interested application. In that case, the metadata server only copies the significant (leftmost) bits of those important attributes (which are used to generate the indexing entry based on the $(application_name/ID, description)$ table, combines them together, and forms an incomplete indexing entry. The missing part of the bits in the generated indexing entry can be filled with some special characters (e.g., ‘&’), indicating they are unknown. Then the indexing entry (including unknown bits) is compared with the indexing table to find matches. Since some positions of the indexing entry are unknown, multiple indexing entries in the indexing table can be

matched and found, indicating multiple drives could potentially store the data. The metadata server then returns those possible drives' IP addresses to the user.

Here is an example of this scenario. The user sends an attribute search request "Search(Sales_vehicles: Price==25000, Color==Red, Type==Sedan)". This request does not include all the attributes which are used to generate the indexing entry. Obviously, the attribute "Mileage" is missing. The metadata server then only copies the significant (leftmost) bits of the three attributes "Price", "Color" and "Type", and generates an incomplete indexing entry. We assume it is "1100 010 01 &". Here, '&' in the last position of this indexing entry means the unknown bit for the missing attribute "Mileage". The metadata server compares this generated indexing entry "1100 010 01 &" with all the entries in the indexing table. Eventually, it finds out that two indexing entries ("1100 010 01 0") and ("1100 010 01 1") are matched, indicating the drives pointed by the indexing entry 786 and 787 could potentially store the data with the attributes (Price==25000, Color==Red, Type==Sedan). Hence, those drives' IP addresses are returned to the user.

From the above example, we can see that it is important to assign more bits for the more popular (in terms of search frequency) attribute when we decide how to generate the indexing entry. In this example, the attribute "Mileage" is searched least frequently than other three attributes, so that only one bit is assigned. When the attribute "Mileage" is missing (which happens often), the metadata server is only uncertain for one bit of the generated indexing entry, which further leads to returning drives pointed by two possible indexing entries (i.e., 786 and 787). This does not cause too much search overhead. However, if the attribute "Mileage" is assigned with four bits in generating the indexing entry and is often missing in the attribute search requests, then frequently the metadata server has to generate the indexing entry missing four bits. In that case, often, the drives pointed by 16 indexing entries in the indexing table will be selected for returning their IP addresses, which obviously causes more search overhead.

4.4.5 Data Searches in the Drive

When the IP address(es) of the drive(s) are returned to the user for data retrieval, the user can directly contact the drive(s) to search the data. Given a *key*, Kinetic Drives

can search the data by themselves. If the user initiates the data retrieval by providing a complete *key*, the metadata server returns the IP address of the drive that may have the interested data. Then the user directly sends a “Get(*key*)” request to the drive. This drive then searches the data to see whether it has that or not. The entire process is not complicated because the user knows and submits a complete *key*. The drive understands what to search.

If the user sends an attributes search request to the metadata server, we need to consider more to reduce the data searches in the drives. When the user receives the IP address(es) of some drive(s) which could potentially have its interested data, it will need to send something to those drive(s) for them to search the data. Since the user does not know the complete *key* (Instead it only has some knowledge about some attributes, and that is why it initiates an attributes search request.), the user has no way to send the *key*. A naive approach can be that the user just asks the drive(s) to search every data. Obviously, we can see it has a huge data search cost. Drives have to go through every data from the beginning to the end. Hence, we should propose an approach that can reduce the data search amount for those drives.

An important feature of Kinetic Drives is that they support *key* range search. Given a *key* range (i.e., a starting *key* and an ending *key*), the Kinetic Drive can search the data with their *keys* between this range. Here, we take advantage of this property to reduce the data search amount in the drives. From our previous discussion, we can find that data of different applications can be in the same drive, as long as their indexing entries are identical. Since the metadata server understands which application the user is interested in, it is unnecessary to ask the drives to search those data which do not belong to this specific application. If we can use the *key* range to differentiate data of the user’s interested application from other applications’ data, data search amount can be greatly reduced. Even the drives can further decrease the data to be searched within that application if a proper *key* range is provided. Once the metadata server generates the *key* range, it sends that to the user along with the IP addresses of those selected drives. Then the user can directly contact those drives via their IP addresses and initiate a *key* range search to retrieve the data. Those drives can skip many data and only search data within the range.

We consider two scenarios of the attributes search request as we discussed previously.

One is that the user provides the complete attributes which are selected for generating the indexing entry of this application. We know that those attributes selected for generating the indexing entry are the most popular (with highest search frequencies) ones. Also, when a *key* is generated, the attributes are assembled in the order of their popularity, i.e., the most popular attribute is at the beginning of the *key* right after the *application_name/ID*. Hence, the metadata server can form a *key* range that includes the *application_name/ID* followed by the values of those attributes which are used to generate the indexing entry.

For example, the attributes search request from a user is “Search(Sales_vehicles: Price==25000, Color==Red, Type==Sedan, Mileage==50000)”, and those four attributes are complete ones used to generate the indexing entry in the order of their popularity. Then the *key* range generated by the metadata server is “Sales_vehicles, 25000, Red, Sedan, 50000, ****...****”, where “****...****” means unknown parts for other possible attributes. Hence, the starting *key* and ending *key* sent to the user are “Sales_vehicles, 25000, Red, Sedan, 50000, 0000...000” and “Sales_vehicles, 25000, Red, Sedan, 50000, 1111...111”. This *key* range includes every possible *keys* starting with “Sales_vehicles, 25000, Red, Sedan, 50000”. The user later can send this *key* range to the selected drives to search the data.

The other scenario is that the user only provides part of the attributes which are used to generate the indexing entry for its interested application. In this case, the metadata server tries its best to put as many most popular attributes as possible in the order of their popularity to follow the *application_name/ID* to form a *key* range until one of them is missing. If the attributes (A_1, A_2, \dots, A_m) are used to generate the indexing entry in the order of their popularity and A_i is missing, then the metadata server forms a *key* range “*application_name/ID*, $A_1, A_2, \dots, A_{i-1}, ****...****$ ”. Obviously, the worst case is that the *key* range is like “*application_name/ID*, ****...****”, which means the most popular attribute is missing.

Here is an example. A user’s attributes search request is “Search(Sales_vehicles: Price==25000, Color==Red, Mileage==50000)” for the application “Sales_vehicles”. The metadata server understands that “Price”, “Color”, “Type” and “Mileage” are the most popular attributes used to generate the indexing entry in the order of popularity. However, in this request, the attribute “Type” is missing. Hence, the *key* range is

“Sales_vehicles, 25000, Red, ****...****”.

4.4.6 Discussion

From our previous discussion, we can see that in our approach, data of different applications can be stored in the same Kinetic Drive, as long as their indexing entries are identical. Each Kinetic Drive has mixed data belonging to various applications. A simple alternative approach of data allocation is that the data are stored among the drives based on their *application_name/IDs*. Data of the same application are stored together in one or more drives. When the metadata server receives a key-value pair, it just decides the location based on its *application_name/ID*.

This alternative approach seems simple and efficient. However, we do not adopt that for the following reason. There are many applications in the entire environment. Some applications are very popular and have intensive store/retrieval requests from users, while others are less popular and have few requests. If data allocation is based on applications, it is very possible that some drives receive high workload for data access, while other drives are less active. This would cause the workload unbalance among the drives. Some drives even do not have enough bandwidth to sustain the requests.

Our proposed approach, however, does not allocate data based on their applications. Instead, we consider generating indexing entries depending on their attributes in the data. Hence, in our approach, data in the same application are distributed across multiple drives. Each drive has a chance to store different applications’ data. In other words, we introduce some “randomness” into the data allocation, which can balance the workload among the drives.

For the users’ attributes search requests, our proposed solution mainly focus on the exact match of the attributes. In our examples, we show that in the attributes search request, the user provides a specific value to search (e.g., $A_1 == 150$), instead of a range query (e.g., $100 < A_1 < 200$). However, our approach can also be applicable for the range query search. The metadata server picks up the most significant (leftmost) bits to generate the indexing entry so that the range query search is also supported. It can generate the indexing entries as long as the possible values of the attributes are within the range. For example, there is a range query for the attribute A_i , (i.e., $0 < A_i < 5$). If A_i takes 5 bits in the *value* of the key-value pair and 1 bit to generate the indexing

Table 4.1: Applications of Attributes Search Requests

| Application | Number of attributes of the actual data | Number of attributes selected for indexing | Percentage of data amount |
|-------------|-----------------------------------------|--------------------------------------------|---------------------------|
| 1 | 4 | 3 | 15% |
| 2 | 5 | 4 | 8% |
| 3 | 6 | 4 | 10% |

entry, then any value of A_i within this range ($0 \sim 5$) should have its very 1 bit for the indexing entry to be ‘0’ rather than ‘1’. This is because the bit ‘0’ covers the values of A_i from 0 to 15 ($(15)_{10} = (01111)_2$), and the bit ‘1’ covers the values of A_i from 16 to 31 ($(31)_{10} = (11111)_2$). Hence, the metadata server can also generate the correct indexing entry.

4.5 Performance Evaluation

4.5.1 Experiment Setup

In this section, we conduct the following simulation to show the performance. We assume there are 4096 Kinetic Drives and each drive has its storage capacity of 4TB. In our simulation, there are 10 different applications. We randomly generate key-value pairs for these 10 applications and store them into those Kinetic Drives. The total data amount is about half of the total storage capacity of all the drives.

Among these applications, we generate attributes search requests for 3 applications from users to show the performance. Table 4.1 shows the information of these 3 applications.

Here is the explanation of this table. For example, from Table 4.1, we can see that the actual data of application 1 has 4 attributes as their metadata. Because of the popularity searched by users, only 3 attributes (We just denote them as attributes A_1 , A_2 and A_3 .) are selected to generate the indexing entry. The data amount of application 1 is 15% of all the data for those total 10 applications. We can also find the information of application 2 and 3 as Table 4.1 shows.

Table 4.2: Number of bits assigned for attributes

| Application | Number of bits in attributes | | | |
|-------------|------------------------------|-------|-------|-------|
| | A_1 | A_2 | A_3 | A_4 |
| 1 | 7 | 3 | 2 | |
| 2 | 6 | 3 | 2 | 1 |
| 3 | 2 | 1 | 5 | 4 |

For each of these 3 applications, the information about the popularity of their attributes searched is as follows.

- Application 1: 30% of all the attributes search requests include all 3 attributes; 20% of all the attributes search requests miss the attribute A_3 ; 50% of all the attributes search requests miss the attributes A_2 and A_3 .
- Application 2: 10% of all the attributes search requests include all 4 attributes; 15% of all the attributes search requests miss the attribute A_4 ; 30% of all the attributes search requests miss the attributes A_3 and A_4 ; 45% of all the attributes search requests miss the attributes A_2 , A_3 and A_4 .
- Application 3: 20% of all the attributes search requests include all 4 attributes; 20% of all the attributes search requests miss the attribute A_2 ; 45% of all the attributes search requests miss the attributes A_1 and A_2 ; 15% of all the attributes search requests miss the attributes A_2 and A_4 .

Since there are 4096 Kinetic Drives in total, the indexing entry has 12 bits. Based on the above popularity of those attributes, the number of bits for each attribute in these 3 applications are in the Table 4.2.

4.5.2 Number of Drives Searched

When the metadata server receives an attribute search request, one or more Kinetic Drives are selected and their IP address(es) are returned to the user. As we discussed before, we should keep the number of selected drives as few as possible.

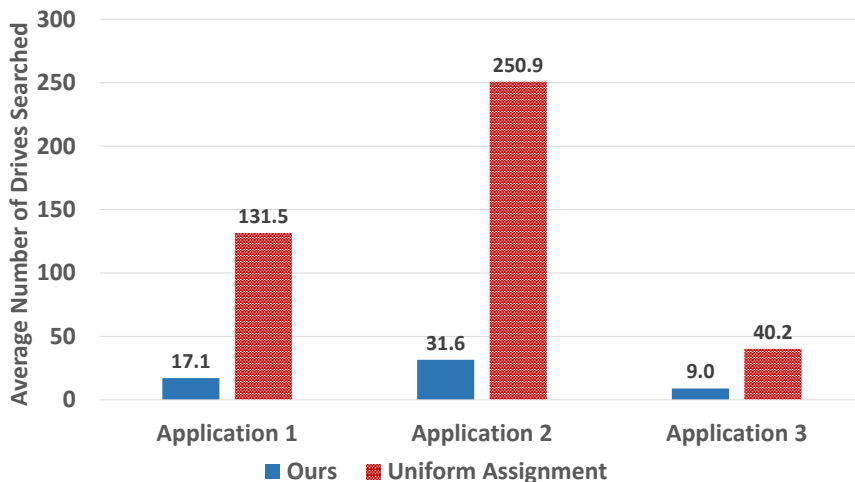


Figure 4.2: Comparison of average number of drives searched

In order to show the performance of our approach, we use the number of drives to be selected and searched as the metric. Here, we calculate the average number of drives searched of all the attributes search requests for each of these 3 applications. We compare our approach with the uniform assignment of bits for attributes in the indexing entry. In other words, our approach assigns different number of bits to attributes based on the search popularity, whereas the “uniform assignment” just assigns the same number of bits to each selected attribute in the indexing entry. Hence, in the “uniform assignment”, for application 1, each attribute in the indexing entry gets 4 bits. In application 2 and 3, each attribute gets 3 bits for the indexing entry in the “uniform assignment”.

Based on the search popularity mentioned in the experiment setup, we get the results as shown in Figure 4.2.

From Figure 4.2, we can see that the number of drives searched in our approach is much smaller than that of “uniform assignment”. For example, in application 1, the average number of drives searched for attribute search requests is 17.1 in our approach, whereas it is 131.5 if the “uniform assignment” for bits is used. We can also find out the difference between our approach and the “uniform assignment” for application 2 and 3. The reason of these differences is that our approach takes advantage of the various popularity among different attributes. If an attribute is often missed and rarely appears

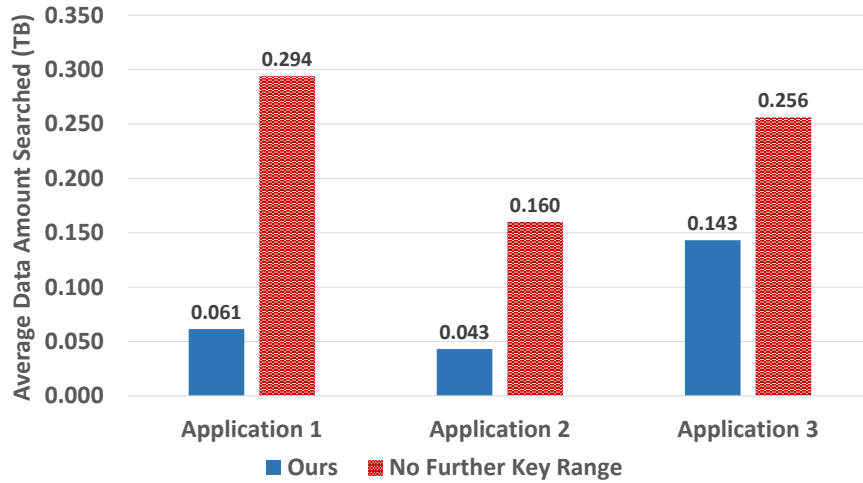


Figure 4.3: Comparison of average data amount searched

in the attributes search requests, it should be assigned with fewer bits, and vice versa. In this way, the average number of drives selected and searched can be reduced.

4.5.3 Data Searches in the Drives

In the previous section, we discussed the way to reduce the data searches in the drives. The user receives the IP address(es) of the chosen Kinetic Drives along with a *key* range. This *key* range can make Kinetic Drives limit the number of data searched in those chosen drives.

In this part, to have a fair comparison, we compare our approach of how to generate the *key* range with the method of “no further *key* range”. From our previous discussion, we know that each drive can store data from different applications. At the beginning of the *key*, the *application_name/ID* is included to differentiate various applications. With the method of “no further *key* range”, it only uses the *application_name/ID* to differentiate the specific application’s data from that of all the applications, without further providing a smaller *key* range related to the attributes search request. For example, if the user initiates an attribute search request for application 1, this “no further *key* range” method only directs the selected drives to search all the data of application 1 in them.

Here, we use the average data amount searched for selected drives per attributes

search request as the performance metric. Figure 4.3 shows the results. We can see that by using our approach, the data search amount in those selected drives are greatly reduced. For example, in application 1, our approach only leads to average data search amount of 0.061TB, whereas the “no further *key* range” method has the amount of 0.294TB data searches. The data amount searched in the drives of application 3 in our approach are greater than that of application 1 and 2. This is because the attributes search requests in application 3 have many misses with the first and second attributes. In that case, the provided *key* range becomes larger and covers more data, which yields more data searches.

From the results, we can conclude that by providing a simple *key* range with a starting *key* and a ending *key*, huge amount of data searches in the drives can be avoided so that many disk resources can be saved.

4.6 Conclusion

In Big Data applications, key-value store systems have been widely deployed to provide better performance. Kinetic Drives, which were recently invented by Seagate, provide Ethernet based data access. Given a *key*, a Kinetic Drive can process the key-value pair by itself. This greatly changes the storage stack, which enables a direct data access from a user to the Kinetic Drives.

In many applications, it is very important to support users’ attributes search requests. In this work, given a set of Kinetic Drives, we propose a large-scale searchable key-value store system. We design an indexing scheme in the metadata server for mapping data to Kinetic Drives. We also investigate a *key* generation method to reflect the actual data’s metadata information and satisfy the attributes search requests from users. From performance evaluation, we can see that our approach can reduce the number of Kinetic Drives to be searched for users’ requests. Also, data searches in the drives can be decreased as well.

Chapter 5

Data Processing in MapReduce with Scale-up NUMA Architecture

In this chapter, we present data processing in MapReduce with scale-up NUMA architecture.

5.1 Introduction

MapReduce [27][28] has become more and more prevalent in many applications (e.g., data analytics, high performance computing) in the last ten years. It provides a framework to allow processing huge amount of data in parallel. In a typical MapReduce application, *map*, *shuffle* and *reduce* are three major phases. Mapper nodes take input files, run *map* functions and produce intermediate key-value pair results in the *map* phase. In the *shuffle* phase, intermediate results generated in the *map* phase are transferred to reducers. Those reducers finally run the *reduce* tasks to produce the output results.

As an important operation in MapReduce, the *shuffle* phase usually takes a very long time to finish depending on many factors (e.g., network bandwidth, reducer placement). In [29], 188,000 MapReduce jobs from Facebook's traces were analyzed. The results

showed that, on average, the *shuffles* take 33% of the entire running time. Also, *shuffle* phases account for more than 50% and 70% of the entire running time in 26% and 16% of the jobs respectively. This shows that reducing shuffling time can greatly decrease the entire job running time in MapReduce applications.

Traditionally, MapReduce is assumed to be executed on scale-out inexpensive commodity machines. Therefore, most of the current optimizations [58][59][60][61][62][63][64] on *shuffle* phase in MapReduce are based on scale-out computing architecture. In Scale-out architecture, machines are typically connected by Ethernet or TCP/IP network which transfers the intermediate results produced by *map* phase to the reducers. Even though the network speed is becoming higher and higher, huge amount of data still take a lot of time to transfer which greatly impacts the optimization of *shuffle* phase.

In recent years, some research work about MapReduce on scale-up architecture have been investigated. In scale-up architecture, a single machine is equipped with more powerful CPUs and larger memories. In [30], a 16-node scale-out cluster was compared with a scale-up server running MapReduce jobs. The experiment results showed that compared to a 16-node scale-out cluster, a scale-up server provided better performance per dollar. Research work in [31][32] also had similar results. Nowadays, the CPUs are becoming more and more powerful and memory size in one single machine is becoming larger and larger. With larger memories in the scale-up architecture, intermediate results produced by *map* tasks can be kept in memories rather than in disks, which yields better performance. In [72], it was shown that median jobs input sizes in at least two analytics production clusters (at Microsoft and Yahoo) were under 14GB. In a Facebook cluster, 90% of jobs have input sizes under 100GB [72]. It is not difficult to purchase a server with memory of hundreds of GB [32][72]. Hence, it has become technically and financially feasible to run MapReduce jobs on a scale-up hardware with a single machine.

Nowadays, Non-Uniform Memory Access (NUMA) [34] system has emerged as a typical example of the scale-up architecture with large memory (e.g., SGI UV 2000 machine can scale up to 64TB memory [33]). It provides a better processing power in high performance computing (HPC) environment. In the NUMA architecture, computing blades are connected to each other by NUMALink(s). Each socket (processor) on a blade can access its own local memory with very low latency. Also, data from remote

memories of other sockets can be transferred to local memory via NUMALink(s). In addition to NUMALink(s), different blades can also be connected by Ethernet (e.g., In SGI UV 2000, 1Gb/s and 10Gb/s ethernet connections are supported.) in scale-out manner. The data transfer bandwidth of NUMALink(s) are dependent on the topology of how these sockets are connected. Although remote data transfer bandwidth between different blades is lower than that in the same blade, however, it is much higher than that of Ethernet or TCP/IP networks as shown in Table 5.1 in Section 5.2. Hence, with a scale-up NUMA machine, the data shuffling time can be reduced via NUMALink, compared with scale-out architecture via Ethernet connections. Also, considering the data locality and the difference of data transfer bandwidth from different memory locations in NUMA machine, the data shuffling time can be further reduced if we carefully decide where to place reducers on sockets to fully take advantage of the NUMALink(s). To the best of our knowledge, no previous work investigates such research issue on which our work focuses.

In [65], data shuffling on NUMA machine was investigated and a ring algorithm was proposed. As shown in Section 2.2.2, the algorithm of the data shuffling in [65] was not specifically focused in MapReduce environment. In this work, however, we particularly focus on the data shuffling in the MapReduce framework. We use the SGI UV 2000 machine [33] as an example to investigate data shuffling in scale-up NUMA architecture. Our goal is to reduce the data shuffling time in the MapReduce computing framework with the consideration of variations of different data transfer bandwidth based on different locations using NUMALink(s). This work has the following contributions.

1. We investigate the scale-up NUMA architecture for MapReduce. We find that it provides a much higher data transfer bandwidth via NUMALink compared with scale-out architecture using Ethernet or TCP/IP networks.
2. We propose a new topology-aware reducer placement algorithm for the scale-up NUMA architecture to reduce the shuffling time in the MapReduce framework.
3. We further extend our approach to a larger computing environment with multiple NUMA machines, and design a reducer placement scheme to expedite the inter-NUMA machine data shuffling.

4. Experimental results show that the data shuffling time is significantly reduced in the scale-up NUMA architecture with our reducer placement algorithm.

The rest of this work is organized as follows. Our motivation are presented in Section 5.2. We introduce the problem statement and propose our solution in Section 5.3. We further extend our approach to a larger computing environment with multiple NUMA machines in Section 5.4. Performance evaluation is presented in Section 5.5. Finally, we conclude our work in Section 5.6.

5.2 Motivation

Data shuffling is an important phase in the MapReduce framework. It transfers the intermediate key-value pair results from mappers to reducers. As shown in [29], the data shuffling phase accounts for a huge amount of portion in the entire job running time. Therefore, reducing the data shuffling time is a necessary and significant approach to accelerate the entire MapReduce job finishing time.

As discussed in the introduction, scale-up architecture can be used to run MapReduce jobs and yield better performance compared with scale-out architecture in many cases. With more powerful CPUs and larger memories in a scale-up machine, the computing productivity and efficiency can be greatly improved, thus leading to better performance results. Hence, it is promising to investigate the data shuffling optimization in scale-up architecture for MapReduce computing framework. In this work, we focus on the NUMA machine which is a typical example of scale-up architecture.

Essentially, the data shuffling destination depends on the location of *reduce* tasks. If a *reduce* task runs on the node that is closer to a mapper node where the intermediate results that *reduce* task requires are, the data shuffling time for this reducer is less. In an extreme case, if a *reduce* task can run exactly on the same node where the *map* task is, there is no need to *shuffle* the data for these amount of required intermediate results. Hence, the data locality is important for data shuffling. In the NUMA scale-up architecture such as SGI UV 2000 machine, the intermediate results can be stored in large memories. In the data shuffling phase, intermediate data results can be transferred via NUMALink instead of Ethernet or TCP/IP network. On the other hand, data in local memory in the same socket can be accessed by *reduce* task directly without data

Table 5.1: Data transfer bandwidth between memories in different locations

| Memory locations | Bandwidth | Latency |
|-----------------------------------------------------|-----------|---------|
| Different sockets on the same blade | 9.6GB/s | 411ns |
| Different sockets on different blades (1 hop away) | 7.7GB/s | 527ns |
| Different sockets on different blades (2 hops away) | 7.7GB/s | 650ns |

transfer. Data shuffling time therefore, can be reduced by carefully placing the *reduce* task on the processor (socket) which has the most required intermediate results as local data.

The difference of data transfer bandwidth depends on the hardware topology of how the blades are connected in SGI UV 2000 machine. To further validate the variations of different data transfer bandwidth of different locations, we use a standard benchmark - Intel Memory Latency Checker (MLC) [73] to conduct the experiments to compare the bandwidth and latencies of different memory locations in SGI UV 2000 single image system. The machine hardware specification is shown in Section 5.5.1. Table 5.1 shows the results. We can clearly see that data locality greatly affects the bandwidth of data transfer in different locations. Data transfer between different sockets but in the same blade has larger bandwidth. Remote data transfer between different blades, however, has a smaller bandwidth performance. Latencies also vary depending on memory locations.

Another important fact is that although the remote data transfer is slower, it still greatly outperforms the 1Gb/s and 10Gb/s Ethernet connections which SGI UV 2000 machine supports as an alternative way to connect different blades. Thus, it shows another strong promising reason that the scale-up NUMA architecture can improve the MapReduce job performance by reducing the data shuffling time with faster internal NUMAlink(s).

From the above discussion, we can see the difference of data transfer bandwidth from different memory locations. Hence, the data locality becomes an important factor to consider when transferring data from different locations in NUMA machine. In the data shuffling phase, the *reduce* tasks placement therefore, is significant in order to reduce the shuffling time. Intermediate data in the shuffling phase can be transferred faster if *reduce* tasks are placed with hardware topology such as shown in Figure 2.6

in mind. Hence, in order to speed up the data shuffling, we hope more intermediate data to be transferred are in the same blade. It is a better case if there is no need to transfer the data between memories. In other words, if a reducer is carefully placed in the socket whose local memory has the required data, there will be no data transfer which can save a huge amount of time. Strategically placing *reduce* tasks on different sockets in different blades to utilize the location of intermediate data generated by *map* tasks becomes crucial to accelerate the data shuffling in scale-up architecture. With an intelligent *reduce* tasks placement algorithm for the data shuffling considering different data transfer bandwidth from different memory locations, the overall data shuffling time can be reduced.

In this work, we take the advantage of data locality with different data transfer bandwidth. We consider the hardware topology of blade interconnections in NUMA machine and propose a topology-aware *reduce* tasks placement algorithm for data shuffling acceleration shown in the next section.

5.3 Problem Statement and Proposed Solution

5.3.1 Problem Statement

In this work, we consider the data shuffling acceleration in the MapReduce computing environment in scale-up NUMA machine. In our problem, all *map* tasks are assumed to be executed in parallel. In the shuffling phase, the intermediate key-value pair results are divided into different ranges based on the number of reducers. Each reducer is responsible for running *reduce* tasks in a specific *key* range. As discussed before, in scale-up NUMA architecture, the data transfer bandwidth varies depending on the locations of source and destination memories. Hence, how to select the locations of reducers on different sockets is essential for data shuffling acceleration. To reduce the data shuffling time, the reducers should be placed in such a way that the total time cost of shuffling key-value pairs is as low as possible.

Therefore, our problem is formulated as follows. In the MapReduce framework, given N mappers and M reducers ($M < N$ typically), each mapper have $m_j, j = 1, 2, \dots, N$ intermediate key-value pair results. Our goal is to design a *reduce* task placement algorithm to select the locations for these M reducers among N nodes running *map*

tasks, such that the total data shuffling time for these M reducers is minimized.

5.3.2 Proposed Solution

In our solution, we consider the data locality in the memory locations and take advantage of the blade interconnection topology shown in Figure 2.6 to place *reduce* tasks on sockets to accelerate the data shuffling process. Our placement algorithm assigns the proper locations of these M reducers on N nodes running *map* tasks. Our solution is as follows.

Since there are M reducers, *keys* in the intermediate results are divided into M disjoint ranges, $R_i, i = 1, 2, \dots, M$ for each *map* task. Some ranges have more *keys* while others have fewer *keys*. In our approach, we decide the locations of M reducers one by one. In other words, the shuffling destinations of *keys* in each R_i are decided sequentially. For each reducer $i, i = 1, 2, \dots, M$, we define the term *AFFINITY* to calculate the time cost of data shuffling for each range $R_i, i = 1, 2, \dots, M$. Suppose there are s_{ij} amount of *keys* in each range R_i on j th mapper, $j = 1, 2, \dots, N$. We use KEY_{ij} to denote these *keys*, i.e., $\|KEY_{ij}\| = s_{ij}$. When we select the location of the first reducer, there are N possible locations which are corresponding to N mappers (Without losing generality, the locations of reducers will be chosen from these nodes running *map* tasks. Some locations from N candidates may not be used to place the reducers due to restrictions, e.g., CPU limit. In that case, our approach just simply skips these locations and proceed with next candidate.)

We define the concept of number of hop counts n_{ijk} . From Table 5.1, we know that the data transfer bandwidth are different dependent on the memory locations. If source and destination memories are in the same socket, we define it as 0 hop count since it does not require data transfer. If they are on the different sockets on the same blade, we define it as 1 hop count. Otherwise, it has 2 hop counts. Hence n_{ijk} means the number of hop counts that those *keys* of KEY_{ij} need to go through to k th node. In this way, the factor of hop counts is considered.

In our approach, we consider these N possible locations one by one, by calculating the $AFFINITY_k, k = 1, 2, \dots, N$. Here, $AFFINITY_k = \sum_{j=1}^N s_{ij} \cdot n_{ijk}$. (At first, $i = 1$, since we are selecting the location of the first reducer). In other words, $AFFINITY_k$ is the sum of number of *keys* in range R_i , (i.e., s_{ij}) to be transferred times the number

Algorithm 1 Topology-aware reducer placement

Input: The intermediate results KEY_{ij} of N map tasks, the hop counts number n_{ijk}

Output: The locations of each reducer

```

for Each reducer  $i, i = 1, 2, \dots, M$  do
  for Each possible  $k$ th location,  $k = 1, 2, \dots, N$  do
     $AFFINITY_k = \sum_{j=1}^N s_{ij} \cdot n_{ijk}$ , where  $s_{ij} = \|KEY_{ij}\|$ 
  end for
   $h = \operatorname{argmin}\{AFFINITY_h\}$ , for all  $h = 1, 2, \dots, N$ 
  Put reducer  $i$  on the  $h$ th location of all  $N$  candidates.
end for

```

of hop counts n_{ijk} ($i = 1$ at first) to reach the reducer if it is selected to place on the k th node. Then we select the $AFFINITY_k$ of the smallest value and put the first reducer on that corresponding node. After that, we iteratively decide the next reducer until all of them are decided. The idea of this approach is that we hope the reducer can be placed in the node that cause least amount of data movement cost with the consideration of the hop counts. Algorithm 1 shows the above selection process.

As indicated in the introduction, many MapReduce jobs have input sizes less than 100GB. Hence, we assume memory in SGI UV 2000 NUMA machine has sufficient space to hold the data of jobs, considering the very large memory it can support (up to 64TB).

Although in SGI UV 2000 machine, the maximal hop counts between different blades is 2, our solution is not limited to 2 hop counts. For other possible future scale-up architecture with more hop counts from source to destination memory, our solution can still work efficiently.

5.4 Extension to Multiple NUMA Machines

In this section, we extend our approach to a larger computing environment with multiple NUMA machines, and design a reducer placement scheme for inter-NUMA machine data shuffling. We first introduce the problem, then describe our solution in details.

5.4.1 Problem

We consider the follow scenario. A large MapReduce job is submitted to the system with certain Service Level Agreement (SLA). In other words, this large MapReduce job

is expected to finish in certain amount of time. In HPC environment, it is a typical scenario. In this case, one single NUMA machine may not be sufficient to finish the job on time due to the limitation of CPU capability. For example, one SGI UV 2000 machine only supports up to 128 CPU cores. If a job has a very urgent deadline to finish, it may require more CPU resources in terms of cores to participate in the *map* and/or *reduce* phases. Hence, we need multiple NUMA machines which are connected to each other by Ethernet to compute. However, our approach to place reducers shown in the previous section is only limited to one single NUMA machine, and cannot be directly applied to multiple NUMA machines. Therefore, it is necessary to extend our approach to a larger computing environment with multiple NUMA machines and consider how to design a reducer placement scheme for that environment. We need to decide which NUMA machines are active for *reduce* tasks. Also, we should consider within an active NUMA machine, the locations of sockets and blades to place the *reduce* tasks as well. Our problem is described as follows.

There are P NUMA machines running Q *map* tasks in total. Each NUMA machine has a CPU resource of C and is connected to each other by Ethernet. Without losing generality, we assume each *reduce* task consumes U amount of CPU resources. Each *map* task produces certain amount of intermediate key-value pair results. There are D *reduce* tasks in total. Our goal is to select a subset of these P NUMA machines to run *reduce* tasks, and decide where to place those D *reduce* tasks on selected machines, such that the data shuffling time is reduced. We also decide within a NUMA machine running *reduce* tasks, the locations of sockets and blades to place reducers.

5.4.2 Our Solution

Before the *reduce* task placement, we first consider how many NUMA machines are sufficient. Without any doubt, we can use as many NUMA machines (up to P) to run *reduce* tasks as possible. However, in the perspective of high performance computing and other data intensive applications, each NUMA machine is a valuable resource for computing. If we can consolidate *reduce* tasks into as few NUMA machines as possible, we can save more entirely more available NUMA machines to run other tasks. Hence, our first step is to decide the number of NUMA machines for *reduce* tasks. A simple way is introduced here. Since each *reduce* task consumes U amount of CPU resources

Algorithm 2 Reducer placement on multiple NUMA machines

Input: P NUMA machines, the intermediate results of Q *map* tasks, D *reduce* tasks

Output: The locations of each reducer

```

for Each reducer  $i, i = 1, 2, \dots, D$  do
  if Current chosen NUMA machine has sufficient CPU resource then
    Use Algorithm 1 to place current reducer on the proper socket;
  else
    Choose a new NUMA machine with the most intermediate data in current re-
    ducer's key range;
    Use Algorithm 1 to place current reducer on the proper socket on this new chosen
    NUMA machine;
  end if
end for

```

and there are D *reduce* tasks, the total required CPU resources are DU . We just use $\lceil DU/C \rceil$ to get the number of NUMA machines to run *reduce* tasks. Again, similar to the previous approach, we assume the SGI UV 2000 machine has sufficient memory space to hold the data of jobs.

Then, similar to Algorithm 1, we decide the *reduce* task placement one by one. For the first *reduce* task, we need to decide which NUMA machine is used to place that. The idea is similar with our previous approach. Since each NUMA machine running *map* tasks produces intermediate key-value pair results, we select the NUMA machine with the most intermediate data in the first *key* range to place the first *reduce* task. In this way, we can minimize the number of data transferred from other NUMA machines belonging to that range.

After we determine the NUMA machine for current *reduce* task, we decide which socket that *reduce* task should reside. We use our previous approach as shown in Algorithm 1 to decide that. The purpose of this approach is to minimize the data transfer within the NUMA machine. At this stage, we have decided the first *reduce* task placement.

Next, we move on to place the subsequent *reduce* tasks. For each *reduce* task, we first examine whether the current chosen NUMA machine has sufficient CPU resources. If there are available CPU resources for current *reduce* task, we just use the same method shown in Algorithm 1 to select the appropriate socket to place it. Otherwise, we have to choose another new NUMA machine to place that *reduce* task. We will select the

Table 5.2: Simulation setup I (one single machine)

| Set | Number of mappers (N) | Locations of mappers (sockets #) | Number of reducers (M) |
|-----|-----------------------|----------------------------------|------------------------|
| 1 | 4 | 0, 1, 2, 3 | 2 |
| 2 | 8 | 0, 1, 2, ... ,6, 7 | 4 |
| 3 | 16 | 0, 1, 2, ... ,15, 16 | 8 |

NUMA machine with the most intermediate data in current *reduce* task’s *key* range to minimize the data transfer, and place the *reduce* task on the proper socket. At this stage, the current chosen NUMA machine is this new machine, and will be used for next *reduce* tasks placement.

We continue using the above approach to place the remaining *reduce* tasks until all of them are placed. Algorithm 2 shows the entire process.

5.5 Performance Evaluation

5.5.1 Experiment Setup

We use the SGI UV 2000 machine with the following configurations to conduct the experiments shown in Table 5.1. There are 16 sockets on 8 blades in the machine which can supports up to 128 CPU cores. Each socket has 256GB memory with 8 DDR3 DIMMs per node @ 1333MHz. The CPUs are Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz. The machine is running SUSE Linux Enterprise Server 11 SP3.

5.5.2 Performance Comparison within one NUMA machine

In order to show the performance of our topology-aware reducer placement algorithm within one NUMA machine, we conduct the following simulations using the data transfer bandwidth shown in Table 5.1. we use the total data shuffling time as the metric to compare our algorithm with random reducer placement and scale-out scenario.

There are three sets of simulations. In each set of simulation, each mapper runs on one socket and has intermediate key-value pair results of 4GB. The simulation setup is shown in Table 5.2. Within the 4GB data in each mapper’s intermediate results, the

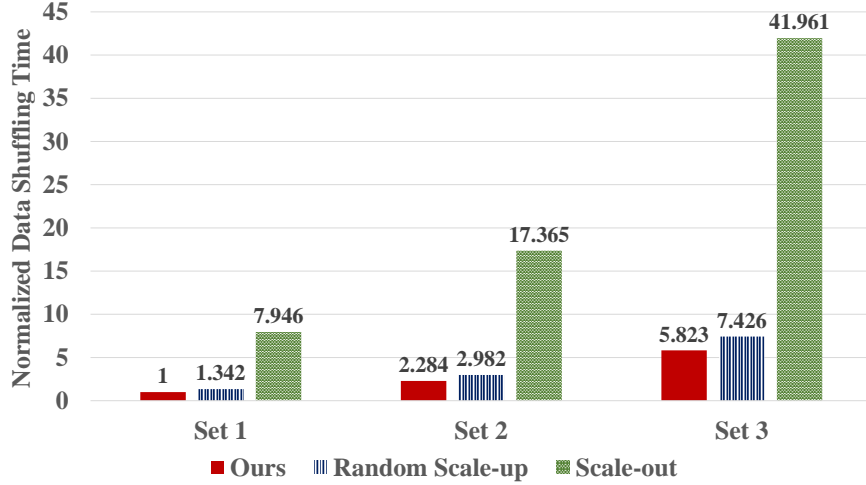


Figure 5.1: Normalized data shuffling time among different schemes

number of *key* ranges varies dependent on the number of reducers. The number of *keys* in each range are randomized. For example, in the first set of simulations, there are 4 mappers running on socket 0, 1, 2 and 3. We need to decide the locations of 2 reducers on these 4 sockets.

To show the performance, we compare our approach with random reducer placement. In other words, we randomly pick up M reducers out of N locations where mappers reside and calculate the total data shuffling time cost. For the scale-out scenario, these mappers are distributed in scale-out architecture with the Ethernet connections of 10Gb/s which SGI UV 2000 machine supports, instead of NUMalink. We also randomly choose the locations of reducers in the scale-out scenario, which is similar to default approach in MapReduce. (In default, MapReduce uses Hash function to decide the shuffling destinations of intermediate results which brings some similar randomness as well.)

Figure 5.1 shows the comparisons of normalized data shuffling time in three sets of simulations. We can see that our topology-aware reducer placement algorithm outperforms others with minimal *shuffle* time. Since Ethernet connections are used in the scale-out scenario with much smaller data transfer bandwidth, it has the longest data shuffling time. With our topology-aware reducer placement algorithm, we consider the difference of data transfer bandwidth based on memory locations. Hence, our approach

Table 5.3: Simulation setup II (multiple machines)

| Set | Number of machines running mappers | Number of mappers | Number of machines running reducers | Number of reducers |
|-----|------------------------------------|-------------------|-------------------------------------|--------------------|
| 1 | 4 | 64 | 2 | 32 |
| 2 | 8 | 128 | 4 | 64 |
| 3 | 16 | 256 | 8 | 128 |

to place reducers can yield a better performance in terms of data shuffling time compared with random placement. We can find that our approach can reduce 20% - 30% of the data shuffling time. In addition, from the results shown in Figure 5.1, we can find that scale-up NUMA architecture using NUMALink(s) can greatly speed up the shuffling time compared with traditional scale-out architecture with slower Ethernet connections.

5.5.3 Performance Comparison with multiple NUMA machines

In this subsection, we show the performance comparison with multiple NUMA machines. We conduct three set of simulations. Similar to the previous simulations, in each set, each *map* task runs on one socket and has intermediate key-value pair results of 4GB. The simulation setup is shown in Table 5.3. Without losing generality, we assume each *reduce* task runs on one socket as well. All the NUMA machines are connected with each other in 10Gb/s Ethernet.

We compare our solution with random selection which is similar to default approach in MapReduce. In the random selection, we use the same number of NUMA machines to run *reduce* tasks for fair comparisons. However, we randomly choose the locations of NUMA machines for *reduce* tasks placement rather than considering the data locality in our approach.

Figure 5.2 shows the normalized data shuffling time in three sets of simulations. We can see that our approach described in Algorithm 2 outperforms the random selection. With the consideration of data locality in the data shuffling phase in our approach, less amount of intermediate data are transferred among NUMA machines. The data shuffling time can be reduced by 20% - 30%.

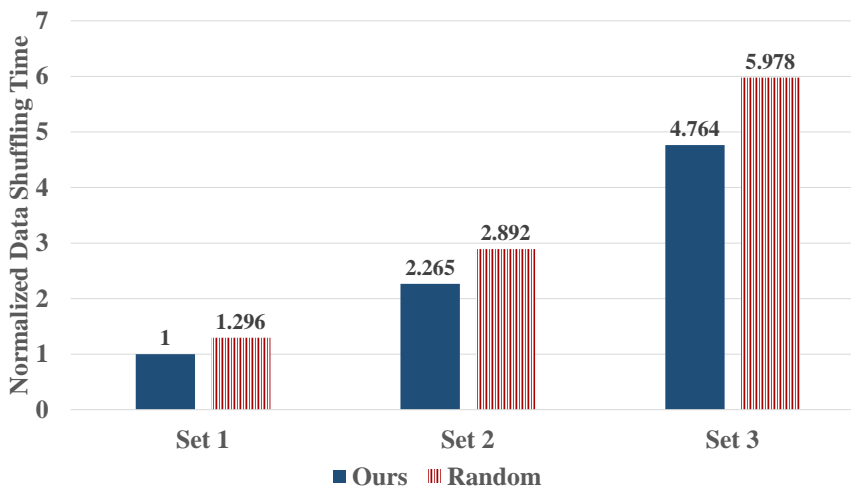


Figure 5.2: Comparison of normalized data shuffling time (multiple machines)

5.6 Conclusion

MapReduce is an important processing framework in HPC environment. As a significant phase, data shuffling accounts for large portion of running time in MapReduce jobs. It is necessary to reduce the data shuffling time in order to accelerate the entire running time of the MapReduce jobs. Scale-up architecture for MapReduce applications has been shown some advantages compared with traditional scale-out machines. With more powerful CPUs and larger memory, the performance can be improved in scale-up architecture. NUMA machine provides multi-processor, multi-core architecture and large shared memories for computing with NUMALink(s) for interconnections.

In this work, we use SGI UV 2000 machine as an example to investigate the data shuffling in scale-up architecture. With NUMALink, data can be quickly transferred. We propose a topology-aware reducer placement algorithm that can accelerate the shuffling time. Furthermore, we extend our approach to a larger computing environment with multiple NUMA machines. Experiment results show the performance results.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Big Data has drawn many people's attention. Research in Big Data has been investigated prevalently. In Big Data applications, huge amount of data are being generated all the time. With the rapid growth of data, it is important to manage and process data efficiently. In this thesis, we present efficient data management and processing in Big Data applications.

In Chapter 2, we show the background and related work. Nowadays, unstructured data is a significant portion of the entire data amount. Object Storage and NoSQL databases are designed to provide better performance and flexibility to deal with the unstructured data. Key-value store is a simple but efficient type of NoSQL database. As a novel active disk recently developed, Kinetic Drive can run key-value operations by itself with its CPU, RAM and built-in LevelDB. It greatly changes the storage stack by providing direct Ethernet based data access. The application can access Kinetic Drives via their IP addresses without going through a separate storage server.

MapReduce has been widely used for data processing in many applications. Data shuffling in MapReduce usually takes a long time to finish. Recently, scale-up computing architecture for MapReduce framework has been developed to provide better performance. As an example of scale-up computing architecture, NUMA machines provide better computing capability with powerful CPUs and large memories. They also have internal NUMALink(s) that are faster than Ethernet connections.

In Chapter 3, we present the data management of key-value store system using Kinetic Drives. A metadata server is necessary to manage data stored in the Kinetic Drives. In this work, we discuss the data allocation schemes for a large amount of key-value pairs and map them into Kinetic Drives. We investigate the *key* indexing schemes and propose several data migration approaches for different *key* distributions.

In Chapter 4, we design the Kinetic Drives based searchable key-value store system. In many applications, a user is not only interested in fetching a key-value pair by a *key*, but also it often searches data by providing metadata information. In this work, we focus on users' attributes search requests. We design a *key* generation method and an indexing scheme to support the searchable key-value store system.

In Chapter 5, we investigate the data processing in MapReduce with scale-up NUMA architecture. In this work, we show the advantage of the NUMAlink(s). We consider the topology of computing blades in NUMA machines and take advantage of it to expedite the data shuffling in MapReduce. We propose a reducer placement algorithm and extend our work to multiple NUMA machines.

6.2 Future Work

An immediate extension of this thesis could consider the bandwidth requirement of the key-value store system using Kinetic Drives. Each Kinetic Drive has its sustained bandwidth limit. Users' data access requests on the drives should not exceed their limits. In real applications, some data in certain *key* ranges are more popular and have more data access frequencies, while other data may be accessed less frequently. Also, data amount in different *key* ranges are various. Considering these factors including disk bandwidth, imbalance of data amount and different access frequencies in the practical deployment, it is very important to design a data management scheme for a large-scale key-value store system with Kinetic Drives. It would be interesting to investigate this topic beyond this thesis.

A broad extension of this thesis is to consider applications in Internet of Things (IoT) environment [74][75][76][77]. Recently, many IoT applications have greatly changed people's work and life. Key-value pair is a popular data format in IoT. As we discussed, Kinetic Drives are novel devices and particularly designed for key-value pairs. They

also have in-storage data processing capability. Some work of traditional storage server can be replaced by Kinetic Drives.

A new challenge could emerge if Kinetic Drives for data storage were brought into IoT environment. From the architecture design to data management schemes for IoT with Kinetic Drives, it is very interesting to consider this research direction. It would be an opportunity to investigate and pursue these related research topics.

References

- [1] 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/>.
- [2] Manas Minglani, Jim Diehl, Xiang Cao, Bingzhe Li, Dongchul Park, David Lilja, and David Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. Submitted to IEEE 25th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2017).
- [3] SGI UV 2000 system user guide. <http://techpubs.sgi.com/library/manuals/5000/007-5832-001/pdf/007-5832-001.pdf>.
- [4] Big data, https://en.wikipedia.org/wiki/big_data.
- [5] Big data, <https://www.ibm.com/big-data/us/en/>.
- [6] Emc digital universe with research & analysis by idc, <https://www.emc.com/leadership/digital-universe/index.htm>.
- [7] Nosql databases, <http://nosql-database.org/>.
- [8] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.
- [9] Marc Seeger and S Ultra-Large-Sites. Key-value stores: a practical overview. *Computer Science and Media, Stuttgart*, 2009.
- [10] Redis, <https://redis.io/>.

- [11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
- [12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2012.
- [13] Mike Mesnier, Gregory R Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.
- [14] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*, pages 119–123, 2005.
- [15] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.
- [16] Hyeran Lim, Vikram Kapoor, Chirag Wighe, and David 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*, 2001.
- [17] Erik Riedel, Christos Faloutsos, and David Nagle. Active disk architecture for databases. Technical report, 2000.
- [18] Kinetic open storage project, <https://www.openkinetic.org/>.
- [19] Kinetic hdd, <http://www.seagate.com/enterprise-storage/hard-disk-drives/kinetic-hdd/>.
- [20] <http://www.seagate.com/>.
- [21] Leveldb, leveldb.org.
- [22] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

- [23] Couchdb, <http://couchdb.apache.org/>.
- [24] MongoDB, <https://www.mongodb.com/>.
- [25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
- [26] Hbase, <https://hbase.apache.org/>.
- [27] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th Symposium on Operating Systems Design & Implementation (OSDI'04)*, 2004.
- [28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [29] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of the ACM SIGCOMM*, 2011.
- [30] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? In *Proc. of ACM SoCC*, 2013.
- [31] K Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Optimization techniques for “scaling down” Hadoop on multi-core, shared-memory systems. In *EDBT*, 2014.
- [32] K. Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Hone: “scaling down” Hadoop on shared-memory systems. *Proc. of VLDB Endow.*, 6(12):1354–1357, August 2013.
- [33] Technical advances in the SGI UV architecture, <https://www.sgi.com/pdfs/4192.pdf>.

- [34] Christoph Lameter. NUMA (non-uniform memory access): An overview. *ACM Queue*, 11(7):40–51, July 2013.
- [35] Hyperdex, <http://hyperdex.org/>.
- [36] Robert Escriva, Bernard Wong, and Emin Gün 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*, pages 25–36, 2012.
- [37] Document stores, <https://db-engines.com/en/article/document+stores>.
- [38] Document-oriented database, https://en.wikipedia.org/wiki/document-oriented_database.
- [39] Column (data store), [https://en.wikipedia.org/wiki/column_\(data_store\)](https://en.wikipedia.org/wiki/column_(data_store)).
- [40] Graph database, https://en.wikipedia.org/wiki/graph_database.
- [41] Arangodb, <https://www.arangodb.com/>.
- [42] Neo4j, <https://neo4j.com/product/>.
- [43] Key-value database, https://en.wikipedia.org/wiki/key-value_database.
- [44] A survey of modern key-value stores. https://wiki.csc.calpoly.edu/csc560/raw-attachment/wiki/Cronin_Bibliography/Cronin_survey.pdf.
- [45] Key-value stores, <https://db-engines.com/en/article/key-value+stores>.
- [46] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.
- [47] Xingliang Yuan, Xinyu Wang, Cong Wang, Chen Qian, and Jianxiong Lin. Building an encrypted, distributed, and searchable key-value store. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS '16*, pages 547–558, New York, NY, USA, 2016. ACM.

- [48] B. Y. Zhao, Ling 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.*, 22(1):41–53, September 2006.
- [49] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In Keith Marzullo and M. Satyanarayanan, editors, *SOSP*, pages 188–201. ACM, 2001. *Operating System Review* 35(5).
- [50] Antony I. T. Rowstron and Peter 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*, pages 329–350, 2001.
- [51] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore 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*, pages 46–66, 2001.
- [52] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari 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*, pages 149–160, 2001.
- [53] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, pages 1–14, 2003.
- [54] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.
- [55] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.

- [56] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 207–219, 2015.
- [57] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael 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*, pages 429–444, 2014.
- [58] Mohammad Hammoud and Majd F. Sakr. Locality-aware reduce task scheduling for MapReduce. In *Proc. of the IEEE Third International Conference on Cloud Computing Technology and Science (CLOUDCOM'11)*, 2011.
- [59] Long CAI Kokula Krishna Hari K, Vignesh R and Rajkumar Sugumaran. Big data - reduce task scheduling. In *Proc. of International Conference on Systems, Science, Control, Communication, Engineering and Technology*, 2015.
- [60] Chen He, Ying Lu, and David Swanson. Matchmaking: A new MapReduce scheduling technique. In *Proc. of the IEEE Third International Conference on Cloud Computing Technology and Science (CLOUDCOM'11)*, 2011.
- [61] Engin Arslan, Mrigank Shekhar, and Tevfik Kosar. Locality and network-aware reduce task scheduling for data-intensive applications. In *Proc. of the 5th International Workshop on Data-Intensive Computing in the Clouds (DataCloud'14)*, 2014.
- [62] P. Qin, B. Dai, B. Huang, and G. Xu. Bandwidth-aware scheduling with SDN in Hadoop: A new trend for big data. *arXiv:1403.2800*, March 2014, 1403.2800.
- [63] Anupam Das, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Curtis Yu. Transparent and flexible network management for big data processing in the cloud. In *Proc. of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*, 2013.

- [64] Jingui Li, Xuelian Lin, Xiaolong Cui, and Yue Ye. Improving the shuffle of Hadoop MapReduce. In *Proc. of the IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM'13)*, 2013.
- [65] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *Proc. of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR'13)*, 2013.
- [66] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. of the IEEE International Symposium on Workload Characterization (IISWC'09)*, 2009.
- [67] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular MapReduce for shared-memory systems. In *Proc. of the Second International Workshop on MapReduce and Its Applications (MapReduce'11)*, 2011.
- [68] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. of VLDB Endow.*, 5(10):1064–1075, June 2012.
- [69] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proc. of the 4th Annual International Conference on Systems and Storage (SYSTOR'11)*, 2011.
- [70] C. McCurdy and J. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. of IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*, 2010.
- [71] Spark. <http://spark.apache.org/>.
- [72] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proc. of the 1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP'12)*, 2012.
- [73] Intel memory latency checker v2. <https://software.intel.com/en-us/articles/intel-memory-latency-checker>.

- [74] Karen Rose, Scott Eldridge, and Lyman Chapin. The internet of things: An overview. *The Internet Society (ISOC)*, pages 1–50, 2015.
- [75] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.
- [76] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.
- [77] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.