High-Performance and Cost-Effective Storage Systems for Supporting Big Data Applications

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Zhichao Cao

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

Prof. David H.C. Du

July, 2020

Acknowledgements

My Ph.D. study at the University of Minnesota is a fantastic, impressive, incredible, and a happy journey in my life. I grew up from a young undergraduate student to a real researcher who enjoys exploring and solving new research issues and challenging problems. To many people I owe thanks in this process of learning and training, which I sincerely appreciate. Your supervision, advice, guidance, help, and love are the most precious treasure for me and I will bring them together with me to start my next journey.

I would like to express my sincere appreciation to my advisor, Prof. David H.C. Du for his unreserved supervision, advice, and training. His high-level thinking, critical thinking, research methodologies, research sense and taste, and hard-working educate and improve me in research, social, and daily life. I'm always proud, honored, and lucky to be his Ph.D. student. At the beginning of my study, it was a hard time for me to transfer from a passive learner to an active researcher who is able to define the problem, clarify the challenges, explore all the possible solutions, and come up with a scheme with better tradeoffs. At that time, I even cannot discuss more than 3 pages of slides in a personal meeting with my advisor due to so many issues in my research and presentation. I will miss these meetings in his office which grow me up. After several years of learning and training, I started to enjoy the meeting time with him and understand why he always said "research is a kind of fun!". I can always get advice from him for my research projects and I was always surprised by his deep knowledge, insightful feedback, and sharp research sense. I'm so grateful for his kindness to support my whole Ph.D. study and my happy life in Minnesota.

I am grateful to my mom Guilin Lai, my dad Jiacheng Cao, and my grandmother Yuanwen Huang. With their support, encouragement, and help, I spent about 7 years aboard without any loneliness and vapidity. I will always miss their love and their

support to my Ph.D. study.

I would also like to thank Prof. Abhishek Chandra, Prof. Rui Kuang, and Prof. Chris Kim for serving as my committee members and for their invaluable comments and suggestions.

It is an honor for me to be a member of the Center for Research in Intelligent Storage (CRIS). I would like to thank CRIS members: Xiongzi Ge, Ziqi Fan, Hao Wen, Fenggang Wu, Baoquan Zhang, Bingzhe Li, Ming-Hong Yang, Chai-Wen Hsieh, Shiyong Liu, Yixun Wei, Huibing Dong, Xuchao Xie, Jim Diehl, Hebatalla Eldakiky, Jinfeng Yang, Yaobin Qin, Xiang Cao, Alireza Haghdoost, and Weiping He. I learned so much through the discussion and collaboration with them. Thanks for their help and support.

I am grateful to Siying Dong who mentored my summer internship and supported my research collaboration with Facebook. I am grateful to Christopher Dickson who was the mentor of my CRIS research project collaborated with Veritas.

Finally, I would like to thank NSF I/UCRC Center Research in Intelligent Storage and the following NSF awards 1439622, 1525617, 1536447, and 1812537 for supporting my research.

Dedication

To my parents Jiacheng Cao and Guilin Lai. To my grandmother Yuanwen Huang. To my aunt Wenli Huang, her husband Yunsong Wu, and her son Wenhan Wu. To my aunt Shufen Cao, her husband Yangbo Lv, and her son Hang Lv. To my aunt Shuxian Cao, her husband Heqing Huang, and her son Chenglin Huang. To my uncle Jialin Cao, his wife Ping Ren, and his son Jianwei Cao. To those who cared and supported me over the years

Abstract

Coming into the 21st century, the world innovations are shifting from the IT-driven (information technology-driven) to the DT-driven (data technology-driven). With the fast development of social media, e-business, Internet of Things (IoT), and the usage of millions of applications, an extremely large amount of data is created every day. We are in a big data era. Storage systems are the keystone to achieving data persistence, reliability, availability, and flexibility in the big data era. Due to the diversified applications that generate and use an extremely large amount of data, the research of storage systems for big data is a blue sky. Fast but expensive storage devices can deliver high performance. However, storing an extremely large volume of data in the fast storage devices causes very high costs. Therefore, designing and developing high-performance and cost-effective storage systems for big data applications has a significant social and economic impact.

In this thesis, we mainly focus on improving the performance and cost-effectiveness of different storage systems for big data applications. First, file systems are widely used to store the data generated by different big data applications. As the data scale and the requirements of performance increases, designing and implementing a file system with good performance and high cost-effectiveness is urgent but challenging. We propose and develop a tier-aware file system with data deduplication to satisfy the requirements and address the challenges. A fast but expensive storage tier and a slow tier with much larger capacity are managed by a single file system. Hot files are migrated to the fast tier to ensure the high performance and cold files are migrated to the slow tier to lower the storing costs. Moreover, data deduplication is applied to eliminate the content level redundancy in both tiers to save more storage space and reduce the data migration overhead.

Due to the extremely large scale of storage systems that support big data applications, hardware failures, system errors, software failures, or even natural disasters happen more frequently and can cause serious social and economic damages. Therefore, improving the performance of recovering data from backups is important to lower the loss. In this thesis, two research studies focus on improving the restore performance of

the deduplication data in backup systems from different perspectives. The main objective is to reduce the storage reads during the restore. In the first study, two different cache designs are integrated to restore the data chunks with different localities. The memory space boundary between the two cache designs is adaptively adjusted based on the data locality variations. Thus, the cache hits are effectively improved and it lowers the essential storage reads. In the second study, to further address the data chunk fragmentation issues which cause some of the unavoidable storage reads, a look-back window assisted data chunk rewrite scheme is designed to store the fragmented data chunks together during deduplication. With a very small space overhead, the rewriting scheme transfers the multiple storage reads of fragmented data chunks to a single storage read. This design can reduce the storage reads that cannot be lowered by the caching schemes.

Third, in the big data infrastructure, compute and storage clusters are disaggregated to achieve high availability, flexibility, and cost-effectiveness. However, it also causes a huge amount of network traffic between the storage and compute clusters, which leads to potential performance penalties. To investigate and solve the performance issues, we conduct a comprehensive study of the performance issues of HBase, which is a widely used distributed key-value store for big data applications, in the compute-storage disaggregated infrastructure. Then, to address the performance penalties, we propose in-storage computing based architecture to offload some of the I/O intensive modules from compute clusters to the storage clusters to effectively reduce the network traffic. The observations and explorations can help other big data applications to relive the similar performance penalties in the new infrastructure.

Finally, designing and optimizing storage systems for big data applications requires a deep understanding of real-world workloads. However, there are limited workload characterization studies due to the challenges of collecting and analyzing real-world workloads in big data infrastructure. To bridge the gap, we select three large scale big data applications, which use RocksDB as the persistent key-value storage engine, at Facebook to characterize, model, and benchmark the RocksDB key-value workloads. To our best knowledge, this is the first research on characterizing the workloads of persistent key-value stores in real big data systems. In this research, we provide deep insights of the workload characteristics and the correlations between storage system behaviors and big

data application queries. We show the methodologies and technologies of making better tradeoffs between performance and cost-effectiveness for storage systems supporting big data applications. Finally, we investigate the limitations of the existing benchmarks and propose a new benchmark that can better simulate both application queries and storage behaviors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Big Data

Coming into the 21st century, the world innovations are shifting from the IT-driven (information technology-driven) to the DT-driven (data technology-driven). Data is generated by all kinds of devices from everywhere on the earth. With the development and fast growth of social media, e-business, streaming, IoT (Internet of Things), and new applications and Apps, a huge amount of data is created, stored, and used every day. The incessantly generated data creates a new virtual world and the world is all about data. Billions of data records about human activities, photos, files, and videos reflect the evolution of the physical world and change the virtual world consistently. The data created from the virtual world influences the real world in return. Due to the extremely large volume of the data, we are currently in the *Big Data* era [1, 2, 3]. Everyone in the world is interacting with the big data virtual world through the laptop, smartphone, sensors, and other intelligent devices that are connected to the Internet.

Big data era creates a huge market and highly influences our daily life and the whole society. The global big data market size is about 138.9 billion in 2020 [4] and can achieve a compound annual growth rate of 10-18% in the next 5 years based on different forecast resources [5, 6, 7]. Importantly, big data will have a deep and significant effect on the global industry and world economy, which will improve the global GDP by $10 to $15 trillion in the next 10 to 15 years [8]. Big data is an important driving force of industrial upgrading and pushing the development of intelligent manufacturing [9, 10], smart city

[11, 12, 13], and smart living [14, 15].

In 2019, there is about 40 zettabytes (ZB) of data being created over the world [16]. With about a 30% compounded annual growth rate, the collective sum of the world's data will grow to 175 ZB in 2025 according to IDC prediction [17]. Big data era brings new challenges for the data storage industry and for the storage systems supporting big data. On one hand, a huge amount of data is used by different applications for various purposes such as decision making, real-time data processing, recommendation, AI system training, and revisited by many different users and services. Thus, **high performance** is the top requirement that needs to be achieved by the storage systems. On the other hand, storing an extremely large volume of the data and dealing with the fast growth of the newly created data causes very high storage costs and leads to a high economic pressure to IT companies. Therefore, achieving a good **cost-effectiveness** is also the top objective of designing and developing storage systems in the big data era. However, developing a high-performance and cost-effective storage system is very challenging and difficult, especially in such a large scale and globally distributed environment.

In order to effectively reduce the storage system latency and improve the throughput, fast storage devices such as flash-based SSD [18, 19, 20] and non-volatile memory (NVM) [21, 22, 23, 24] are preferred by the big data companies [25, 26]. However, the per gigabytes (GB) storage space prices of these fast storage devices can be several times or even tens times higher than that of the traditional hard disk drives (HDD) based storage systems [27]. The storage system investment increases exponentially over the last few years due to the high demands of the fast storage device based storage systems. Even worse, the volume of data to be stored also increases at a very fast speed. It creates a high investment pressure for big data companies to purchase and construct data centers with larger and expensive storage systems. Therefore, the researches of designing a high-performance and cost-effective storage system for big data have significant social values and economic values, which is the main focus of this thesis.

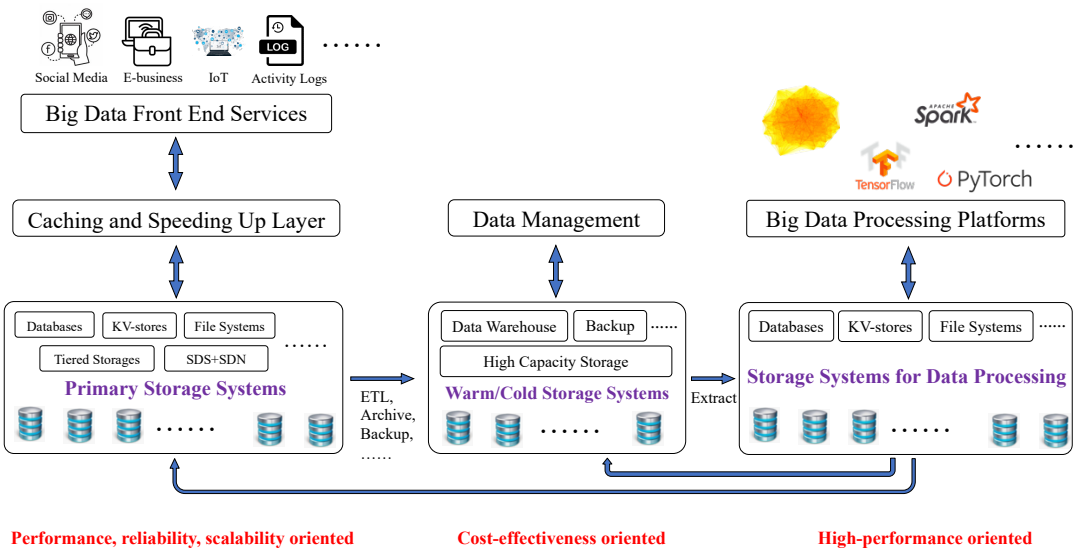


Figure 1.1: A typical big data infrastructure with different storage systems

1.2 Storage Systems for Big Data

There are different types of storage systems being used in big data infrastructure as shown in Figure 1.1. The research focuses and optimizing objectives of these storage systems are very different. Based on the usage and storage purposes of the data, we classify the storage systems in big data infrastructure into three categories: primary storage systems, warm/cold storage systems, and storage systems for data processing. There are various storage techniques being used in these storage systems and their designing focus can be very different. We first present the general introduction of the three categories of storage systems. Then, some of the storage techniques, such as tiered storage, data deduplication, and key-value stores, that our research focuses on are summarized. At the same time, the benefit, issues, and challenges of applying these storage techniques in storage systems for big data are discussed.

Primary Storage Systems Services and applications such as social media, e-business activities, and IoT services create a large amount of data in each minute. These data (called *primary data*) is persistently stored by databases, file systems, key-value stores, and object storages based on different data interfaces, such as SQL, file, NoSQL,

and object-based interfaces, required by different upper-layer applications. These storage systems are called *primary storage systems* which store the data from business production related services. Different types of storage systems are used independently, or they are integrated together. For example, on one hand, key-value stores are used as the embedded storage engines for SQL databases. On the other hand, key-value stores persist their data as large files on the file system. The metadata service of object storage systems uses key-value store to store the object metadata and the objects are maintained as files in a separate file system. Since primary storage stores the data closely related to the users and applications, the data scale is extremely large. Due to the importance, usage, and the scale of primary data, primary storage systems usually require high performance, high reliability, and high scalability.

In primary storage systems, file systems and key-value stores are two important and fundamental storage systems. Any type of data can be stored as files and files are easily managed with directories and namespaces. File systems are almost used everywhere in big data infrastructure. It is the main bridge between low-level storage such as disk drives, flash-based storage systems, or tiered storages and high-level storage systems such as key-value store, databases, graph data store, and object storage. By integrating high-performance and low-performance storage devices into single block storage, tiered storage makes a good tradeoff between performance and cost, which is preferred in the big data primary storage systems. Key-value stores usually store a large number of small size unstructured data. The key is used to access the data in the value portion. Due to the simple interfaces, high performance, and flexibility, key-value stores become popular for big data applications to maintain raw data, processed records, small models, and even logs.

Warm/Cold Storage Systems The cost of primary storage is usually high due to its high performance and high reliability requirements. Storing all the data in primary storage is not cost-effective, especially for the data that is not used for a long period of time (called *cold data*). These data can be migrated to a cheaper storage system with a higher capacity for long term persistence. Also, to ensure data reliability, all primary data needs to be backed up periodically. Moreover, using primary data for data processing is very important, which is usually I/O intensive. Data processing creates extremely heavy read and write requests during the processing time. Directly

accessing these data from primary storage systems can cause serious performance issues. Therefore, the data that is used for data processing purposes such as data mining and machine learning services is loaded to the data warehouse for better indexing and management. A warm/cold storage system is designed for the aforementioned data warehousing, archiving, and backup purposes. The storage devices with high capacity but low cost such as HDDs and tapes are used in warm/cold storage to lower the storage hardware cost.

Warm storage systems store the data that is used by the internal application and services. Since most of the data in warm storage systems is usually not directly used by online services and applications, warm storage systems do not need to achieve the performance as good as that of the primary storage systems. For example, the SQL table snapshots, videos, and machine learning training data batches are stored in the warm storage and periodically read by different applications. Usually, object storage systems and distributed file systems are used in warm storage systems.

Different from warm storage systems whose data is frequently used by the internal applications, cold storage systems usually manage the archive data and backup data, which is rarely accessed. Due to the high content redundancy existing among the archive and backup data, technologies such as data deduplication and compression are developed and applied to eliminate the content level redundancy and further improve the storage space utilization. Therefore, cost-effectiveness is the main focus of cold storage. Although the data in cold storage is usually not accessed for a long period of time, the high read performance especially the high read throughput is required which is even more important in big data. For example, when the primary storage system was corrupted due to an unexpected failure or disaster, millions of users and applications will not be able to access the data. It is a fatal issue in the big data era: data is unavailable and all services are forced to be stopped. The backup data is read out to recover the primary storage. However, the longer recovery time is, the more economic loss and more serious negative social influence will be. The recovery time window should be as small as possible to limit the system failure penalties. Therefore, how to reduce the extremely high read overhead introduced by the space saving technologies such as data deduplication is one of the top research areas of cold storage systems.

Storage Systems for Data Processing Processing the raw data generated by

users and applications, and using machine learning (ML) and Artificial Intelligence (AI) services to further explore the data is the main focus of big data. Big data processing platforms such as Spark [28], TensorFlow [29], and Pytorch [30] provide data processing, training, prediction, and inferencing services for different purposes. A large amount of data is extracted from warm storage and processed on these platforms. During data processing, these platforms create a large volume of middle-stage data, training models, and refined data. Writing and reading these data either on warm storage or on primary storage can cause performance and cost issues. Therefore, high-performance storage systems (e.g., distributed file systems and distributed key-value stores) are needed to specifically support the data processing platforms. After a data processing job is finished, the generated data will be written back to other storage systems or directly deleted. Therefore, different from the primary storage systems which have data reliability and cost-effectiveness requirements and warm/cold storage systems which have high space saving demands, data processing platforms mainly require high performance from the storage system. How to design a data processing oriented storage system for big data creates a set of new research opportunities.

1.2.1 Tiered Storage and File System

In primary storage systems, tiered storage is a popular storage architecture to achieve high performance and provide large storage space with low cost [31, 21, 32, 33]. Tiered storage uses high-performance but also expensive storage devices such as SSDs as the fast tier to store the hot data, and uses low performance but cheap storage devices such as HDDs as the slow tier to store the cold data. Due to the good data access locality, most of the data accesses are on hot data and the majority of the data is cold data. When data hotness changes, data is migrated between the tiers. To achieve a high overall cost-effectiveness, the size of the fast tier is usually much smaller than that of the slow tier. The overall performance of tiered storage can be close to the pure high-performance devices based storage system, and the overall cost of tiered storage can be close to that of the slow tier.

Using tiered storage as the architecture of primary storage systems in big data also has issues. Usually, the data migration policies in tiered storage are based on the block level statistics and it directly migrates data blocks between the tiers in block level.

The upper layer applications such as file systems, databases, and key-value stores are unaware of the migration. Block-based tiered storage has better generality and does not require the changes from upper-layer applications. However, it cannot provide enough migration accuracy [32, 34], especially for big data applications which mainly read and write the data on files. Big data applications usually have good file access localities and thus managing tiered storage in file level can be an effective and attractive design.

However, a file system based tiered storage design has the following challenges. First, due to the extremely large volume of data to be stored, the storage space of the fast tier can be smaller than the size of hot data. Many hot files are forced to be stored in the slow tier which causes serious performance penalties. Second, there is a lot of data redundancy that exists among the files. How to effectively save the slow tier storage space is also challenging when data migration happens frequently. Third, the activeness of file changes more frequently due to the file access patterns variation of big data applications. For example, new files are more frequently accessed by users and applications. How to achieve effective file migration is essential but challenging.

1.2.2 Data Deduplication and Restore

Cold storage systems store the backup and archive data from primary storage systems. These data have high redundancy. Therefore, data deduplication is widely used during backup and archive systems to effectively reduce the storage space utilization [35]. During data deduplication, the original data is treated as a byte stream and cut into small data pieces called data chunks (e.g., 4 KB on average). So the original data can be treated as a sequence of data chunks. Then, an encryption function with an extremely low collision probability (e.g., SHA1 or MD5) is used to calculate the hash value of the data chunk content as the fingerprint of that chunk. Therefore, the fingerprint uniquely represents the data chunk. An indexing table maintains a mapping of all the stored data chunks from their fingerprints to their storing information [36].

For each newly generated data chunk, if its fingerprint cannot be found in the indexing table, it indicates that this chunk has never been stored before and is treated as a unique chunk. This chunk will be stored in the storage system and a new mapping entry of this chunk is created in the indexing table. If the fingerprint of a newly generated chunk is found in the indexing table, it indicates that the data chunk has been stored

before and the chunk is called a duplicated data chunk. After searching the indexing table, a metadata entry that contains the data chunk fingerprint and other meta information is created and stored separately. Consequently, the original data is replaced by a sequence of the metadata entries (called recipe), and only the unique data chunks are stored. If 10% of the data chunks of a file are unique chunks and 90% of the data chunks have already been stored before, we only need to store 10% of the original file's data and a recipe which is only about 1-3% of the original data size. Therefore, data deduplication effectively saves the storage space. Usually, for backup data, the data changes among different backup versions from the same source system are very small, data deduplication can save 95% or more storage space according to some deduplication products [35, 36, 37].

Usually, to improve the storage I/O efficiency, a number of unique data chunks (e.g., 1000 data chunks) are written to the storage system as one batch (called a container). The time of reading one container is compatible with that of reading a single data chunk if the storage system is HDD based [36]. When one data chunk is needed, the whole container is read out to memory. Since some nearby data chunks in the original file may be stored in the same container, reading out one container can retrieve multiple data chunks when assembling the original data.

However, data deduplication also introduces serious read performance issues. When the original data is needed, for example, a deduplicated file needs to be read out, the original data will be assembled back based on the information in its recipe. Based on the data chunk information in the metadata entries and their sequence in the recipe, the containers that store these chunks are read out and the file is assembled back. Since data deduplication breaks the original file into data chunks and these data chunks can scatter in different containers especially for the duplicated data chunks, restoring the original data back can be much slower than reading the original data with no deduplication due to a large number of container reads. Improving the restore performance of deduplicated data in cold storage is very important especially in big data systems [38, 39, 40]. It is an important and essential research to explore ways of improving the restore performance.

1.2.3 Key-Value Stores

Key-value stores are widely used by big data applications in primary storage, warm/cold storage, and supporting data processing. Key-value stores (KV-stores) access data based on {key, value} pairs [41]. Some storage engines such as RocksDB [42] and LevelDB [43] are persistent KV-store. Some large scale distributed systems such as HBase [44], Dynamo [45], Cassandra [46], and some object storage systems [47, 48] are the distributed KV-stores. Different from relational databases and file systems, KV-stores do not provide complex operations and interfaces. Therefore, KV-stores are usually used as the basic building blocks for other applications and systems. For example, RocksDB is used as the embedded storage engine for MySQL and metadata storage of Ceph (a distributed file system) [49, 50].

The main focus of KV-stores is to achieve high performance. However, big data brings new challenges for KV-store performance optimization. First, the big data infrastructure setup and system architecture are different. Typically, KV-stores write and read the data from the storage in the same host. In big data infrastructure, all data accesses need to go through the network which causes new challenges including network traffic, transient errors, and data reliability issues. Second, KV-stores are used by various applications for different purposes. General performance optimizations are not enough to satisfy the performance requirement. Therefore, understanding the key-value workloads from different applications in different storage systems is essential and important to make specific performance and cost optimizations for KV-stores and upper-layer applications.

1.3 Summary of Contributions

In this section, we first describe the four research issues we focus on in this thesis and then we discuss the significance and challenges of these issues. The four research issues are closely related to the theme of this thesis: a better tradeoff between performance and cost-effectiveness. Then, we summarize our five research projects which address the four issues in this thesis.

1.3.1 Research Issues

How to make better tradeoffs between performance and cost-effectiveness in storage tier-aware file systems? In primary storage systems, integrating a file system with a tiered storage is a potential way to better balance performance and cost-effectiveness. With rich application behavior information, the file system is able to achieve fine-grained data migration between tiers such that the storage space in different tiers can be more efficiently used. However, it is not enough to adapt to the workloads in big data. First, big data services and applications generate a lot of redundant data or data with high duplicated content. Storing these data in the slow tier still causes high costs. Second, the volume of hot data can easily go beyond the space of the fast tier, accessing hot data from the slow tier can cause performance degradation. Third, migrating a huge amount of data between two tiers due to data hotness changing from big data applications creates a heavy traffic, which can impact the overall performance.

How to better achieve the restore performance improvement and space-saving for deduplication in cold storage? In cold storage systems, we focus on improving the read performance of deduplicated data. As discussed before, temporally losing data access from primary storage can cause serious economic and social impacts for millions of users. Reducing the recovery time window via improving the read performance of deduplicated data is the top choice. Although deduplication read performance improvement has been investigated over 15 years, the potential has not been fully explored. After data deduplication, the data chunks of original data are scattered to different containers. During the restore, containers are read out to fetch the data chunks to be restored. The data chunks from the read-in container can be temporally cached in memory for future use. Due to the good data chunk locality, some of the cached data chunks will be used in the near future, which can avoid some of the container reads. However, due to the memory space limitation, some of the data chunks are forced to be evicted from the cache to reclaim the space for data chunks from the newly read-in containers.

The main issue here is how to decide the data chunk caching and eviction priority such that the container reads can be further reduced. However, it is a very challenging problem. First, different from most of the caching problems in which the optimization goal and caching granularity are the same (e.g, memory page cache), the restore engine

caches data chunks in memory but the optimization objective is to reduce the container reads. The granularity mismatch makes it a very complex optimizing problem and the existing caching algorithms are not as effective as expected. Second, different from all the existing caching problems which cannot know the future access patterns, the restore process is able to know which data chunks will be used during restore and the assembling sequence from the recipe. How to effectively use the "future" information in the recipe is challenging. Third, how to decide the data chunks to be cached or evicted?

In some cases, none of the caching schemes can help to reduce the container reads due to the data chunk fragmentation. For example, if the original file consists of 10 data chunks and these 10 data chunks are stored in 10 different containers, no matter how we design the cache, 10 container reads cannot be avoided. To address this issue, our research explores the solution in another direction: improving the data chunk locality via data chunk rewrite. In the same example, if 10 data chunks are all unique to the deduplication system, the 10 data chunks will be sequentially logged to one container (suppose one container can store 10 data chunks) and stored. There is no space-saving and the data deduplication ratio (original data size divided by the stored data size after deduplication) is 1. During the restore, we only need 1 container read. If 9 out of 10 data chunks are duplicated data chunks and the 9 data chunks have already been stored in 9 different containers, we need 10 container reads (9 for duplicated data chunks and 1 for unique chunk) to restore. The deduplication ratio is about 10. If we can select some of the fully scattered duplicated chunks and store them together in the same container again, we may lower the deduplication ratio for some degree. But we effectively reduce the container reads. In the same example, if we store 4 duplicated chunks with the unique data chunk in a container, the deduplication ratio is lower to 2 but the container reads are reduced from 10 to 6. It is the power of data chunk rewrite: making tradeoffs between space-saving and the restore performance.

The challenge of data chunk rewrite is to decide which duplicated data chunk to be rewritten such that the container reads can be effectively reduced with minimal space cost. First, during deduplication, it is nearly impossible to know how these duplicated data chunks will be used during restore. Evaluating the degree of data chunk fragmentation is hard. Second, the deduplication process is very fast, a complex and time-consuming algorithm is not applicable. Third, the caching scheme design

influences the rewrite efficiency. Caching effectiveness should be considered during the rewrite.

How to improve the performance of storage systems for big data processing without extra hardware costs? In a big data infrastructure, in order to achieve high availability and high flexibility, the compute clusters and storage clusters are disaggregated [51, 52]. All applications and services are running in the compute clusters. When data read/write is needed, the requested data is transmitted through the network, which can cause performance issues due to the heavy network traffic. The situation is even worse for distributed key-value stores like HBase due to its high data read and write amplification. That is, the data being transmitted through the network is much larger than the data being requested by the applications since the size of requested key-value pairs is usually much smaller than the data blocks being readout. Also, when writing the data, the persistent logging, backend garbage collection, and compaction which are the common designs in key-value stores will cause more data to be written to the storage clusters.

Using faster but more expensive storage, computing, and network hardware is a way of solving the performance issue, which requires high hardware investments. In order to effectively reduce the overhead introduced by the compute-storage disaggregation without extra hardware cost, a promising direction is to bring the data intensive services closer to the storage to avoid the transient overheads, which is called *in-storage computing*. We explore the potential of improving the performance of applications in compute-storage disaggregated infrastructure via reorganizing the workflow: offloading some of the data intensive modules to the storage cluster. We focus on the performance issues caused by the huge data traffic between compute clusters and storage clusters. HBase is used as an example to explore the performance issues and demonstrate the effectiveness of in-storage computing based solutions. Applying the in-storage computing concept in big data infrastructure is a new research direction. Many problems and issues such as data computing control logic, data path management, data transmission, and caching needs to be addressed.

How to better understand the KV-store workloads in different big data applications to help optimize the performance and cost-effectiveness? As mentioned above, a big challenge of KV-store performance improvement is the diverse

workloads in big data. Collecting and characterizing various KV workloads from big data applications is essential and has a significant influence on both industry and academia. However, the existing works of workloads collecting and analyzing focus on the block level, file system level, SQL level, etc, which are very different from the KV workloads in big data infrastructure [53, 54]. The proposed workload characterizations and models cannot be applied to the big data storage systems. Key-value based APIs are different from the interfaces of the aforementioned scenarios. If the workloads in big data storage systems can be collected, analyzed, characterized, and modeled in different I/O stack levels, it will help to further improve the performance and reduce the cost of the system. However, there are several difficulties: 1) It is difficult to design and implement a tracing tool in the existing big data storage systems; 2) It is very difficult to collect the workload from the production environments with high accuracy; 3) The methodologies to fully characterize the KV workloads need to be explored; 4) How to model and benchmark the KV workloads from big data applications have not been investigated.

1.3.2 Contributions

Five research studies covering the four research areas are presented in detail in this thesis. First, we propose and develop a tier-aware file system with data deduplication to address both performance and cost issues. Second, we propose a new caching scheme to address the data deduplication restore performance issue, which can effectively use future information in the recipe and be adaptive to the workloads. In another work, we use a sliding window to analyze the fragmentation level of deduplicated data chunks based on the near future and past information to assist the rewrite decisions for higher restore throughput. Third, we explore the potential of offloading the scanning module of HBase to the storage system and propose a new caching scheme for the in-storage computing based HBase. Finally, we characterized, analyzed, modeled, and benchmarked the RocksDB key-value workloads at Facebook from different big data services and applications.

1. TDDFS: a tier-aware data deduplication based file system In a primary storage system of big data infrastructure, due to the activeness difference of the data, storage tiering is an important architecture to achieve high performance and high cost-effectiveness. However, in a big data environment, more accurate data management

between the storage tiers and much lower storing costs are required, which cannot be achieved by the current block-based tiered storage designs. To address this issue, we proposed and developed a storage tier-aware file system with data deduplication. Two storage tiers with different performance and costs are managed by the same file system to improve the data migration accuracy. Importantly, data deduplication is applied in both tiers for different purposes. First, when files are migrated from the fast tier to the slow tier, all the files are deduplicated to eliminate the data redundancy, which further reduces the data storage cost and improves the space utilization of the slow tier. When files in the slow tier are accessed, they are migrated back to the fast tier with the deduplication structure. It can effectively reduce the data migration overhead and improve the space utilization of the fast tier.

2. ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching As mentioned before, during the deduplication process, the original data is partitioned into data chunks and only unique chunks are stored in containers. After deduplication, the data chunks of the original data are scattered over a lot of containers and a large number of container reads are required when restoring the original data back. It causes serious read performance issues. Since some of the data chunks in the same container are usually used within a short range of restored chunks, caching the data chunks from the containers being read-in can effectively reduce the container reads. To further improve the caching scheme with low memory space overhead, we designed and proposed the look-ahead window assisted chunk-based caching scheme. By applying future information stored in the recipe to the caching mechanism and using adaptive policies to make better trade-offs between caching overhead and cache hit ratio, the container reads are effectively reduced compared with the state-of-the-art caching schemes.

3. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance Using caching schemes to improve the read performance of deduplication systems also has its limitations. During the restore process, if the data chunks are used again and they are already evicted from the cache, container reads are still needed. These highly fragmented data chunks will cause extra container reads which cannot be effectively reduced via caching schemes. To address this issue, these highly fragmented data chunks need to be transferred into data chunks with a

good spatial locality. That is, we store these duplicated chunks again (rewrite) together with the newly generated unique data chunks. After rewriting these chunks, instead of reading out the containers that originally store these chunks, these data chunks are read out together with the nearby unique data chunks during the restore. In this way, data chunk rewriting further reduces the container reads which are left out by the caching schemes. However, the data chunk rewrite can lower the deduplication ratio. To accurately rewrite the data chunks and make better tradeoffs between the deduplication ratio and restore performance, we proposed and developed the look-back window assisted data chunks rewriting scheme (LBW). In this approach, each duplicated data chunk is evaluated by a sliding look-back window to decide whether this chunk should be rewritten or not. Compared with the existing rewrite schemes, especially container capping [55], LBW provides both the historical and future information for each duplicated chunk to make the rewrite decision. Moreover, by using the deduplication ratio reduction and container reads reduction to bound the rewrite threshold, the proposed approach can make better tradeoffs between the deduplication ratio and the restore performance.

4. IS-HBase: a in-storage computing based HBase architecture in big data infrastructure HBase is a distributed key-value store based on LSM-tree and it is widely used in data processing platforms such as Spark [28] and MapReduce [56]. Key-value pairs are sorted and stored in files called HFile. The basic I/O units of HBase are data blocks which are usually much larger than a single key-value pair. The queries of Get and Scan key-value pairs from HBase cause high read amplification. Due to the compute-storage disaggregation in big data infrastructure, a high read amplification causes a huge amount of network traffic and it brings in performance issues. To address the performance issue of HBase in the compute-storage disaggregation infrastructure, we proposed a in-storage computing based HBase architecture. Instead of reading data blocks to the compute cluster, part of the read module called *scanner* is offloaded to the servers in the storage cluster. By preprocessing the data blocks in the storage servers and only transmitting requested key-value pairs to the compute cluster, the network traffic is effectively reduced. Offloading the scanner module to the storage cluster causes challenges for caching. The original block-based cache cannot be used. We further proposed the adaptive hybrid cache scheme, which caches both key-value

pairs and data blocks to achieve a high cache hit ratio. The cache space boundary between key-value pairs and data blocks is dynamically changed to maintain a high cache space utilization when the workload changes.

5. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook To help design the storage systems for big data applications, we need a better understanding of the big data workloads in the real production environment. By capturing, analyzing, characterizing, and modeling the workloads, we are able to know the limitations of the existing storage systems for big data applications and propose novel solutions to improve the system performance and cost-effectiveness. To achieve this, we collected the key-value query level workloads of RocksDB at Facebook. It covers the key-value store use cases in the primary storage system, warm storage system, and storage system for data processing platforms. We developed a set of tools to collect, replay, analyze, model, and benchmark the workloads from the three typical use cases. We explored some important insights into the workloads such as query compositions, key-value pair hotness patterns, and key-space locality. We further explored the limitations of widely used key-value store benchmark YCSB [57] when testing the storage I/Os under the key-value store. Based on the workloads models, we developed a new benchmark that can better emulate both the key-value query level workloads and the storage I/Os than those of YCSB.

In general, designing and developing high-performance and cost-effective storage systems to support big data applications is a very important research area in the next 10 to 20 years. As the boundary of big data quickly expands due to billions of intelligent devices being used, more data will be created and stored. At the same time, thousands of different applications and services are developed to use these data to better serve our daily life. In this thesis, the five pieces of research show the methodology and value of making better tradeoffs between the performance and cost-effectiveness from different perspectives for storage systems in big data infrastructure.

1.4 Thesis Outline and Previously Published Work

The rest of this thesis is organized as follows. In Chapter 2, we will summarize the background of big data infrastructure, new storage devices, tiered storage systems, data

deduplication, and key-value stores. Chapter 3 presents the tiered-aware file system with data deduplication. Chapter 4 discusses a novel way of improving data deduplication restore performance with a hybrid workload adaptive caching scheme. The look-back window assisted data chunk rewrite is another approach to improve the restore performance, which is described in Chapter 5. In Chapter 6, the in-storage computing based HBase is introduced in detail to show the value of offloading I/O intensive module of key-value stores to the storage cluster in big data infrastructure. The characterizing, modeling, and benchmarking of RocksDB key-value workloads at Facebook's big data infrastructure is presented in Chapter 7. Finally, the thesis is concluded and the future research vision is presented in Chapter 8.

This thesis including the materials from several of our previous publications [24, 58, 59, 60, 61, 62, 63, 64, 65]. Specifically, Chapter 3 includes material from [58]. Chapter 4 revises material from [59] and Chapter 5 revises from [60]. Chapter 6 revises material from a paper that is under review. One should be able to look at the paper by the forthcoming citation: "Zhichao Cao, Huibing Dong, Yixun Wei, ShiyongLiu, and David H.C. Du. IS-HBase: A New In-Storage Computing based HBase Architecture with I/O Offloading and Self-Adaptive Caching". Chapter 7 revises material from [61].

Chapter 2

Background

In this section, we will first introduce the big data infrastructure, including the basic system high-level overview, requirements from big data applications, and the potential research issues. Then, as the building block of the storage system, the new storage devices and the tiered storage are presented. In the end, data deduplication and key-value stores are introduced.

2.1 Big Data Infrastructure

The big data infrastructure [54] is a very complex system. It includes all the infrastructure components including network, computing, and storage. Each component is used in different levels and granularities and they are fully distributed. In a big data infrastructure, data is the center. All components are constructed related to how to transmit data, how to use data, and how to store data [53]. If we look at the life cycle of data in a big data infrastructure, we can have a clear overview.

First, services and applications generate the data at the beginning. On one hand, users are using services and applications all the time, data is generated and collected at the front end of the infrastructure. On the other hand, internal applications also generate a huge amount of data. For example, data is also generated by processing the original raw data from users [28]. The front end computing servers cannot persistently hold the data. All the newly generated data is stored in different storage systems such as databases, key-value stores, file systems, and object storage systems. We call these

storage systems the *primary storage systems*. Network connect users, applications, and storage systems [66]. Primary storage systems mainly support the regular data operations from users and applications. Primary storage usually stores a huge amount of data. Therefore, data storing cost and performance needs to be considered at the same time.

Besides storing data in the primary storage systems, data should also be stored in other places again for the following reasons. First, not all the data is used by users and applications after the data is stored. Data in big data scenarios usually has a very good temporal locality. That is, new data is more actively used than the old data. For example, in a machine learning based recommendation application, the recently generated user behavior data is used for training and prediction instead of using the data that has been generated years ago [67]. Storing those data in the primary storage systems wastes the storage space and increases both cost and overhead. Therefore, most of the cold data is migrated to storage systems with very low storing costs, which are called the *warm storage systems* in the big data infrastructure. Warm storage systems usually use low-cost storage devices such as HDDs.

Second, machine learning and other data processing applications read a huge amount of data. If they directly read the data from primary storage systems, it can cause unstable and spiky workloads [68]. It can cause serious performance issues for users and applications. Therefore, the warm storage system is also used to store the data which is used for backend data processing, for example supporting data warehouses. In this case, applications have high throughput requirements on data read and write.

Third, data reliability is very important for big data. Since the size of data being stored in the primary storage systems is extremely huge, there are millions of servers and storage devices being used. Due to such a large number of servers and devices, hardware and software failures can happen very frequently which can potentially cause data loss [69]. Moreover, accidents like earthquakes, power failures, and data center accidents can happen at any time. It is essential to backup all the data from the primary storage system to the storage systems with very low storing cost which is called *cold storage systems* [70]. Backup can happen daily, weekly, or even hourly based on different requirements. Data changes between two backups can be very small. Technologies such as delta backup [71], data deduplication [36], and compression [72] are widely used to

improve warm storage space utilization.

Data itself is the center of the big data era. After the data is stored, it is widely used by different applications such as data batch processing applications (e.g., Spark [28]), machine learning services, data mining services, AI training/inference services, and graph processing applications. Data is usually read out from the warm storage systems to the data processing platforms. Although most of these services use the large memory space to speed up the processing, the storage system is very important in these platforms. First, the data being processed can be even larger than the memory space, some data needs to be temporarily stored in the persistent storage for further processing. Second, the temporally generated data, for example, the RDD in Spark, cannot be cached in memory. A fast storage system is required to store these data. Third, the data processing results such as the AI/ML models and newly generated data after processing should be persistently stored to support other applications [29, 30]. The space requirement of the storage system for data processing platforms is much smaller than the primary storage systems and warm storage systems. However, it has higher performance requirements than the other two such as high throughput and low latency.

The network is also important in big data infrastructure [73]. On one hand, due to the extremely large scale of data and in order to support a large number of big data related applications, compute and storage clusters are disaggregated to achieve high scalability, flexibility, and reliability [74]. The high-speed network is needed to set up the connections between compute and storage clusters [75]. On the other hand, big data infrastructure is highly distributed. Data is created all over the world and applications are also geographically distributed. Network setup the connection between data sources, data storage, and data consumers. However, the network bandwidth is fixed after it is set up. If applications generate heavy network traffic in a short period of time, it can cause serious performance issues. Moving computing close to storage or moving storage close to computing becomes a new area to explore in big data infrastructure [76, 77].

2.2 New Storage Devices and Tiered Storage

In recent years, different types of storage devices are developed such as Non-volatile Memory (NVM) [78], flash-based Solid State Drives (SSD) [79], Shingle Magnetic Recording based drives (SMR) [80], DNA based storage [81], and glass-based storage [82]. NVM is non-volatile and its data will be persistently stored after power-off, which makes it powerful but also brings in several challenges. Some studies use NVM as the whole memory and other studies combine DRAM with NVM to deliver better performance and storing density. SSD has a better performance than HDD. The performance of SSD can be up to 100 times higher than that of HDD, especially for random I/O. SMR extends the storage capacity of regular HDD, but it also suffers from the non-sequential write performance. DNA- and glass-based storage devices are explored and investigated in the recent 2-3 years, it is promising to store a huge amount of data in DNA- or glass-based storage devices with an extremely low cost.

By using different types of storage devices in a storage system, tiered storage is a way to make better tradeoffs between costs and performance. Fast and slow devices are used to construct tiered storage systems with distinct performance characteristics. Many storage companies deliver block-based tiered storage products, such as IBM's Tivoli [83], EMC's FAST [84] and Dell Compellent's Data Progression [85]. These products may consist of several tiers with different performance characteristics and cost-effectiveness to adapt to different levels of data activeness. Typically, in a two-tier storage system, a fast-tier has higher performance, higher cost, and smaller capacity while a slow-tier has lower performance, lower cost but bigger capacity [31]. The fast-tier is used for storing active or hot data blocks while the slow-tier is used to store less active or cold blocks. Due to the dynamic changing of data activeness, data blocks are migrated between the two tiers to make the most of data accesses happen on the fast-tier. Accordingly, the performance of the whole system can be close to that of the fast-tier, but its cost-effectiveness can be close to that of the slow-tier.

When deploying a file system on top of the tiered storage, some issues need to be addressed. A file system may easily collect certain information like file access patterns, file activeness, and file boundaries that can benefit data allocation and migration between different tiers. Although there are existing efforts focusing on passing high-level

hints to the lower level storage layer [86], the collected information cannot be easily passed to block level and used by the tiered storage management for efficient data migration. Besides, some file system related data like file metadata data, which is small but latency-sensitive, should never be migrated to the slow-tier. A tiered storage system without knowing the semantics of the file and data may migrate the file metadata to the slow-tier since the data chunk which contains the metadata has not been used for a long time and causes higher latency when a related file is accessed.

2.3 Data Deduplication Preliminary

Data deduplication was introduced to eliminate the duplicate network traffic [87] more than 20 years ago. Later on, it is widely used in the storage systems such as archiving and backup systems to improve the storage space utilization [36, 88, 35, 89, 90, 91]. Recently, data deduplication is also applied in the primary storage systems such as SSD array to make a better trade-off between cost and performance [92, 93, 94, 95, 96, 86, 97, 98].

In a data deduplication system, data is processed as a byte stream during the deduplication process. The byte stream is segmented into either a fixed size or variable size chunks via different chunking algorithms [99]. Typically, the average chunk size is from 2KB to 16KB. After chunking, a chunk ID (the hash value of the chunk) is generated based on the content of the data chunk. The chunk ID is used for looking up in a large globally indexed table to verify whether this data chunk has appeared and been stored before. If the chunk is unique (i.e., the chunk ID cannot be found in the indexed table), the chunk will be stored in an active container in memory and a new entry which stores the data chunk metadata (e.g., chunk size, chunk ID and other metadata) and chunk address (container address and offset) will be added to the indexed table. If the chunk exists, it will not be stored since this chunk is duplicated.

In either case, the corresponding chunk information is added to the recipe. When the active container is filled up, it will be written out to the storage and a new empty active container is created in the memory to accept the upcoming unique chunks. Since each container holds only unique data chunks, a data chunk identified as a duplicated chunk has actually been stored in a container created some time ago.

Restore the data stream back is the reverse process of deduplication. Starting from

the beginning of the recipe, the restore engine identifies the data chunk information sequentially, accesses the chunks either from memory or from storage, and assembles the chunks into an assembling buffer in memory. To get a data chunk from the storage, the entire container holding the data chunk will be read to memory. To access a duplicated data chunk, the restoring process may have to read out a container located far away from the last accessed container since the containers are stored sequentially based on the order of their creation. Once the assembling buffer is full, data is flushed out to the storage. Due to the size mismatch between the data to be restored and the total size of containers being readout, the restore performance is much lower than reading the data without deduplication. Different data chunk caching schemes and duplicate chunk rewriting schemes are proposed to improve the restore performance (e.g., container-based caching [100, 101, 102, 103], chunk-based caching [55, 104, 59, 55], and duplicated data chunk rewrite schemes [100, 101, 102, 103, 55, 60]).

2.4 Key-Value Stores

Key-value store (KV-store) is a type of data storage that stores and accesses data based on {key, value} pairs [41]. The applications, data structure, storage systems that can satisfy the simple data read/write interfaces can be defined as KV-stores. Hash table, hash-based cache, other data structures based on different tree structures are one category of KV-stores. Some storage engines such as RocksDB [42] and LevelDB [43] are also KV-stores, which is defined as a persistent KV-store. Some large scale applications such as HBase [44], Dynamo [45], Cassandra [46], and some object storage systems [47, 48] are the distributed KV-stores. Different from relational database and file systems, KV-store does not provide complex operations and power interfaces on data. Therefore, KV-stores are usually used as the basic building blocks for other applications and systems. Due to its simple interface requirement, KV-store usually achieves higher performance.

In big data, data is generated from different applications and users. The relationship between data is much weaker than the data written in file systems and relational databases. For example, the record of users using smartphone Apps, the statistics collected by the IoT devices, and the people's location information do not have explicit

connections. The powerful attributes of relational databases and file systems are not useful for those data. In contrast, these features can bring unnecessary overhead, such as the locking mechanism of the relational database and the indexing structure of the file system.

KV-store usually provides Get, Put, and Delete interfaces to support data read, write, and delete operations [42]. If the KV-store is hash-based, it usually does not support range query since the KV-store does not maintain data ordering based on keys. Most persistent KV-stores provide the range query since scanning the KV-pairs is required for a persistent KV-store such that applications are able to know the KV-pairs that are stored in the system. In order to support range queries, persistent KV-store is designed based on the data structure in which the KV-pair ordering can be maintained in the persistent storage, such as B-tree [105], LSM-tree [106], or Trie [107].

In big data infrastructure, KV-stores play very important roles. In the primary storage system, KV-stores are used to support different applications or used as storage engines for other storage systems such as relational databases, distributed KV-stores, and some object storage systems. In warm storage, KV-stores are usually used in metadata services or indexing for cold data due to its high performance. In big data processing platforms, KV-stores are used as a temporary storage for data being stored, processed, or small model parameters. KV-stores are also used to support high speed data read and write in big data processing platforms. Understanding the workloads of KV-stores in big data infrastructure is very important to optimize the configuration, designs, and development of KV-stores. Also, in big data scenarios, using KV-store also has different challenges such as how to deal with large scale issues, how to address the performance bottleneck caused by big data infrastructure, and how to integrate KV-store with other applications in a better way.

Chapter 3

TDDFS: A Tier-aware Data Deduplication based File System

3.1 Introduction

According to International Data Corporation (IDC), more than 19 zettabytes (billion terabytes) of various storage media will need to be shipped between 2017 and 2025 to meet world data storage demands [108]. The amount of data created by use of social media, e-business, and other large-scale IT systems continues to grow exponentially [109]. This growth will cause the scale of some future file systems to become extremely large with billions of files stored in a single system. Several file systems such as GPFS [110], ZFS [111], HDFS [112], and Ceph [113] are already designed to satisfy this extremely large-scale storage requirement. In these file systems, except for a small number of active files that are used very frequently, most files are not re-accessed for a long period of time after they are first written [114]. Keeping all files, even infrequently accessed ones, in high-performance storage negatively impacts the overall cost-effectiveness of the storage system [115]. Thus, we are looking for approaches to improve the file system performance while reducing the total storage cost of large-scale file systems.

In recent years, different types of storage devices have been developed such as Non-Volatile Memory (NVM), Solid State Drives (SSD), and hard drives that use Shingled Magnetic Recording (SMR). These devices are used to construct different storage tiers with distinct performance characteristics. Typically, in a two-tier storage system, the

fast tier has higher performance, higher cost, and smaller capacity while the slow tier has lower performance, lower cost, but bigger capacity [31]. The fast tier is used for storing active, or hot, data blocks while the slow tier is used to store less-active, or cold, blocks. Due to dynamic changes in data activity, data blocks are migrated between the two tiers to make more data accesses occur in the fast tier while most inactive files are stored in the slow tier. Therefore, the performance of the whole system can be close to that of the fast tier, but its cost-effectiveness can be close to that of the slow tier.

To manage storage tiers at the file system level and to achieve a good trade-off between performance and cost-effectiveness, we propose a Two-tier aware Data Deduplication-based File System called TDDFS. TDDFS allocates and migrates files between the two tiers. TDDFS conducts file-based activity evaluation and manages file-based migrations efficiently and accurately. Files are always accessed from the fast tier as in a regular file system to achieve high performance. An access to a file in the slow tier triggers the file to be migrated to the fast tier. TDDFS also deduplicates the files that are going to be migrated from the fast tier to the slow tier. Since data deduplication identifies redundant data and replaces it with corresponding metadata, TDDFS reduces the amount of data being migrated to the slow tier and improves its space utilization.

TDDFS creates and operates on new files in the fast tier the same as files in normal file systems to gain the best performance. After a period of time, some less-active files are migrated to the slow tier. These files are deduplicated with a Content Defined Chunking (CDC) process, while their file metadata is still maintained in the fast tier. If a file (deduplicated) in the slow tier is requested by a file *open*, the file will be migrated back to the fast tier. We have designed special ways of handling the files to be migrated from the fast tier to the slow tier or vice versa. To reduce the amount of data to be migrated from the slow tier to the fast tier, TDDFS does not assemble the file back to its original state; instead, it keeps the structure of the data chunks generated by data deduplication. That is, the file recipe created during the deduplication is used to index the data chunks in the fast tier. Therefore, data chunks that already exist in the fast tier do not need to be migrated. In this way, the amount of data to be migrated from the slow tier to the fast tier is reduced. To further reduce the data to be migrated from the fast tier to the slow tier, TDDFS keeps track of whether or not an existing data

chunk in the fast tier has been modified. If a data chunk is not modified and a copy of the chunk exists in the slow tier, it does not need to be migrated from the fast tier back to the slow tier. Consequently, the amount of data to be migrated between the two tiers is reduced 80% to 90% compared with the design without deduplication or the design that only applies deduplication to the slow tier.

This research describes the architecture design, prototype implementation, and evaluations of TDDFS. We implemented TDDFS based on FUSE and compared it with other file system level tiering designs such as *AllDedup* (all files in both tiers are deduplicated) and *NoDedup* (no files in either tier are deduplicated). The evaluation results show that the overall performance of the proposed system is very close to that of a comparable file system with the *NoDedup* tiering design, and its cost-effectiveness is very close to that of the *AllDedup* design. This means TDDFS achieves close to the best overall performance and cost-effectiveness among various file-tiering designs for a two-tier storage system. For example, as shown in Section 5.6.1, the throughput of TDDFS can be close to or even better than that of *NoDedup*, and the per GB cost of TDDFS is only about 2% higher than that of *AllDedup*. Our key contributions include:

- By keeping deduplicated unique data chunks in both tiers and creating new files as regular files in the fast tier, TDDFS simultaneously achieves high performance and high cost-effectiveness.
- When a file is migrated from the slow tier to the fast tier, if some of the data chunks of the file already exist in the fast tier, these chunks will not be migrated. Thus, TDDFS reduces the transfer costs of migrating data between the slow tier and the fast tier.
- Data chunks are shared among reloaded files in the fast tier. Consequently, TDDFS achieves a good space saving in the fast tier (no duplicate data chunks) with only a small read/write performance penalty.
- TDDFS improves the deduplication performance and migration efficiency. This is because the unmodified data chunks of the reloaded files in the fast tier do not need to be deduplicated, and these chunks will not be migrated to the slow tier if they already exist in the slow tier.

The rest of the chapter is arranged as follows: We discuss both background and challenges in Section 3.2. Section 3.3 provides the proposed system design and architecture in detail, and our prototype implementation is described in Section 3.4. We evaluate and analyze the performance of our approaches in Section 5.6.1. The related work is reviewed in Section 3.6. Finally, we conclude our work in Section 3.7.

3.2 Background and Challenges

In this section, we first briefly discuss tiered storage, file activity, and file access locality. Then, the challenges of integrating data deduplication with file migration are discussed.

3.2.1 Tiered Storage

The management of tiered storage is usually achieved at the block level [85]. Typically, a tiered storage system uses either flash-based storage drives or high-performance hard disk drives as a fast tier and uses either low-performance hard disk drives or tape drives as a slow tier [83, 85, 84, 116]. The migration granularity in a tiered storage system is a data chunk with the size varying from 4KB to 1GB or even larger [85, 116, 84]. A smaller chunk size is usually more accurate and efficient for selecting data chunks to be migrated, but its maintenance cost is higher and it may cause more data fragmentation. A larger chunk size reduces the management overhead; however, the data migration efficiency and accuracy are relatively low.

Deploying file systems on top of a block-based tiered storage system is a straightforward approach to achieve good cost-effectiveness. As the number of files continuously increases, their corresponding data volume will go beyond the capacity of the fast tier. Therefore, a number of files have to be migrated from the fast tier and stored in the slow tier. In this case, data migration is completely managed by the block-based tiered storage system [32, 117, 31]. The physical location of a file is transparent to the file system, and existing file systems and applications can be directly used in a tiered storage environment without any changes.

When deploying a file system on top of a tiered storage system, some potential issues still need to be considered. First, since data activity measurement and data migration are chunk-based, the need to migrate one file may cause the whole data chunk to be

migrated. This would consume more storage I/O bandwidth of the two tiers than necessary. Similarly, a small but active file may prevent a big data chunk from being migrated to the slow tier. This would mean that the valuable space of the fast tier is not economically used. Second, a file system may easily collect certain information like file access patterns, file activity, and file boundaries that can benefit data allocation and migration between different tiers. However, the collected file system information cannot be easily passed to the block level and used by the tiered storage management system, though there are existing efforts focusing on passing high-level hints to low-level storage layers [86, 118]. Therefore, moving the migration management logic up to the file system level can potentially solve some of the aforementioned issues and make the management more efficient and accurate. Third, some file system related data like file metadata, which is small but latency sensitive, should not be migrated to the slow tier if space is adequate. A tiered storage system, without knowing the semantics of the data, may migrate the file metadata to the slow tier if the data chunk that contains the metadata has not been used for a long time. This will cause higher latency when the file is accessed later and for other file system operations that depend on the metadata (e.g., the *ls* command used to list directory contents).

Although file systems that can manage multiple storage tiers have been developed such as GPFS [119], ReFS [120], and Oracle HSM [121], determining policies that allow a file system to manage multiple storage tiers with good tradeoffs between performance and cost remains challenging. Deduplication adds another tradeoff. By deduplicating the files that are migrated to the slow tier, migration cost is reduced and the space utilization of the slow tier is improved. However, data deduplication brings a performance penalty especially when applications are accessing the deduplicated files. Thus, the challenges of integrating data deduplication with file migration need to be carefully addressed.

3.2.2 File Activity and Access Locality

To better understand file activity and access locality, researchers have measured and analyzed file system workloads. Most of these studies have similar findings on file activity and file access patterns [122, 123, 114]. Leung *et al.* [114] collected three months of file system workload traces from NetApp's data center used by more than

1500 employees. They discovered that more than 90% of the over 22TB of active storage was untouched over the three-month period. Of the files accessed, 65% were only opened once and 94% were accessed fewer than five times. Meyer *et al.* [124] collected file system traces from 857 desktop computers at Microsoft and focused on file modifications and file writes. Analysis of the month-long file traces showed that less than 10% of files were modified within the last 10 days, and more than 80% of files were not modified within the last month. From these studies, it is easy to conclude that in large-scale file systems most files are less active and infrequently accessed. It also implies that these files can be migrated to the slow tier to reduce hardware costs if the tiered designs are used.

Leung *et al.* [114] also found that more than 60% of file re-opens occurred within one minute of these files being closed. This result indicates that a file may be opened and closed a couple of times within a short period. Thus, it is reasonable to migrate files from the slow tier to the fast tier to achieve better performance when the files are accessed. In addition, file operations display locality. Based on the traces, Leung *et al.* concluded that about 21% of file operations accessed file data (reads or writes) while about 50% were strictly metadata operations. Thus, in a two-tier storage system, storing the metadata of hot files, or even all of the metadata if possible, in the fast tier can improve the overall performance. In general, all metadata plus the frequently accessed files are still a small portion of the total data in a file system. Therefore, storing these data in the fast tier can achieve good performance and high efficiency if the capacity of the fast tier is large enough. In another study, Roselli *et al.*[123] found that files tend to be either read-mostly or write-mostly, and the total number of file reads tends to be several times more than the total number of file writes. The proposed design of TDDFS incorporates these findings into tradeoffs between performance and cost-effectiveness for reads and writes.

3.2.3 Deduplication and Challenges of Integrating Deduplication with Migration

Data deduplication is a popular feature in many storage software products. By identifying and only storing unique data chunks, data deduplication considerably reduces the space required to store a large volume of data [125]. Data deduplication is broadly

classified into two categories: primary data deduplication and secondary data deduplication. Primary deduplication usually refers to deduplicating primary workloads, and secondary deduplication refers to using deduplication in backup or archival systems [126].

Both primary data deduplication and secondary data deduplication have similar processing operations [90, 93]. Typically, the data to be deduplicated forms a byte stream and is segmented into multiple data chunks according to a chunking algorithm. Fixed-size chunking and Content Defined Chunking (CDC) are two major types of chunking algorithms [99]. The former has a higher throughput, while the latter can achieve a better deduplication ratio (i.e., the original data size divided by the data size after deduplication, which is always ≥ 1). A chunk ID based on the fingerprint of the data chunk content is calculated by a cryptographic hash function (e.g., MD5, SHA-1, or SHA-256). Once a chunk is determined, the deduplication system searches for the chunk ID in the indexing table (i.e., the collection of all existing chunk IDs). If the chunk ID already exists, the data chunk is a duplicate and it will not be written to the storage again. If the chunk ID is new (does not exist), the data chunk is unique and will be stored. Then, a new indexing entry that maps this chunk ID to the physical location of the chunk will be created and added to the indexing table. Other mapping metadata (i.e., a file recipe) is created to map all data chunks (redundant or unique) to their stored locations in the same sequence as they were in the original byte stream such that it can be reconstructed in the future.

Although integrating data deduplication with file migration can improve the slow tier space utilization and potentially reduce migration cost, it also brings some performance issues such as high compute and memory resource utilization, high latency, and low throughput [127, 128, 129]. The chunking process, chunk ID generation, and chunk ID searches in the indexing table are time-consuming. Thus, data deduplication increases the I/O latency and impacts the throughput [55, 103, 89]. Caching a large portion of the indexing table in memory can increase the chunk ID search speed, but the memory space available for the file system or other applications will be reduced which may further degrade the overall performance. To address these challenges, TDDFS does not assemble the reloaded files (i.e., deduplicated files to be migrated from the slow tier to the fast tier) back to their original state in the fast tier. This avoids creating a file with

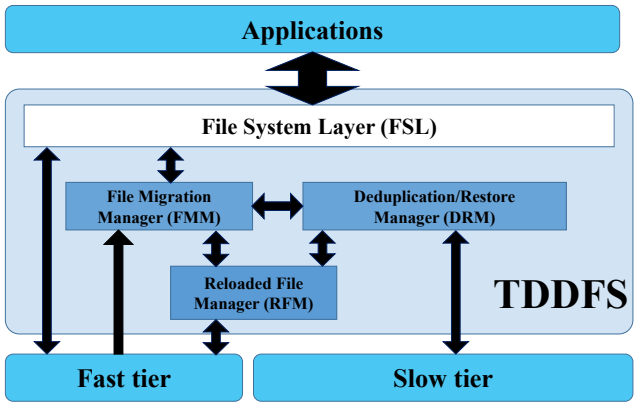


Figure 3.1: Overview of TDDFS architecture

many duplicate data chunks, since some data chunks may already exist in the fast tier. By tracking the updates to data chunks of a file in the fast tier, extra deduplication can be avoided if the data chunks are not modified. These concepts are discussed in more detail in Section 3.3.

3.3 System Architecture and Design

In this section, we first overview the architecture and important modules of TDDFS including the File System Layer (FSL), File Migration Manager (FMM), Deduplication/Restore Manager (DRM), and Reloaded File Manager (RFM). Then, four important and unique features of TDDFS are described. Finally, several major data operations are discussed. This description of TDDFS is intended to show the design principles of our proposed approach.

3.3.1 Architecture Overview

An overview of the TDDFS architecture is shown in Figure 3.1. The FSL provides a generic interface similar to that of a typical file system, so upper-layer applications can use TDDFS without any modifications. This FSL allows applications to directly read files from and write files to the fast tier until the files are migrated to the slow tier. Since all metadata is stored in the fast tier, the FSL processes all metadata requests directly on the fast tier. The FMM collects file information from the FSL to evaluate

the file activity and manage the migration process between the two tiers.

In addition to the FMM, two other modules, the RFM and DRM, cooperate to support the major operations of TDDFS. When file migration starts, the FMM selects the file migration candidates and, if these files are newly created, tells the DRM to read the candidate files from the fast tier as a byte stream to be deduplicated. After the files are successfully deduplicated and stored in the slow tier, their data is deleted from the fast tier. The file recipe that is used for future file restore and read/write operations is stored in the fast tier as part of the file metadata. If a deduplicated file in the slow tier is requested by a file *open* call from an application, the FSL tells the FMM to migrate the file from the slow tier to the fast tier (this action is called a reload). During this process, the DRM identifies, reads, and then writes the required unique data chunks from the slow tier to the fast tier. Note that some of the unique data chunks may already exist in the fast tier. These data chunks will not be reloaded. A unique feature of TDDFS is that the reloaded file references its data through an index of the unique data chunks in the fast tier. These data chunks may be shared by the same file or different reloaded files. Note that read or write requests may happen during a file reloading process; TDDFS is able to process the read and write requests concurrently with file reloading. Another optimization is that when a reloaded file is selected to be migrated from the fast tier to the slow tier again, unmodified data chunks do not need to go through the deduplication process. Only modified data chunks of a previously reloaded file that is now moving to the slow tier will be deduplicated by the DRM. File migration and reloading are transparent to the applications.

Based on the file's location and state, files in TDDFS are classified into three categories:

- **Newly created files:** These are files newly created in the fast tier. Their metadata and data are the same as files in a typical file system. Since there is no special operation needed for newly created files, their performance is considered similar to files in a typical file system deployed on the fast tier.

- **Deduplicated files:** These are files migrated from the fast tier to the slow tier that go through the deduplication process against the unique chunks already stored in the slow tier. Their metadata is stored in the fast tier; however, their unique

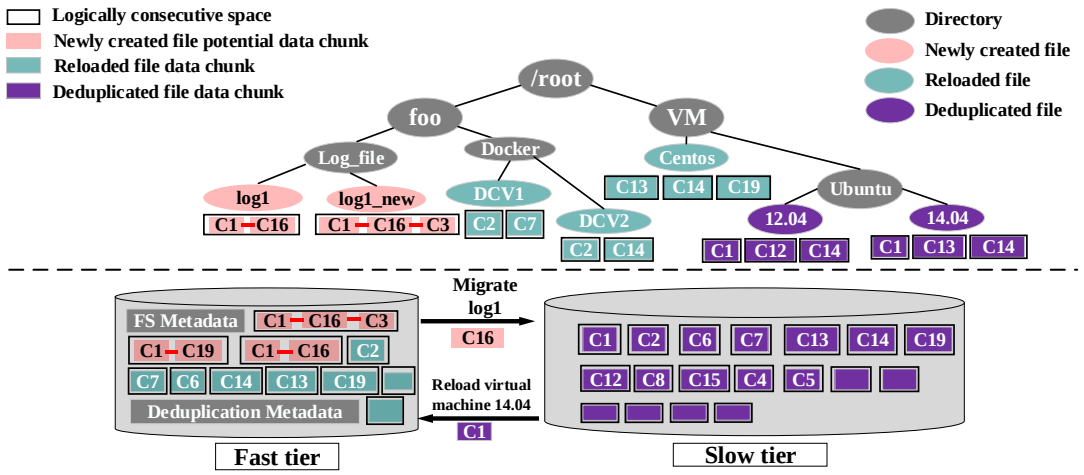


Figure 3.2: An example of file states and data layout of TDDFS

data chunks are stored in the slow tier without duplicates. TDDFS creates a file recipe for each file as part of the metadata, and the file recipe is used to access unique data chunks of this file in the slow tier.

- **Reloaded files:** These are files reloaded back from the slow tier to the fast tier. During the reloading, only unique data chunks that do not exist in the fast tier are reloaded back to the fast tier. Some data chunks in the fast tier are referenced by only one reloaded file, while others are referenced several times by one file or even concurrently referenced by multiple reloaded files. After reloading, the file recipe is updated with data chunk addresses located in the fast tier.

One example of different file states and their data layout in TDDFS is shown in Figure 3.2. In the figure, `log1` and `log1_new` are newly created files, and `log1_new` is a new version of `log1` with newly appended content. `DCV1`, `DCV2`, and `Centos` are reloaded files. `DCV1` and `DCV2` are the data volumes of Docker containers. Chunk C2 and C14 are shared by the reloaded files. `Ubuntu/12.04` and `Ubuntu/14.04` are deduplicated files. Since `log1_new` is a newly created file, its data content, indicated as three chunks C1-C16-C3, is stored in the fast tier in a logically consecutive space and managed as a regular file. Note that C1, C16, and C3 are the data chunks that will be partitioned by deduplication in the future, and C3 is the newly appended content based on `log1`. When

TDDFS decides to migrate the newly created file `log1` (which has content indicated by two chunks C1-C16) from the fast tier to the slow tier, the FMM reads the file data and passes it to the DRM. The DRM partitions the file into two data chunks, C1 and C16. Because C1 already exists in the slow tier, only C16 will be migrated and written to the slow tier. This saves I/O bandwidth and improves space utilization of the slow tier. `Ubuntu/12.04` and `Ubuntu/14.04` are deduplicated files and have two identical data chunks, C1 and C14, in both files. However, only one copy of C1 and C14 is stored in the slow tier. If `Ubuntu/14.04` (consisting of data chunks C1, C13, and C14) is reloaded back to the fast tier, only one data chunk, C1, needs to be reloaded to the fast tier. This is because C13 and C14 are parts of the reloaded file `Centos`, and they are already in the fast tier. This type of optimization reduces the I/O loads of both the fast tier and slow tier.

Given the above overview, the following four subsections present four features which make TDDFS special and achieve better tradeoffs between performance and cost-effectiveness.

3.3.2 File Migration Selection

As more newly created files and reloaded files are accumulated in the fast tier, the free space in the fast tier eventually becomes lower than a given threshold. At this point, some less-active files need to be migrated to the slow tier to reclaim space for more newly created or reloaded files. As discussed in Section 3.2.2, most files in a file system are less active and they will not be re-accessed for a long period of time. Therefore, migrating these files to the slow tier can potentially improve the overall performance of the file system. To ensure the performance of primary workloads, migration can be done during the off-peak time. Dedicated evaluations of file activity and file selection policies can be further exploited for file migration in TDDFS. However, this is not the focus of this research. Here, TDDFS is simply based on the file access recency to select file migration candidates. More aggressive or efficient policies will be explored in our future work.

For selecting migration candidates, the least recently used files are identified by the FMM and migrated to the slow tier until the free space of the fast tier is higher than a pre-determined threshold. To speed up the deduplication process of migration, files that are smaller than a *maximal chunk size* (e.g., 16KB) will not be chunked. In this

case, the whole file is treated as a single chunk and stored in the slow tier if it is unique. If these small files only occupy a small portion of the fast tier space and there is little probability of finding their duplicates in the slow tier, migrating and deduplicating these files has little performance and space benefit. Therefore, to further reduce the migration cost, if the cumulative total size of these small files is smaller than a pre-determined threshold (e.g., 5% of the fast tier's capacity), these small files will not be selected as migration candidates.

3.3.3 Integrating Deduplication with File Migration

When migrating a newly created file through deduplication, a file recipe is created and added to the file metadata to reference to the data chunks of this file. File recipes will be used when reloading deduplicated files to the fast tier, reading/writing reloaded files, and deduplicating reloaded files. Each metadata entry corresponding to a data chunk in the file recipe stores the following information: *(Chunk ID, Chunk Size, Chunk Address, Restore Flag, and Modification Flag)*. *Chunk ID* is used to find the corresponding Key-Value Pair (KV-pair) in the deduplication indexing table. In TDDFS, we use the hash value of the chunk as the chunk ID, and this value is related to the cryptographic hash function being used. A unique data chunk may exist in both the fast tier and the slow tier. If the unique chunk is referenced by a deduplicated file, the *Chunk Address* of this chunk in the file recipe points to the chunk location in the slow tier. If the unique chunk is referenced by a reloaded file, the *Chunk Address* in the file recipe points to the chunk location in the fast tier. A *Restore Flag* (RF) in the metadata entry is used to indicate whether this chunk has been restored to the fast tier or not. The RFs of deduplicated files are 0. After a data chunk of the deduplicated file is migrated to the fast tier, the RF of this chunk in the metadata entry is set to 1. One bit is used for the *Modification Flag* (MF) in the metadata entry of a data chunk in the file recipe. When a file is reloaded from the slow tier to the fast tier, the MFs of metadata entries of these migrated chunks are set to 0. During the file's lifetime in the fast tier, if one of its chunks is modified, a copy of the modified chunk is created in the fast tier. At the same time, in the metadata entry of this new chunk, the *Chunk ID* is cleaned and cannot be used to identify the original chunk. The *Chunk Size* and the *Chunk Address* will be changed accordingly to reference the new data chunk, and the MF of this chunk

Table 3.1: KV-pair Structure

key	Value
Chunk ID	chunk fast tier address: P1 chunk fast tier reference count: C1 chunk slow tier address: P2 chunk slow tier reference count: C2 chunk size: L Status: S

is set to 1.

In the indexing table, each unique chunk's indexing information is managed as one KV-pair as shown in Table 3.1. The chunk ID is used as the key, and other information is stored as the value. If we use 8-byte unsigned integers to store the addresses and reference counts and 4-byte unsigned integers to store the size and status, the size of a KV-pair is 24 bytes plus the chunk ID size. For one KV-pair, in a technique different from the previous deduplication works, we use two address pointers (one for the fast tier address and another for the slow tier address) and two reference counts (fast tier reference count and slow tier reference count) to indicate the chunk location and its usage in the fast tier and the slow tier. The fast tier (or slow tier) reference count indicates how many times the chunk appears in that tier's files. If the reference count of the chunk is 0, it will be deleted in the corresponding tier during garbage collection. We use the Status to indicate the storing status of the chunk in both tiers: existing in the fast tier only, existing in the slow tier only, or existing in both tiers. After garbage collection is applied in the fast tier or the slow tier, the Status of the deleted chunks is updated for corresponding KV-pairs.

When a newly created file is selected to be migrated to the slow tier, first, the file content is read as a byte stream and segmented into multiple data chunks via a CDC scheme. Then, the fingerprint of each chunk is generated using a cryptographic hash function (e.g., MD5, SHA-1, or SHA-256) and the fingerprint is used as its chunk ID. Each chunk ID is searched in the deduplication indexing table. If the data chunk cannot be found in either the fast tier or the slow tier, this chunk is new to the system and the data chunk will be written to the slow tier. A new KV-pair representing this new chunk is created and inserted into the indexing table. The slow tier reference count in

the new KV-pair is set to 1, and the data chunk location in the slow tier is stored in its slow tier address. If the data chunk exists in the fast tier but it does not have a copy in the slow tier, we need to store a copy of this chunk in the slow tier. At the same time, the slow tier reference count in the KV-pair of this chunk is set to 1, and the data chunk location in the slow tier is stored in its chunk slow tier address. If the data chunk already exists in the slow tier, the data of the chunk will not be stored. The slow tier reference count in the KV-pair of this chunk increases by 1. After the KV-pair of the data chunk is created or updated accordingly, a new metadata entry of the chunk is created and appended to its file recipe. Finally, after the whole file is processed, the metadata of the file is updated and the file is labeled as a deduplicated file. After that, the content of the file in the fast tier is deleted.

If any failure happens during the migration, the process will restart from the beginning. If all newly created files are migrated to the slow tier and there are no other newly created files being created, the whole fast tier space is occupied by reloaded files. In this extreme situation, the performance of TDDFS is close to that of the *AllDedup* design. The size of the file recipe is about 0.5% to 1% of the entire file size if we use 8KB as the average chunk size. If the size of the deduplicated files is very large, the total size of the corresponding file recipes can occupy a large portion of the fast tier. In this situation, there will be less space for reloaded files and newly created files, which can cause a performance penalty. To handle this case, a threshold is set for the cumulative total size of all the file recipes in the fast tier (e.g., 10% of the fast tier's space). If the total size of the file recipes is higher than the threshold, some of the least recently used file recipes will be migrated to the slow tier.

When a reloaded file is selected to be migrated from the fast tier to the slow tier, TDDFS sequentially processes the metadata entries in the file recipe of the file. For each metadata entry, TDDFS first checks the value of MF. For an unmodified chunk (MF is 0), we use its chunk ID to fetch the corresponding KV-pair from the indexing table. If the slow tier reference count is ¿ 0, the chunk has a copy in the slow tier. Its *Chunk Address* in the recipe is directly changed from the chunk fast tier address to the chunk slow tier address. Thus, there is no data being read from the fast tier and written to the slow tier. If the slow tier reference count in its KV-pair is 0 and the data chunk has already been garbage-collected in the slow tier according to its Status value, a copy

of the chunk is written to the slow tier and the chunk location is stored in its slow tier address in the KV-pair. Then, the *Chunk Address* in the recipe is updated with the slow tier address. In both cases, the chunk fast tier reference count decreases by 1, and its slow tier reference count increases by 1 in its KV-pair. If the MF is 1 (i.e., the chunk has been modified during its lifetime in the fast tier), the chunk will go through the same deduplication process as newly created files. Usually, one write will change multiple consecutive data chunks. Thus, the consecutively modified chunks will be deduplicated together to improve efficiency. New data chunks might be generated during the deduplication process of the modified chunks. Therefore, the whole file recipe will be rewritten with the updated metadata entries and newly generated metadata entries. After the data chunk is deduplicated, its RF and MF are both set to 0 in its metadata entry. Finally, the migration process of a reloaded file from the fast tier to the slow tier is completed, and the file is labeled as a deduplicated file. Files are exclusively locked from write requests during the deduplication process to maintain their consistency. To ensure data reliability and consistency, file recipe changes in memory are frequently synchronized with the copy in the fast tier (e.g., every 500ms). If a system failure happens before the most current synchronization, the file will roll back to the state of the last synchronization.

Each unique data chunk is represented by a KV-pair in a Key-Value Store (KVS), which is used as the deduplication indexing table. Each data chunk related operation will trigger reads or updates of the corresponding KV-pair. If the data chunk is garbage collected in both tiers, the corresponding KV-pair will be deleted. Since the KV-pairs are very frequently searched and updated, a high-performance flash-adaptive KVS is needed in TDDFS. The KVS design and implementation are not the focus of this research, and existing flash-adaptive KVS designs such as BloomStore [130], ChunkStash [94], and RocksDB [42] can be used in TDDFS as the indexing table. As the total amount of data in TDDFS increases, the indexing table itself can grow to a large size. If we use a SHA-1 hash as the chunk ID, one KV-pair is 44 bytes. If we use 8KB as the average chunk size, the KVS size is about 0.5% of the total stored data size. How to design and implement a high-performance and low-storage-overhead indexing table that can store a large amount of KV-pairs can be an interesting and important research work in the future. In our prototype, we use BloomStore for the KVS as discussed in detail in

Section 3.4.2.

3.3.4 Non-Redundant Chunk Reloading

When a deduplicated file is called by a file *open*, the request will trigger the whole file to be reloaded from the slow tier back to the fast tier. In most primary or secondary deduplication systems, the file restoring or reloading process assembles the whole file back to its original state (i.e., one consecutive file). Thus, it requires all the data chunks of the file to be read from the slow tier and written to the fast tier. If the reloaded file is later migrated to the slow tier again, the whole file will be deduplicated the same way as a newly created file even if it has not been modified or was only partially modified. By having no extra indexing structure to track the updates to the file and no maintaining data chunk information in the file recipe, previously obtained deduplication information cannot benefit the migration process.

In contrast, TDDFS applies non-redundant chunk reloading so reloaded files can maintain a similar kind of chunk-level sharing in the fast tier as deduplicated files do in the slow tier. During file reloading, only the relevant chunks that are not in the fast tier are read from the slow tier and written to the fast tier. The performance of the whole system can benefit from non-redundant chunk reloading. One benefit is that the amount of data to be migrated is reduced, since some data chunks that already exist in the fast tier do not need to be migrated. In addition, space is saved in the fast tier due to the data chunk sharing among reloaded files.

The process of migrating reloaded files to the fast tier is described by the following list of detailed steps. 1) The DRM reads the file recipe from the fast tier via the RFM. 2) The DRM examines the metadata entry of each chunk and finds the corresponding KV-pair in the indexing table. 3) If the fast tier reference count of the data chunk is 0, which means the chunk currently does not exist in the fast tier, TDDFS reads the chunk from the slow tier and then writes it to the fast tier with help from the RFM. After this, the fast tier chunk address in the metadata entry of the chunk is updated accordingly by the RFM. The DRM updates the fast tier reference count to 1 and reduces the slow tier reference count by 1 in the corresponding index table KV-pair entry. 4) If the chunk's fast tier reference count is greater than 0, which means the chunk already exists in the fast tier, its fast tier reference count in the KV-pair entry increases by 1, its slow tier

reference count decreases by 1, and the fast tier address in the metadata entry of the chunk is updated to its current fast tier address. 5) The RF and MF are set to 0 in its metadata entry. 6) Steps 2 to 5 are repeated until the whole file recipe is processed. Finally, the file state is changed from deduplicated file to reloaded file.

Usually, flash-based SSD technology is used for the fast tier, and this raises concerns about the wear-out problem. Non-redundant chunk reloading can reduce the amount of data written to the fast tier which can potentially expand the lifetime of the SSD storage. According to Leung *et al.* [114], if a file is accessed, it will probably be re-accessed frequently within a short period of time. Therefore, by reloading accessed deduplicated files from the slow tier to the fast tier, the file system's overall performance will be much better than if these files are kept in the slow tier without reloading. However, it may potentially be wasteful to reload all data of a large file to the fast tier if the file is only partially accessed. With additional information from the FSL or upper-layer applications, there could be a more efficient way to load a portion of the file to the fast tier. Since designing partial file reloading is not the main focus of TDDFS, we reload whole files from the slow tier to the fast tier when they are accessed to simplify the implementation.

3.3.5 Shared Chunk Management

Since some of the data chunks in the fast tier are shared among reloaded files, the file read and write operations should be carefully handled to ensure the data correctness, persistence, and consistency. TDDFS relies on the RFM to cooperate with the FSL to achieve the shared chunk management. Here, we describe how the read and write requests to a reloaded file are processed by TDDFS.

Several steps are needed to finish a write request. When a write request comes, according to the offset in the file and requested data size, the RFM locates the corresponding data chunks by checking the file recipe. First, for each chunk, the corresponding chunk KV-pair in the indexing table is examined by the DRM. If the chunk fast tier reference count is 1 and the chunk slow tier reference count is 0, which means the data chunk is uniquely used by this file, the original chunk in the fast tier can be directly updated. A new KV-pair for the updated chunk is created and the old KV-pair of this chunk is deleted in the indexing table if this chunk is garbage collected in both tiers. If

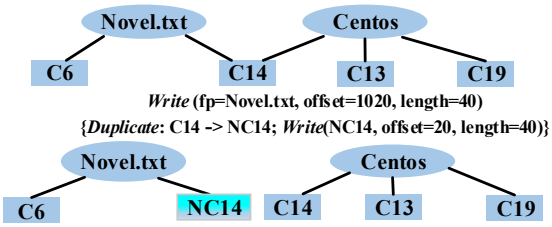


Figure 3.3: An example of reloaded file shared chunk updating

the chunk is shared by other reloaded files (fast tier reference count ¿ 1) or the chunk is used by some deduplicated files in the slow tier (slow tier reference count ≥ 1), TDDFS applies Copy on Write (CoW) to create a new data chunk with the updated content. The *Chunk Address* in the metadata entry points to the new data chunk and the MF is set to 1. At the same time, the fast tier reference count of the original chunk decreases by 1 in its KV-pair. Future updates to this new data chunk can be directly applied without changes to the metadata entry of the original chunk.

One example is shown in Figure 3.3. `Novel.txt` and `Centos` are reloaded files, and they share the same data chunk, C14. When a user updates the 40 bytes of `Novel.txt` from offset 1020, TDDFS calculates that C14 will be modified starting from offset 20 in that chunk. It reads C14 and updates its content in memory. Then, a new data chunk, NC14, is created, and the corresponding chunk metadata entry of the `Novel.txt` file recipe is updated to reference the new chunk. At the same time, the fast tier reference count of C14 decreases by 1 in its KV-pair.

For read requests, according to the read request offset in the file and the read size, the RFM first locates the metadata entries that correspond to the data chunks containing the requested data in the file recipe. Then, the requested data is read from these data chunks to the read buffer, and this buffer is returned to the application. The data chunks in the fast tier are not exclusively locked for read operations, and they can serve read requests from different files simultaneously.

Compared to the data of newly created files, the data of reloaded files is more fragmented. Therefore, the read performance of reloaded files is relatively lower than that of newly created files. Due to the CoW design for write requests, the write performance of reloaded files is also lower. However, according to the discussion of file access locality presented in Section 3.2.2, files tend to be repeatedly written by applications if they are

written once. Only the first write to a shared chunk has the high performance overhead, and the overhead can be amortized over the following writes to that chunk. Also, we usually use a flash-based drive as the fast tier, and its random access performance is not much lower than its sequential access performance. Thus, the average read and write performance of reloaded files are acceptable.

3.3.6 Major Data Operation Cases in TDDFS

In TDDFS, files fall into one of three categories: newly created file, deduplicated file, or reloaded file. TDDFS stores all the metadata in the fast tier, and most metadata operations are still processed the same way they would be processed in a regular file system. However, the processes used to respond to data requests are different because of the aforementioned four unique features of TDDFS. When handling a data request, TDDFS first checks the file category. Then, the request is responded to with different logic paths as described below.

Newly Created File Read and Write: If the file is a newly created file, the FSL directly reads or writes data in the fast tier. In order to maintain good performance on newly created files, there is no change in their data and metadata structures.

Reloaded File Read and Write: The details of the data read and write process for reloaded files are described in Section 3.3.5. The read and write performance of reloaded files is relatively lower than that of the newly created files due to the extra overhead of writing new chunks, file recipe updates, and KV-pair updates.

Deduplicated File Read and Write: The deduplicated files cannot be read from or written directly to the slow tier. When a file *open* happens on a deduplicated file, TDDFS starts a backend thread to reload the file back to the fast tier. This reloading process is described in Section 3.3.4. If read and write requests happen during the file reloading, TDDFS uses the following logic to response to the requests. For a read request, according to the read request offset in the file and the read size, the RFM first locates the metadata entries that correspond to the data chunks containing the requested data in the file recipe. Then, TDDFS reads the requested data from the corresponding data chunks. Note that these data chunks can be located in either the fast tier (the data chunks have been reloaded to the fast tier and its RF is 1) or the slow tier (the data chunk will be reloaded shortly and its RF is 1). Then, TDDFS returns

the requested data to the application.

For a write request, if the data chunks being written are already reloaded to the fast tier, updates are performed directly on the unique data chunks and CoW is performed on the shared chunks. The MFs of these chunks are set to 1 in the file recipe. If the data chunks being updated have not yet been reloaded to the fast tier (their RFs are 0), TDDFS first reads the corresponding data chunks from the slow tier to the memory. Then, these data chunks are updated in the memory and written to the fast tier as new data chunks. The RFs and MFs of these chunks in the file recipe are set to 1. The DRM will skip a chunk whose RF has already been set to 1 and continue to reload the next data chunk. To maintain data consistency, concurrent reads or writes are not allowed during file reloading. Applications experience degraded read and write performance during the file restoration. After the file is completely reloaded to the fast tier, the read and write performance will be the same as that of reloaded files.

Other Data Operations: TDDFS also supports other data operations including file deletion and file truncation. The FSL responds to these operations on newly created files directly. File truncation (e.g., the file truncate interface in Linux), which is used to cut the file, is treated as a file write (update) and handled by the write operation logic. File deletions of reloaded files and deduplicated files are handled by metadata operations including metadata deletion and KV-pair updates done by the FMM and RFM. An un-referenced chunk whose fast tier or slow tier reference count is 0 in its KV-pair is cleaned in the corresponding tier by the garbage collection process.

3.4 Implementation

In this section, we present the implementation details of TDDFS. The prototype consists of about 12K LoC (Lines of Code) and can be executed on top of most Linux systems. Rather than implementing TDDFS in kernel space, we developed the TDDFS prototype based on FUSE [131]. FUSE is a user space file system with a standard POSIX interface. Although the FUSE-based file systems do not have the same performance as kernel-based file systems, it is used for simplicity and flexibility as a proof of concept. In the following subsections, we elaborate on the implementation details of the data management, indexing table, and file deduplication.

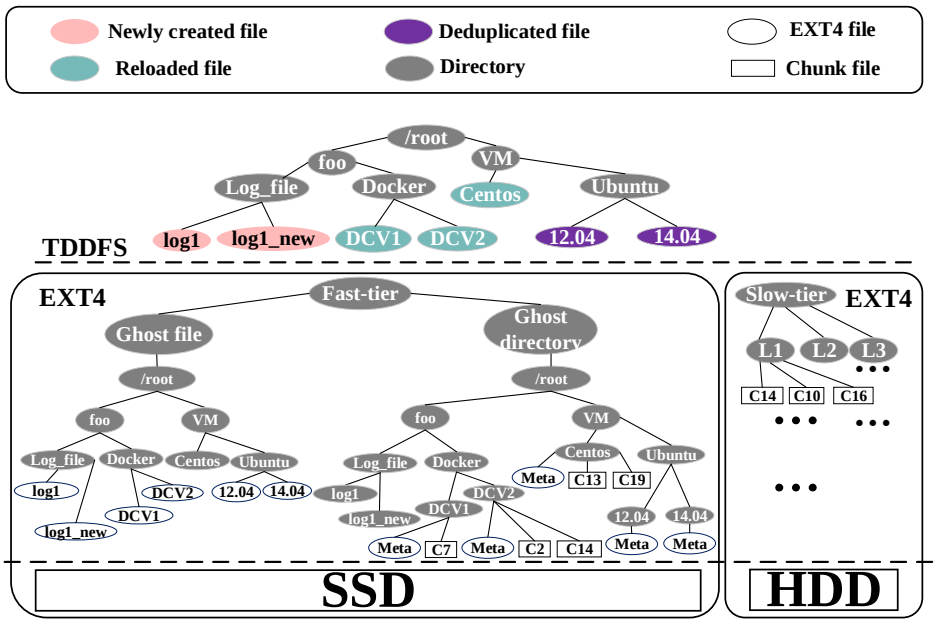


Figure 3.4: Data Layout of TDDFS Prototype

3.4.1 Data Management

In TDDFS, we use the underlying file system to store the data passed through by FUSE. TDDFS creates two top-level directories for the fast tier and the slow tier, respectively, as shown in Figure 3.4. One directory, named Fast-tier, is located on the underlying file system (e.g., Ext4) formatted SSD. The Slow-tier directory is on the underlying file system formatted HDD and is used to store data chunks of deduplicated files. The data chunks generated by deduplication are stored as files under the Slow-tier directory with unique names. One directory under Slow-tier directory acts as one chunk store, and 10,000 data chunks are stored in the same chunk store. As shown on the right side of Figure 3.4, directory L1 under the Slow-tier is one of the chunk stores, and data chunks are stored under this directory. In some data deduplication designs [36, 55], a large number of chunks (e.g., more than one thousand data chunks) are packed in one container to speed up the chunk write and restore performance. Container-based chunk storing can be easily adopted in TDDFS, and it can be combined with other restore performance improvement schemes [55, 103, 59] to further increase the file deduplication and reloading performance.

Referring to the left side of Figure 3.4, the shape of the directory tree from an application's view through FUSE is mimicked twice under the Fast-tier directory in the underlying Ext4 file system. TDDFS creates one Fast-tier subdirectory called *Ghost file* and one Fast-tier subdirectory called *Ghost directory*. These two directories contain ghost files and ghost directories, respectively. The file and directory structure under *Ghost file* matches the application's view including directories, newly created files, deduplicated files, and reloaded files. *Ghost directory* is similar but has a directory entry for each file in the application's view. If the application view file is a newly created file, its data is stored in its matching ghost file and its ghost directory is empty. During the migration process, the data in the ghost file of a newly created file is read, deduplicated, migrated (if there are unique chunks), and deleted. The empty ghost file remains, and its file metadata information (e.g., *file_stat*), file recipe, and other information used for reloading are maintained as files under its ghost directory. When reloading the deduplicated files back to the fast tier, the unique data chunks are written to the fast tier under the ghost directory as files. We assume that the underlying file system guarantees the reliability, persistence, and consistency. TDDFS takes care of the failures that happen before the data is written to the underlying file system. When migrating the files from the fast tier to the slow tier, the least recently used files (including the files whose size is smaller than 16KB) are selected to be migrated to the slow tier.

To expand upon the above description, the same example that is used in Figure 3.2 is also presented here in Figure 3.4. The same directory structures are created under both *Ghost file* and *Ghost directory*. Since `log1` and `log1_new` are newly created files, their data is stored in their ghost files under *Ghost file* with the same file names, and their ghost directories are empty. `Ubuntu/12.04` and `Ubuntu/14.04` are deduplicated files, and their data is stored under Slow-tier as files of unique data chunks. `DCV1`, `DCV2`, and `Centos` are reloaded files, and the unique data chunks like C2, C7, and C14 are stored under the corresponding ghost directories of these files. The file recipe of a reloaded file or a deduplicated file is stored as a file named *Meta* under its ghost directory. A unique data chunk stored in the fast tier or the slow tier can be found using its path and name. In this prototype, we assume the fast tier's space is large enough, and we do not apply the file recipe migration design in order to simplify the implementation.

3.4.2 Indexing Table

We follow the design of BloomStore [130], which is flash-adaptive, to implement the deduplication indexing table. In BloomStore, the whole key space is hashed into several instances. When inserting a KV-pair, the new KV-pair is hashed to one of the instances according to its key, and the KV-pair is logged to the KV-pair write buffer of this instance. This KV-pair write buffer is the same size as a flash page. When the buffer is full, it will be flushed to the fast tier as an immutable KV-block with a timestamp. Each KV-block has one Bloom Filter (BF) to speed up the KV-pair lookup process. Since the flash-based SSD cannot be overwritten before the whole block is cleaned [19], BloomStore does not change the KV-blocks directly. Updates and deletions are achieved by inserting new KV-pairs with the same key into the BloomStore. To delete a KV-pair, a new KV-pair with the same key and a value of *null* is inserted.

When searching for a key in BloomStore, the key is hashed to one corresponding instance. The BFs of this instance are checked first. The KV-blocks whose BFs have positive search indications are fetched to the cache and searched from the newest one (the KV-pair write buffer) to the oldest one according to the sequence of their timestamps. The first matching KV-pair with a valid value is returned. If the key cannot be found in all of the related KV-blocks or the first KV-pair has a *null* value, a negative value is returned. The invalid KV-pairs (the KV-pairs which are updated or deleted) will be cleaned during the garbage collection process. More implementation details can be found in [130]. In the current implementation, the KVS is locked when it is processing one request in order to ensure data consistency, which will have a performance penalty. TDDFS stores the KV-blocks in the fast tier as files in the underlying file system.

3.4.3 File Deduplication

When deduplicating newly created files and modified data chunks of reloaded files, TDDFS reads the file data into the chunking buffer in memory and uses Adler32 to calculate the 32-bit checksum of data in the CDC sliding window. If the checksum matches the pre-determined chunking mask value, a new cutting point is decided. While we used SHA-1 as our deduplication fingerprint algorithm, it is no longer considered to be secure and practical hash collisions have been demonstrated [132]. Therefore, a

practical implementation of TDDFS would have to change to a stronger hash function. When we search for a chunk ID in the indexing table, if the chunk ID already exists in the indexing table, a metadata entry will be created and appended to the file recipe to index the data chunk. If the chunk ID is new, the new data chunk is stored in the slow tier. Then, a new KV-pair will be created and stored in the KVS to index the new data chunk, and a metadata entry will also be appended to the file recipe.

3.5 Experimental Evaluation

In this section, we present the results of the performance evaluation. First, we introduce our evaluation methodology, baselines that are compared with TDDFS, and the workloads we use in the evaluation. Then, the performance results and comparisons are discussed. Finally, the cost-effectiveness is analyzed.

3.5.1 Experimental Setup and Workloads

To evaluate TDDFS performance, we deployed our prototype on a Dell R420 server with a 2.2GHz Intel Xeon 4-core processor and 12GB of memory. A Samsung 850 PRO 512GB with a SATA interface is the SSD used as the fast tier. The slow tier is a Western Digital WD RE4 WD5003ABYX 500GB 7200 RPM HDD with 3.0Gb/s SATA connecting to the motherboard. The operating system is Ubuntu 14.04 with a 3.13.0 kernel, and the underlying file system on both tiers is Ext4.

We implemented four different designs based on FUSE to compare with TDDFS. *NoDedup* manages the fast tier and the slow tier at the file system level, and there is no deduplication performed on either tier. Files are migrated between the two tiers according to the LRU policy. *AllDedup* is the design where the fast tier and the slow tier are managed at the file system level, and all files are deduplicated before these files are stored in either tier. When a file is read, the requested data chunks are located and returned. In this design, data chunks are migrated between the two tiers according to the LRU policy. *FastTierNoDedup* is a tradeoff design between *NoDedup* and *AllDedup*. Files in the fast tier are in their original states. When a file is selected to be migrated to the slow tier, it is deduplicated with the same implementation as TDDFS. If a file in the slow tier is accessed, it will be restored to its original state and stored in the

fast tier. *FastTierNoDedup* uses the design concept of the Dell SC Series hybrid storage solution with deduplication [133]. In *SlowTierOnly*, all files are stored in the slow tier to show the slow tier only performance in comparison to other tiering designs.

To measure and compare the performance of TDDFS, we use throughput, latency, and the amount of data migrated as the metrics. Throughput is the average number of requests completed in a time unit. It is calculated as the sum of the data sizes of all requests completed divided by the sum of total completion times of all file requests. Latency is the average request response time, measured at the file system level, from when the request is submitted to its completion. The amount of data migrated is the total amount of data that migrates from the fast tier to the slow tier or from the slow tier to the fast tier when the whole trace is replayed. During the evaluation, we vary the value of the fast tier ratio, the deduplication ratio, and the read ratio to measure and compare the performance of various designs. The fast tier ratio is the percentage of the whole data set that can be stored in the fast tier. If the fast tier ratio is 100%, it means all the data can be stored in the fast tier. If the fast tier ratio is 25%, it means 25% of all the file data is stored in the fast tier and 75% of all the file data is stored in the slow tier. To show the capacity ratio between the fast tier and the slow tier (i.e., how many times larger the slow tier size is than the fast tier size), we increase the slow tier size from 0 times, to 1 times, to N times the fast tier size. Thus, the fast tier ratio is represented from 100%, to 50%, to $\frac{100}{1+N}$%. For each trace, the total amount of data of all files is different. It is hard to compare the performance of different traces by using the same fast tier space size. Therefore, we define another threshold called the actual fast tier usage limit, which is calculated as the fast tier ratio times the total trace size. During the trace replay, when the total data size in the fast tier is larger than the actual fast tier usage limit, migration will be triggered. Note that the real-world capacity of the fast tier (SSD) may not be full when the amount of data in the fast tier reaches the experiment's actual fast tier usage limit. In our tests, when we refer to a fast tier ratio, we really use the calculated actual fast tier usage limit to replay the trace. The deduplication ratio is the original total data size divided by the total data size after the deduplication. The read ratio is the percentage of read requests among the all the data I/O requests (i.e., among the combined total number of read requests and write requests).

Table 3.2: Details of the workload traces

	# files	# requests	Size(GB)	Read ratio
Lab_trace	2,400	39,000	22	10%
Var_trace	10,000	32,000	5	45%
Syn_trace	5,000	100,000	100	N/A

The performance of TDDFS is closely related to the file access patterns, file size, read/write ratio, and the deduplication ratio. Most open-source file traces do not provide the file size and deduplication ratio information. File system benchmarks generate files with no meaningful content, so the deduplication ratio is still unknown. We use three different workloads to evaluate the performance of TDDFS, and the details of these three traces are shown in Table 3.2. Lab_trace is a file access trace collected from a file server in our lab for one week on top of *AllDedup*. We collected the size of each file and the file requests. The overall deduplication ratio of Lab_trace is 4.1. Var_trace is collected when we run the Filebench [134] workload *varmail* with the default Filebench configuration. Again, we collected the size of each file and the file requests. To comprehensively examine the relationship between performance and the deduplication ratio, read ratio, and fast tier ratio, we synthesize a set of file access traces, called Syn_trace, which has 100GB of files and 100,000 access requests. The file size and access patterns follow the Zipfian distribution (20% of the files have 80% of the total accesses and occupy 80% of the total size), which aligns with the observations in [124]. Different deduplication ratios, read ratios, and fast tier size ratios of Syn_trace are evaluated during the experiments. This means we generate a set of Syn_traces with the same file size and access distribution but different read request ratios. When the read ratio is selected, the number of read and write requests on each file is decided, and these requests are distributed to the duration of the whole trace according to the Zipfian distribution (80% of the requests happen after the first request to this file within 20% of the trace duration). The Syn_trace generated in this way with the Zipfian distribution has good locality, which can substantially benefit the tiered designs. Therefore, the evaluation of Syn_trace shows how TDDFS will perform when locality is good. In each test, the trace replays five times and we show the average value. Since the benchmark-generated Var_trace and the synthetic Syn_trace do not have meaningful time stamps, we replay the records of the three traces one after another (i.e., when the current request is finished

and returned, the next request is issued).

A trace provides both file information and access information. To replay the traces, we make some changes to the prototype implementation such that the files are accessed according to the trace content. Before we replay the trace in each test, all the files are written to the prototype system in the same order as they appear in the trace with the collected size and no meaningful content. The earliest created files will be migrated to the slow tier if the fast tier space usage hits the actual fast tier usage limit. Since the trace does not have the fingerprint of the file content, we can only emulate the deduplication process by making small changes in the FMM, DRM, and RFM. During deduplication, the byte stream is still processed by the chunking and fingerprint generating steps. However, the real cutting points are randomly generated between a *minimal chunk size* (512B) and a *maximal chunk size* (16KB). Suppose the deduplication ratio is R_{dr} in this test. Then $(\frac{100}{R_{dr}})\%$ of the chunks of this file are randomly selected as unique chunks and the remaining $(100 - \frac{100}{R_{dr}})\%$ of the file's chunks are duplicate chunks. The fingerprint of a unique chunk is assigned as a unique increasing integer, U_i, which increments from 0. The fingerprints of duplicate chunks are randomly selected between 0 and the largest U_i generated to show that they are duplicates. The data of a unique chunk is stored in the slow tier as a file. After we have the fingerprint of the chunk, the corresponding KV-pair is generated and inserted into the KVS or the existing KV-pair is fetched from the KVS and updated. Then, the metadata entry with the corresponding content is written to the file recipe. In our experiment, we did not migrate the recipes to the slow tier if their size is very large. However, in a production scenario, especially when the accumulated files are very large, migrating the recipes to the slow tier is recommended.

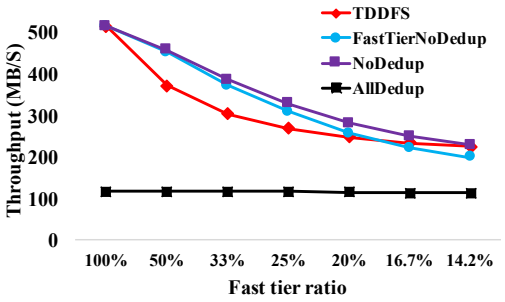
The same emulation process is also used during the trace replay when reloading deduplicated files and updating reloaded files. For example, if the deduplication ratio is R_{dr}, when reloading a file from the slow tier to the fast tier, $(\frac{100}{R_{dr}})\%$ of the chunks of this file are read and written to the fast tier as unique data chunks. We assume that all reloaded files have the same deduplication ratio. Therefore, $(\frac{100}{R_{dr}})\%$ of the chunks are treated as new chunks that are not already in the fast tier. When writing the reloaded file, there is a $(100 - \frac{100}{R_{dr}})\%$ chance that the chunk will be CoW since there will be $(100 - \frac{100}{R_{dr}})\%$ of the chunks of this file that are shared chunks. Although the emulated

deduplication process is slightly different from deduplicating the files with real content, it has similar latency, throughput, and resource consumption to real deduplication. Therefore, we can use it to emulate the migration process with different deduplication ratios, and it can show the performance, cost-effectiveness improvement, and tradeoffs of TDDFS. In the real-world workloads, when more and more data chunks are accumulated in the slow tier, the restore performance penalty caused by the data chunk fragmentation will lower the overall throughput.

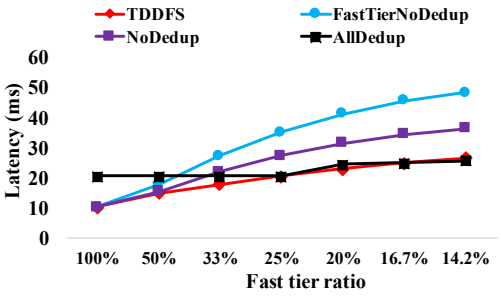
3.5.2 Throughput and Latency

When the deduplication ratio and read ratio are high, TDDFS can deliver better performance according to our tests. In Lab_trace, as shown in Figure 3.5(a), the throughput of TDDFS is slightly lower than that of *FastTierNoDedup* and *NoDedup*. As the fast tier ratio decreases from 100% to 20%, more files are migrated and stored in the slow tier. Thus, the throughput decreases and latency increases in all four tiering designs. In Lab_trace, most of the requests (90%) are write requests due to the heavy uploading and updating of files in the file server. High write ratio causes lower average throughput in TDDFS, which is caused by the CoW and recipe updates of the reloaded files. However, compared with the other three tiering designs when the fast tier ratio is below 33%, as shown in Figures 3.5(a) and 3.5(b), the rate of throughput decrease and rate of latency increase of TDDFS is better since TDDFS can store more files in the fast tier than the other designs. By storing more files in the fast tier, the performance degradation of TDDFS is alleviated. If read requests dominate the file operations, TDDFS can achieve better performance. In Syn_trace, we configure the read ratio as 80% and the deduplication ratio as 3. As shown in Figures 3.5(c) and 3.5(d), when the fast tier ratio is higher than 33%, the throughput of TDDFS is very close to that of *FastTierNoDedup* and *NoDedup*. When the fast tier ratio is lower than 33%, the throughput of TDDFS is the best among the four designs. Again, this is because more reloaded files can be stored in the fast tier, and most of the file accesses are read requests. Note that in a typical tiered storage system, the fast tier ratio is low.

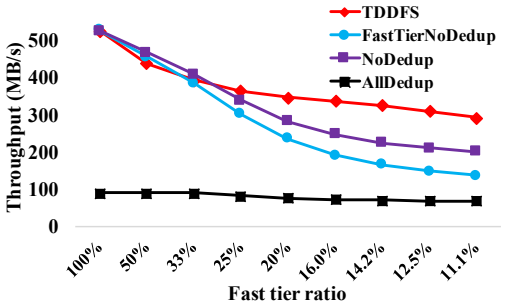
The performance of TDDFS is closely related to the deduplication ratio of the files. Since the fast tier ratio of 25% is the performance turning point in Lab_trace and Syn_trace, we fix the fast tier ratio to 25% in Var_trace to evaluate the performance with



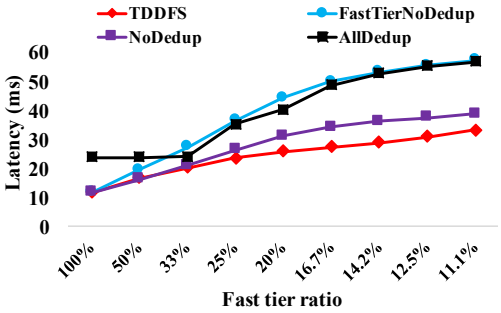
(a) Throughput of Lab_trace



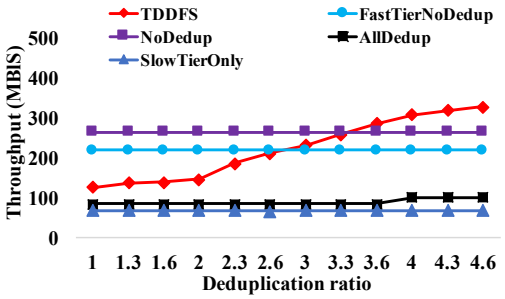
(b) Latency of Lab_trace



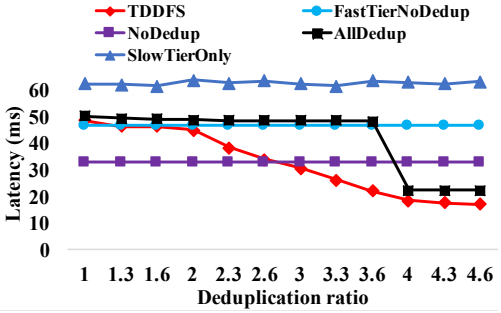
(c) Throughput of Syn_trace



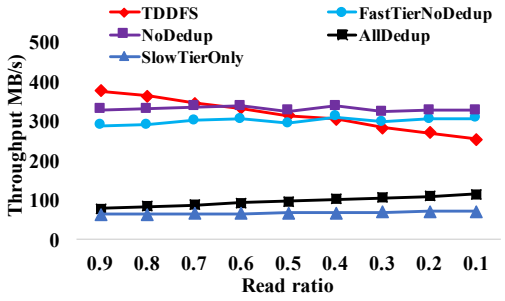
(d) Latency of Syn_trace



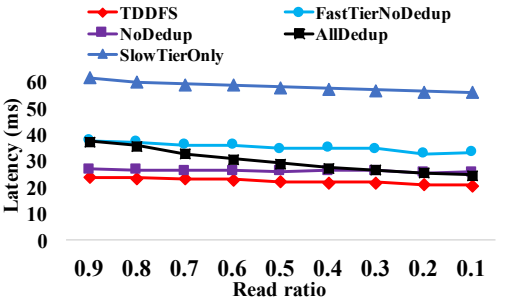
(e) Throughput of Var_trace



(f) Latency of Var_trace



(g) Throughput of Syn_trace



(h) Latency of Syn_trace

Figure 3.5: Performance comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup*, and *SlowTierOnly*

different deduplication ratios. As shown in Figures 3.5(e) and 3.5(f), the throughput and latency of *FastTierNoDedup* and *NoDedup* are nearly the same when the deduplication ratio varies from 1 to 4.6. Since *FastTierNoDedup* and *NoDedup* store the original files in the fast tier, the only difference is that *FastTierNoDedup* deduplicates files in the slow tier and restores files from the slow tier to the fast tier. Thus, deduplication ratio has little impact on the overall performance of *FastTierNoDedup* and *NoDedup*. Compared to the relatively flat throughput and latency of those two designs, *AllDedup* has a large change when the deduplication ratio is about 4. If the deduplication ratio is 4 or larger, *AllDedup* can store all the files in the fast tier since the fast tier ratio is configured at 25% in this test. Therefore, its throughput increases and the latency has a large reduction. In contrast to these three designs, the throughput of TDDFS increases and the latency decreases gradually as the deduplication ratio increases from 1 to 4.6. As the deduplication ratio increases, more reloaded files can be stored in the fast tier with the help of non-redundant chunk reloading and unique chunk sharing as described in Section 3.3. Thus, the performance degradation caused by the data migration is alleviated. When the deduplication ratio is higher than 3.3, the throughput and latency of TDDFS are the best among the four designs. *SlowTierOnly* shows that if we only use the slow tier, its overall performance will be the worst. Its throughput can be more than 50% lower than that of TDDFS, and its latency can be 100% higher than that of TDDFS. In general, the performance of TDDFS is closely related to the deduplication ratio of files, and TDDFS performs better when the deduplication ratio is higher.

Since the read ratio is fixed in Lab_trace and Var_trace, we synthesize a set of Syn_traces with the same files, deduplication ratio, and access patterns but different read ratios. The access distribution we use is Zipfian, so most of the operations are on a small portion of the files. The results are shown in Figures 3.5(g) and 3.5(h). As the read ratio decreases from 90% to 10%, the performance of *FastTierNoDedup* and *NoDedup* does not change much since files are accessed when they are in their regular format in the fast tier. The throughput of TDDFS decreases from 370MB/s to about 250MB/s as the read ratio decreases. Due to the low performance of the write operation on the reloaded files, a lower read ratio (higher write ratio) will cause lower throughput in TDDFS. When the deduplication ratio is 3, the fast tier ratio is 25%, and the read ratio is higher than 70%, the throughput of TDDFS is higher than the others.

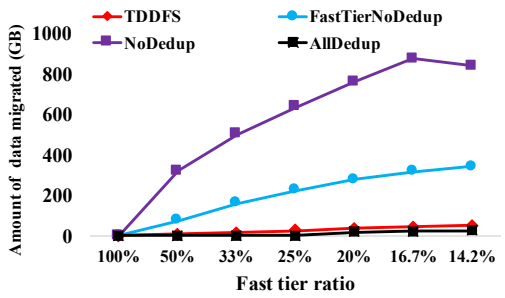
According to Roselli *et al.*[123], the number of read requests in a typical file system is usually several times higher than the number of write requests (e.g., the read ratio is 80% or even higher). Thus, TDDFS can deliver good performance in a real production environment.

Also, as the read ratio decreases, the latency of TDDFS, *FastTierNoDedup*, and *NoDedup* does not change much since most requests are performed in the fast tier. Due to the deduplicated nature of reloaded files, more hot files can be stored in the fast tier by TDDFS than in the others. Thus, its average latency is lower than that of the others. Since *AllDedup* performs deduplication to all the files, the data being written needs to be deduplicated before the data is stored. Also, each read needs to assemble the requested chunks before returning the data to the application. Thus, its performance is lower than that of other tiered designs. In our implementation of *AllDedup*, which does not have any restore optimization, the throughput of read is lower than write and also the latency of read is much higher than write. Thus, as the read ratio decreases, the throughput of *AllDedup* increases slightly and its latency decreases. Clearly, without the help of the fast tier, *SlowTierOnly* shows the worst performance among all the designs.

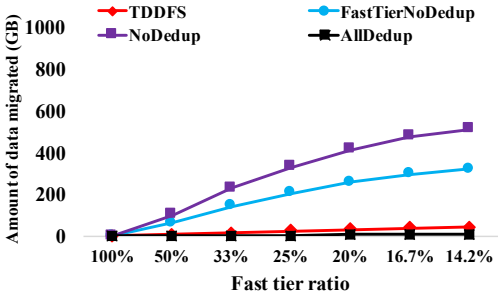
In general, according to our evaluation, the overall performance of TDDFS is better than that of *AllDedup*. In most cases, the latency of TDDFS is lower than the others because more reloaded files are stored in the fast tier. With a higher deduplication ratio, higher read ratio, and lower fast tier ratio, TDDFS performs better, and in some cases TDDFS performs the best, as shown in the evaluations.

3.5.3 Data Migration between the Fast tier and the Slow tier

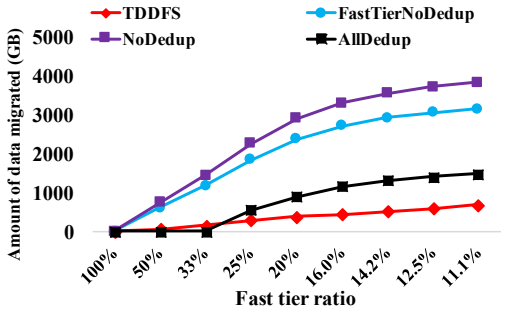
The data migration cost is measured as the total amount of data migrated from the fast tier to the slow tier or from the slow tier to the fast tier after executing the whole trace. The results are shown in Figure 3.6. In the four designs (excluding the *SlowTierOnly* since it does not have any data migration), *NoDedup* always has the largest data migration volume since all the files are in their original states. Also, since fewer files can be stored in the fast tier without deduplication, file migration happens more frequently in *NoDedup*. The migration cost of *AllDedup* is the lowest since all the files are deduplicated, and only the unique chunks are migrated between the two tiers. Also, *AllDedup*



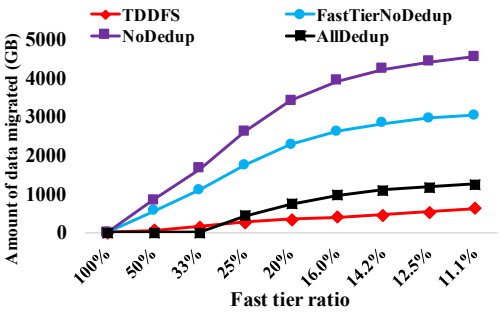
(a) Data migrated from the fast tier to the slow tier in Lab_trace



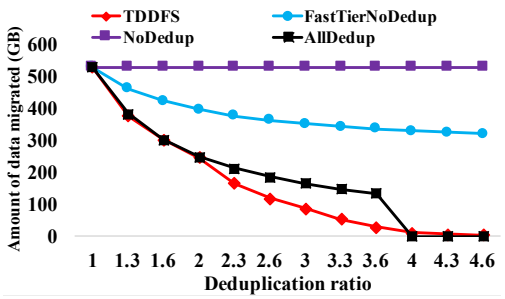
(b) Data migrated from the slow tier to the fast tier in Lab_trace



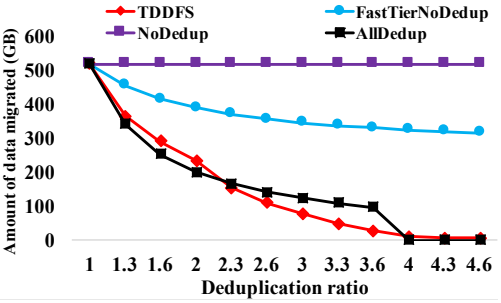
(c) Data migrated from the fast tier to the slow tier in Syn_trace



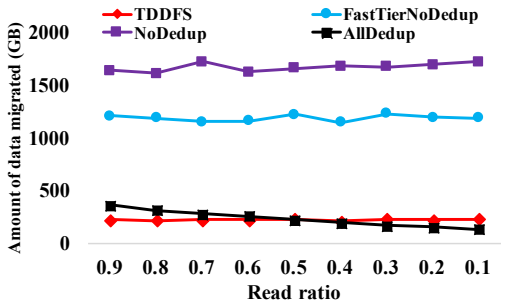
(d) Data migrated from the slow tier to the fast tier in Syn_trace



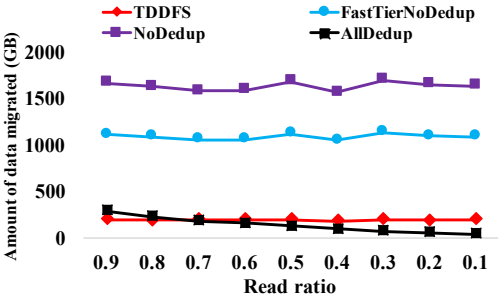
(e) Data migrated from the fast tier to the slow tier in Var_trace



(f) Data migrated from the slow tier to the fast tier in Var_trace



(g) Data migrated from the fast tier to the slow tier in Syn_trace



(h) Data migrated from the slow tier to the fast tier in Syn_trace

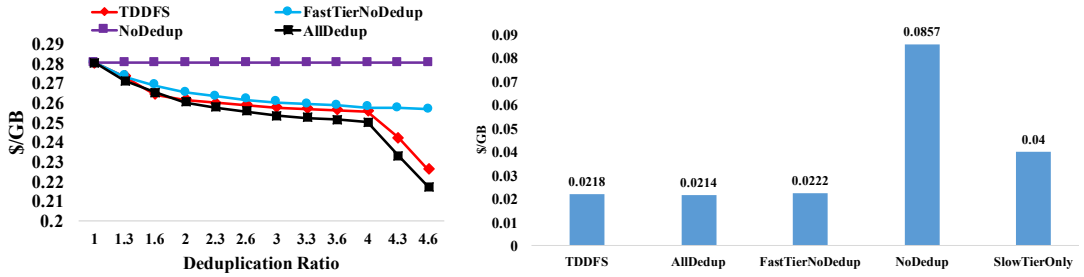
Figure 3.6: Migration cost comparison of TDDFS, *NoDedup*, *AllDedup*, and *FastTierNoDedup*

can hold more files in the fast tier than the others, and migrations happen less frequently. *FastTierNoDedup* performs better than *NoDedup*, but its total migrated data size is still much higher than that of *AllDedup* and TDDFS. *FastTierNoDedup* reduces the amount of data that is written to and read from the slow tier via slow tier deduplication. However, the amount of data being migrated from or written to the fast tier is the same as that of *NoDedup*. Thus, the migration cost of *FastTierNoDedup* is much higher than that of TDDFS and *AllDedup*.

As shown in Figure 3.6, the total amount of data migrated between the fast tier and the slow tier of TDDFS is very close to that of *AllDedup*. Although TDDFS needs to read all the data of a newly created file when it is migrated to the slow tier, TDDFS avoids the re-deduplication and migration if the data chunks of a reloaded file are not changed during its life cycle in the fast tier. Thus, the total migration data volume is very close to that of *AllDedup*. Inevitably, as the fast tier ratio decreases, the fast tier will store fewer files and data migration will occur more frequently. The rate at which the total amount of migrated data of TDDFS increases as the fast tier ratio decreases is the lowest among the four designs as shown in Figures 3.6(a), 3.6(b), 3.6(c) and 3.6(d). With higher deduplication ratios, the migration cost of TDDFS becomes lower as shown in Figures 3.6(e) and 3.6(f). The read ratio does not show a major influence on the TDDFS migration amount as seen in Figure 3.6(g) and 3.6(h). On average, our evaluations show the total amount of data being migrated between the fast tier and the slow tier of TDDFS is about 10% of *NoDedup*, 20% of *FastTierNoDedup*, and close to *AllDedup* in most cases. In general, the migration cost of TDDFS is much lower than that of *FastTierNoDedup* and *NoDedup* and is close to that of *AllDedup*. Lower migration cost can reduce the storage bandwidth used by migration and alleviate the performance impact on the primary workloads during the migration. It can also reduce wear on flash-based SSDs.

3.5.4 Cost-Effectiveness Analysis

We use the average $/GB price to compare the cost-effectiveness of the four designs. We use the final utilized space in the fast tier and the slow tier after replaying the trace to calculate the average $/GB. Here, we use the $/GB of an Intel 750 Series 400GB SSD (SSDPEDMW400G4R5) and a Seagate 6TB enterprise HDD (ST6000NM0024) as



(a) $/GB comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup* with different deduplication ratios

(b) $/GB comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup* and *SlowTierOnly* when the size of slow tier is 20 times larger than fast tier

Figure 3.7: Migration cost comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup* and *SlowTierOnly*

the base to analyze the cost-effectiveness. For these two devices, the $/GB of SSD is about 1 [135] and $/GB of HDD is about 0.04 [136].

We use Var_trace to analyze the $/GB variation as the deduplication ratio increases, and we assume that the fast tier size is 25% of the total data size. As shown in Figure 3.7(a), the $/GB of *NoDedup* remains the same as the deduplication ratio increases because no deduplication is used. The $/GB of *AllDedup*, TDDFS, and *FastTierNoDedup* decreases as the deduplication ratio increases since data deduplication reduces the amount of data being stored in the fast tier and the slow tier. When the deduplication ratio is larger than 4, the $/GB of *FastTierNoDedup* keeps a similar saving ratio while the $/GB of *AllDedup* and TDDFS has a significant decrease. When the deduplication ratio is high enough, *AllDedup* and TDDFS are able to save space in the fast tier which causes a significant reduction in $/GB. In this calculation, when the deduplication ratio is higher than 4.3, TDDFS no longer needs any slow tier space and all files are stored in the fast tier. As the deduplication ratio continuously increases, TDDFS can save more space on the fast tier and the rate of $/GB decrease is nearly the same as that of *AllDedup*. In general, TDDFS can achieve better cost-effectiveness than that of *NoDedup* and *FastTierNoDedup*. Its cost-effectiveness is close to that of *AllDedup*. With a high deduplication ratio, TDDFS can achieve nearly the same cost-effectiveness as that of *AllDedup*.

In a tiered storage system, the size of the fast tier is usually much smaller than that of the slow tier. We let the size of the slow tier be 20 times larger than that

of the fast tier and compare the $/GB of each design when the deduplication ratio is fixed at 4. As shown in Figure 3.7(b), the $/GB of *NoDedup* is about 4 times that of TDDFS due to no deduplication. The $/GB of *SlowTierOnly* is also much higher than that of TDDFS, *AllDedup*, and *FastTierNoDedup*. It shows that the slow tier only system without deduplication cannot provide either good performance or good cost-effectiveness. The $/GB of TDDFS, *AllDedup*, and *FastTierNoDedup* is about 0.02 and very close to each other. This means the cost-effectiveness of TDDFS, *AllDedup*, and *FastTierNoDedup* is very close. Importantly, according to the performance evaluation in Section 3.5.2, TDDFS usually has better performance than that of *AllDedup* and *FastTierNoDedup*. Therefore, TDDFS simultaneously achieves good tradeoffs in terms of performance and cost-effectiveness.

3.6 Related Work

Some previous studies tried to address the challenges of primary data deduplication in different ways [92, 96, 95, 137]. Most of them sacrifice the deduplication ratio by applying fixed-size chunking and make tradeoffs to improve the throughput. iDedup [92] sacrifices deduplication ratio to ensure higher throughput and lower latency. It may choose not to deduplicate non-sequential duplicate blocks to maintain locality so the deduplication and restore performance can be improved. Dmdedup [96] is implemented in a device mapper that can be used by most existing file systems and applications. Although it did place deduplication structures like the fingerprint index in a fast device and the deduplicated data blocks on a slower device, it does not have the concept of tiered storage and data cannot be migrated according to the data activity.

Instead of using the SSD and HDD as a tiered design, some studies use the SSD as a cache for primary storage and apply deduplication to the SSD to improve the system performance, SSD endurance, and space utilization. Nitro [138] combines deduplication and compression to improve the SSD cache capacity. By carefully designing the indexing structure and using Write-Evict Units that are the same size as the flash erase block, Nitro makes a good tradeoff between the SSD endurance and the performance of a primary storage system. CacheDedup [93] also introduces deduplication to the flash-based cache to improve the space utilization and flash lifespan. Uniquely, CacheDedup

uses separate a Data Cache (for data blocks in the storage) and Metadata Cache (for fingerprints and other metadata information about the data blocks in the storage). This is done to reduce the space used by metadata and to better collect the historical metadata information for duplication reorganization. Moreover, CacheDedup proposes the D-LRU and D-ARC cache replacement algorithms, which are duplication-aware, to further improve the cache performance and endurance. Similarly, TDDFS maintains the deduplicated state of reloaded files to achieve better capacity utilization of the fast tier. However, TDDFS keeps some of the new and hot files in their original state without deduplication to gain the best performance for these files.

Other primary deduplication works [139, 140, 111, 98] are implemented at the file system level. There are three file systems with a deduplication function: SDFS [140], Lessfs [139], and ZFS [111]. SDFS and Lessfs are implemented in user space and based on FUSE. SDFS [140] combines fixed-size and variable chunking to eliminate chunk duplicates among files, especially for virtual machines, but SDFS cannot manage two storage tiers. Lessfs uses fixed-size chunking, and it is implemented in C. Thus, its overhead might be lower than that of SDFS, and it can deliver higher throughput. However, its deduplication ratio is worse than that of TDDFS. Different from SDFS and Lessfs, ZFS does not support deduplication in its basic functions. When users enable deduplication in one directory, all data written in the directory will be deduplicated to gain better space utilization. However, its performance is limited by the memory space for deduplication metadata [96]. Although these works apply deduplication at the file system level to improve space utilization, they do not distinguish active and less-active files to selectively deduplicate them, and they do not operate in two-tier storage systems.

3.7 Conclusions and Future Work

It is very challenging to develop high-performance but low-cost file systems on primary storage. Previously, researchers focused on block-based tiered storage designs and primary deduplication to achieve this goal. However, problems still exist in these solutions such as migration efficiency, deduplication penalty, and isolation between the block-level management and the upper-layer applications.

To conquer these challenges, we design TDDFS to manage a two-tier storage system

and achieve efficient file-level migration. Firstly, we reduce the storage I/O bandwidth required between tiers during migration by eliminating duplicate chunks and enabling data chunk sharing among reloaded files. Secondly, in order to solve deduplication latency and throughput issues, we selectively deduplicate less-active files. In this way, we can minimize the performance impact on the whole system. Finally, combining deduplication with data chunk sharing in both tiers further improves the space utilization and reduces costs. In our evaluation, TDDFS achieves high performance and low $/GB at the same time. In the future, we plan to further improve the deduplication ratio and minimize the performance impact by designing a more intelligent file selection policy, applying deduplicated file pre-reloading, implementing a high-performance indexing table, and using adaptive chunking algorithms

Chapter 4

ALACC: Accelerating Restore Performance of Data Deduplication Systems Using Adaptive Look-Ahead Window Assisted Chunk Caching

4.1 Introduction

Coming into the second decade of the twenty-first century, social media, cloud computing, big data, and other emerging applications are generating an extremely huge amount of data daily. Data deduplication is thus widely used in both primary and secondary storage systems to eliminate the duplicated data at chunk-level or file-level. In chunk-level data deduplication systems, the original data stream is segmented into data chunks and the duplicated data chunks are eliminated (not to store). The original data stream is then replaced by an ordered list of references, called recipe, to the unique data chunks. A unique chunk is a new data chunk which has not appeared before. At the same time, only these unique data chunks are stored in the persistent storage. To maximize the I/O efficiency, instead of storing each single data chunk separately, these

unique chunks are packed into containers based on the order of their appearances in the original data stream. A container which may consist of hundreds or thousands of data chunks is the basic unit of data read from or written to storage with a typical size of 4MB or larger.

Restoring the original data is the reverse process of deduplication. Data chunks are accessed based on their indexing order in the recipe. The recipe includes the metadata information of each data chunk (e.g., chunk ID, chunk size, container address and offset). The corresponding data chunks are assembled in a memory buffer. Once the buffer is full, it will be sent back to the requesting client such that one buffer size data is restored. Requesting a unique or duplicate data chunk may trigger a container read if the data chunk is not currently available in memory, which causes a storage I/O and impacts restore performance. Our focus is specifically on restore performance in secondary storage systems.

In order to reduce the number of container-reads, we may read ahead the recipe and allocate the data chunks to the buffers in the Forward Assembly Area (FAA). We can also cache the read-out container (container-based caching) or a subset of data chunks (chunk-based caching) for future use. If a requested data chunk is not currently available in memory, it will trigger a container-read. It is also possible that only a few data chunks in the read-out container can be used in the current FAA. Therefore, to restore one container size of the original data stream, several more containers may have to be read from the storage causing read amplification. Read amplification causes low throughput and long completion time for the restore process. Therefore, the major goal of improving the restore performance is to reduce the number of container-reads [55]. For a given data stream, if its deduplication ratio is higher, its read amplification is potentially more severe.

There are several studies that address the restore performance issues [55, 100, 101, 102, 103, 104, 141]. The container-based caching scheme is used in [101, 102, 103, 141]. To use the memory space more efficient, a chunk-based LRU caching is applied in [55, 104]. In the restore, the sequence of future accesses is precisely recorded in the recipe. Using a look-ahead window or other methods can identify the future information to achieve a more effective caching policy. Lillibridge *et al.* [55] propose a forward assembly scheme. The proposed scheme reserves and uses multiple container size buffer

in FAA to restore the data chunks with a look-ahead window which is the same size as FAA. Park *et al.* [141] use a fixed size look-ahead window to identify the cold containers and evict them first in a container-based caching scheme.

The following issues are not fully addressed in the existing work. First, the performance and efficiency of container-based caching, chunk-based caching and forward assembly vary as the workload locality changes. When the total size of available memory for restore is fixed, how to use these schemes in an efficient way and make them adapt to the workload changing are very challenging. Second, how big is the look-ahead window and how to use the future information in the look-ahead window to improve the cache hit ratio are less explored. Third, acquiring and processing the future access information in the look-ahead window requires computing overhead. How to make better trade-offs to achieve good restore performance, but limit the computing overhead is also an important issue.

To address these issues, in this research we design a hybrid scheme which combines chunk-based caching and forward assembly. We also propose new ways of exploiting the future access information obtained in the look-ahead window to make better decisions on which data chunks are to be cached or evicted. The sizes of the look-ahead window, chunk cache, and FAA are dynamically adjusted to reflect the future access information in the recipe.

In this research, we first propose a look-ahead window assisted chunk-based caching scheme. A large Look-Ahead Window (LAW) provides the future data chunk access information to both FAA and chunk cache. Note that the first portion of the LAW (as the same size as that of FAA) is used to place chunks in FAA and the second part of the LAW is used to identify the caching candidates, evicting victims and accessing sequence of data chunks. We will only cache the data chunks that appear in the current LAW. Therefore, a cached data chunk is classified as either an *F-chunk* or a *P-chunk*. *F-chunks* are the data chunks that will be used in the near future (appear in the second part of LAW). *P-chunks* are the data chunks that only appear in the first part of the LAW. If most of the cache space is occupied by the *F-chunks*, we may want to increase the cache space. If most of the cache space is occupied by *P-chunks*, caching is not very effective at this moment. We may consider to reduce the cache space or to enlarge the LAW.

Based on the variation of the numbers of *F-chunks*, *P-chunks*, and other measurements, we then propose a self-adaptive algorithm (ALACC) to dynamically adjust the sizes of memory space allocated for FAA and chunk cache, and the size of LAW. If the number of *F-chunks* is low, ALACC extends the LAW size to identify more *F-chunks* to be cached. If the number of *P-chunks* is extremely high, ALACC either reduces the cache size or enlarges LAW size to adapt to the current access pattern. When the monitored measurements indicate that FAA performs better, ALACC increases the FAA size, thus reduces the chunk caching space, and gradually shrinks LAW. Since we consider a fixed amount of available memory, a reduction of chunk cache space will increase the same size of FAA or vice versa. For the reason that LAW only involves meta-data information which takes up a smaller data space, we ignore the space required by LAW, but focus more on the computing overhead caused by the operations of LAW in this research.

Our contributions can be summarized as follows:

- We comprehensively investigate the performance trade-offs of container-based caching, chunk-based caching and forward assembly in different workloads and memory configurations.
- We propose ALACC to dynamically adjust the sizes of FAA and chunk cache to adapt to the changing of chunk locality to get the best restore performance.
- By exploring the cache efficiency and overhead of different LAW size, we propose and implement an effective LAW with its size dynamically adjusted to provide essential information for FAA and chunk cache and avoid unnecessary overhead.

The rest of the chapter is arranged as follows. Section 2 reviews the background of data deduplication and the current schemes of improving restore performance. Section 3 compares and analyzes different caching schemes. We first present a scheme with the pre-determined and fixed sizes of the forward assembly area, chunk cache, and LAW in Section 4. Then, the adaptive algorithm is proposed and discussed in Section 5. A brief introduction of the prototype implementation is in Section 6 and the evaluation results and analyses are shown in Section 7. Finally, we provide some conclusions and discuss the future work in Section 8.

4.2 Background and Related Work

In this section, we first review the deduplication and restore process. Then, the related studies of improving restore performance are presented and discussed.

4.2.1 Data Deduplication Preliminary

Data deduplication is widely used in the secondary storage systems such as archiving and backup systems to improve the storage space utilization [36, 88, 35, 89, 90, 91, 142, 143]. Recently, data deduplication is also applied in the primary storage systems such as SSD array to make better trade-offs between cost and performance [92, 93, 94, 95, 96, 86, 97, 98]. To briefly summarize deduplication, as a data stream is written to the storage system, it is divided into data chunks, which are represented by a secure hash value called a fingerprint. The chunk fingerprints are searched in the indexing table to check their uniqueness. Only the new unique chunks are written to containers, and the original data stream is represented with a recipe consisting of a list of data chunk meta-information including the fingerprint.

Restoring the original data stream back is the reverse process of deduplication. Starting from the beginning of the recipe, the restore engine identifies the data chunk meta-information sequentially, accesses the chunks either from memory or from storage, and assembles the chunks in an assembling buffer in memory. To get a chunk from storage to memory, the entire container holding the data chunk will be read, and the container may be distant from the last accessed container. Once the buffer is full, data is flushed out to the requested client.

In the worst case, to assemble N duplicated chunks, we may need N container-reads. A straightforward solution to reduce the container-reads is to cache some of the containers or data chunks. Since some data chunks will be used very shortly after they are read into memory, these cached chunks can be directly copied from cache to the assembling buffer which can reduce the number of container-reads. Another way to reduce the number of container-reads is to store (re-write) some of the duplicated data chunks together with the unique chunks during the deduplication process in the same container. Therefore, the duplicated chunks and unique chunks will be read out together in the same container and thus avoids the needs of accessing these duplicated

chunks from other containers. However, this approach will reduce the effectiveness of data deduplication.

4.2.2 Related Work on Restore Performance Improvement

Selecting and storing some duplicated data chunks during the deduplication process and designing efficient caching policies during the restore process are two major research directions to improve the restore performance. In the remaining of this subsection, we first review the studies of selectively storing the duplicated chunks. Then, we introduce the container-based caching, chunk-based caching and forward assembly.

There have been several studies focusing on how to select and store the duplicated data chunks to improve the restore performance. The duplicated data chunks have already been written to the storage when they first appeared as unique data chunks and dispersed over different physical locations in different containers, which creates the chunk fragmentation issue [101]. During the restore process, restoring these duplicated data chunks causes potential random container-reads which lead to a low restore throughput. Nam *et al.* [101, 102] propose a way to measure the chunk fragmentation level (CFL). By storing some of the duplicated chunks to keep the CFL lower than a given threshold in a segment of the recipe, the number of container-reads is reduced.

Kaczmarczyk *et al.* [100] use the mismatching degree between the stream context and disk context of the chunks to make the decision of storing selected duplicated data chunks. The container capping is proposed by Lillibridge *et al.* [55]. The containers storing the duplicated chunks are ranked and the duplicated data chunks in the lower ranking containers are selected and stored again. In a historical-based duplicated chunk rewriting algorithm [103], the duplicated chunks in the inherited sparse containers are rewritten. Due to the fact that re-writing some selected duplicated data chunks again sacrifices the deduplication ratio and the selecting and re-writing can be applied separately during the deduplication process, we will consider only the restore process in this chapter.

Different caching policies are studied in [55, 100, 101, 102, 104, 141, 22, 144]. Kaczmarczyk *et al.* [100] and Nam *et al.* [101, 102] use container-based caching. Other than using recency to identify the victims in the cache, Park *et al.* [141] propose a future reference count based caching policy with the information from a fixed size look-head

window. Belady's optimal replacement policy can only be used in a container-based caching schema [103]. It requires extra effort to identify and store the replacement sequence during the deduplication. If a smaller caching granularity is used, a better performance can be achieved. Instead of caching containers, some of the studies directly cache data chunks to achieve higher cache hit ratio [55, 104]. Although container-based caching has lower operating cost, chunk-based caching can better filter out the data chunks that are irrelevant to the near future restore and better improve the cache space utilization.

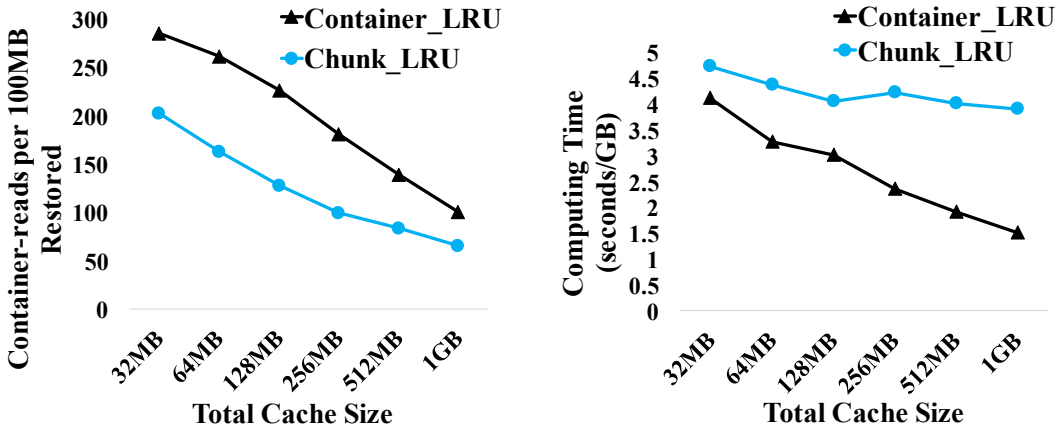
However, the chunk-based caching with LRU also has some performance issues. The historical based LRU may fail to identify data chunks in the read-in container which are not used in the past and current assembling buffers, but they will be used in the near future. This results in cache misses. To address this issue, a look-ahead window which covers a range of future accesses from the recipe can provide the crucial future access information to improve the cache effectiveness.

A special fashion of chunk-based caching proposed by Lillibridge *et al.* is called forward assembly [55]. Multiple containers (say k) are used as assembling buffers called Forward Assembly Area (FAA) and a look-ahead window of the same size is used to identify the data chunks to be restored in the next k containers. FAA can be considered as a chunk-based caching algorithm. It caches all the data chunks that appear in the next k containers and evicts any data chunk which does not appear in these containers. Since data chunks are directly copied from container-read buffer to FAA, it avoids the memory-copy operations from the container-read buffer to the cache. Therefore, forward assembly has lower overhead comparing with the chunk-based caching. Forward assembly can be very effective if each unique data chunk will re-appear in a short range from the time it is being restored.

As discussed, there is still a big potential to improve the restore performance if we effectively combine forward assembly and chunk-based caching using the future access information in the LAW. In this research, we consider the total amount memory available to FAA and chunk cache is fixed. If the memory allocation for these two can vary according to the locality changing, the number of container-reads may be further reduced.

Table 4.1: Data Sets

Data Set Name	ds_1	ds_2	ds_3
Deduplication Ratio	1.03	2.35	2.11
Reuse Distance (# containers)	24	18	26



(a) # Containers-reads per 100MB restored as cache size varies from 32MB to 1GB

(b) Computing time per 1GB restored as cache size varies from 32MB to 1GB

Figure 4.1: The cache efficiency comparison between chunk-based caching and container-based caching

4.3 Analysis of Cache Efficiency

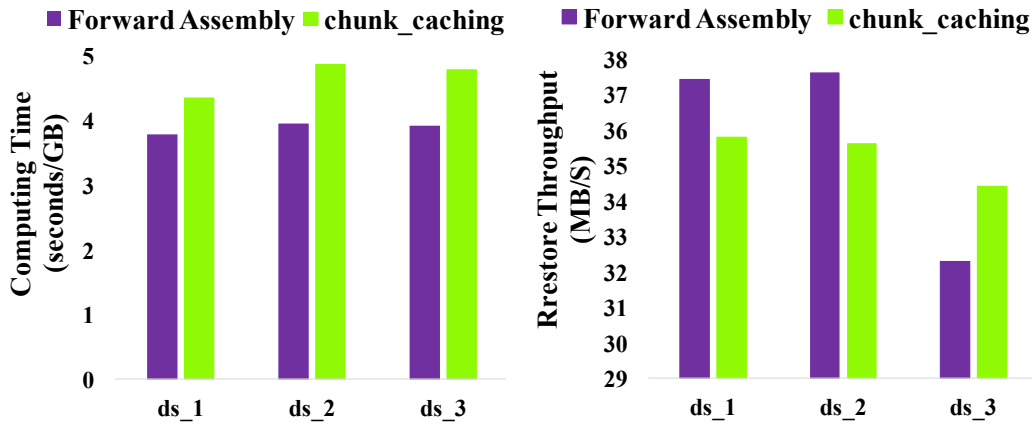
Before we start to discuss the details of our proposed design, we first compare and analyze the cache efficiency of container-based caching, chunk-based caching, and forward assembly. The observations and knowledge we learned will help our design. The traces used in the experiments of this section are summarized in Table 4.1. ds_1 and ds_2 are the last version of EMC_1 and FSL_1 traces respectively which are introduced in detail in Section 5.6.1. ds_3 is a synthetic trace based on ds_2 with larger re-use distance. The container size is 4MB. Computing time is used to measure the management overhead. It is defined as the total restore time excluding the storage I/O time which includes the cache adjustment time, memory-copy time, CPU operation time, and others.

4.3.1 Caching Chunks vs. Caching Containers

Technically, caching containers can avoid the memory-copy from the container-read buffer to the cache. If the entire cache space is the same, the cache management overhead of container-based caching is lower than that of chunk-based caching. In most cases, some data chunks in a read-in container are irrelevant to the current and near-future restore process. Container-based caching still caches these chunks and wastes the valuable memory space. Also, some of the useful data chunks are forced to be evicted together with the whole container which increases the cache miss ratio. Thus, without considering managing overhead, caching chunks can achieve better cache hit ratio than caching containers if we apply the same cache policy in most workloads. This is especially true if we use LAW as a guidance of future accesses. Only in the extreme cases that most data chunks in the container are used very shortly and there is very high temporal based locality, the cache hit ratio of caching containers can be better than that of caching chunks.

We use the ds_2 trace to evaluate and compare the number of container-reads and the computing overhead of caching chunks and caching containers. We implemented the container-based caching and chunk-based caching with the same LRU policy. The assembling buffer used is one container size. As shown in Figure 4.1(b), the computing time of restoring 1GB data of caching chunks is about 15-150% higher than that of caching containers. Theoretically, the LRU cache insertion, lookup and eviction are $\mathcal{O}(1)$ time complexity. However, the computing time drops in both designs as the cache size increases. The reason is that with larger memory size more containers or data chunks can be cached and the cache eviction happens less frequently. There will be fewer memory-copy of containers or data chunks, which leads to less computing time.

Also, due to fewer cache replacements in the container-based caching, the computing time of caching containers drops quicker than that of caching chunks. However, as shown in Figure 4.1(a), the number of container-reads of caching chunks is about 30-45% fewer than that of caching containers. If one container-read needs 30ms, about 3% fewer container-reads can cover the extra computing time overhead of caching chunks. Therefore, caching chunks is preferred in the restore process, especially when the cache space is limited.



(a) Computing time per 1GB restored

(b) Restore throughput

Figure 4.2: The cache efficiency comparison between forward assembly and chunk-based caching

4.3.2 Forward Assembly vs. Caching Chunks

Comparing with chunk-based caching, forward assembly does not have the overhead of cache replacement and the overhead of copying data chunks from container-read buffer to the cache. Thus, the computing time of forward assembly is much lower than that of chunk-based caching. As for the cache efficiency, the two approaches have their own advantages and disadvantages with different data chunk localities. If most of the data chunks are unique or the chunk re-use locality is high in a short range (e.g., within the FAA range), forward assembly performs better than chunk-based caching. In this scenario, most of the data chunks in the read-in containers will only be used to fill the current FAA. In the same scenario, if chunk-based caching can intelligently choose the data chunks to be cached based on the future access information in LAW, it can achieve a similar number of container-reads but it has larger computing overhead.

If the re-use distances of many duplicated data chunks are out of the FAA range, chunk-based caching performs better than forward assembly. Since these duplicated data chunks could not be allocated in the FAA, extra container-reads are needed when restoring these data chunks via forward assembly. In addition, a chunk in the chunk cache is only stored once while in forward assembly it needs to be stored multiple times in its appearing locations in the FAA. Thus, chunk-based caching can potentially cover

more data chunks with larger re-use distances. When the managing overhead is less than the time saved by the reduction of container-reads, chunk-based caching performs better.

We use the aforementioned three traces to compare the efficiency of forward assembly with that of chunk-based caching. The memory space used by both schemes is 64MB. For chunk-based caching, it uses the same size LAW as FAA to provide the information of data chunks accessed in the future. It caches the data chunks that appear in the LAW first, then adopts the LRU as its caching policy to manage the rest of caching space. As shown in Figure 4.2(a), the computing time of forward assembly is smaller than that of chunk-based caching. The restore throughput of forward assembly is higher than that of chunk based caching except in ds_3 as shown in Figure 4.2(b). In ds_3, the re-use distances of 77% of the duplicated data chunks are larger than the FAA range. More cache misses occur in forward assembly, while chunk-based caching can effectively cache some of these data chunks. In general, if the deduplication ratio is extremely low or most of the chunk re-use distances can be covered by the FAA, the performance of forward assembly is better than that of chunk-based caching. Otherwise, chunk-based caching can be a better choice.

4.4 Look-Ahead Window Assisted Chunk-based Caching

As discussed in the previous sections, combining forward assembly with chunk-based caching can potentially adapt to various workloads and achieve better restore performance. In this section, we design a restore algorithm with the assumption that the sizes of FAA, chunk cache and LAW are all fixed. We call the memory space used by chunk-based caching *chunk cache*. We first discuss how the FAA, chunk-based caching and LAW cooperate together to restore the data stream. Then, a detailed example is presented to better demonstrate our design.

FAA consists of several container size memory buffers and they are arranged in a linear order. We call each container size buffer an FAB. A restore pointer pinpoints the data chunk in the recipe to be found and copied to its corresponding location in the first FAB at this time. The data chunks before the pointer are already assembled. Other FABs are used to hold the accessed data chunks if these chunks also appeared in these

FABs. LAW starts with the first data chunk in the first FAB (FAB1 in the example) and covers a range bigger than that of FAA in the recipe. In fact, the first portion of the LAW (equivalent to the size of FAA) is used for the forward assembly purpose and the remaining part of LAW is used for the purpose of chunk-based caching.

An assembling cycle is the process to completely restore the first FAB in the FAA. That is, the duration after the previous first FAB is written back to the client and a new empty FAB is added to the end of FAA to the time when the new first FAB is completely filled up. At the end of one assembling cycle (i.e., the first FAB is completely assembled), the content of the FAB is written back to the requested client as the restored data and this FAB is removed from the FAA. At the same time, a new empty FAB will be added to the end of the FAA so that FAA is maintained with the same size. For the LAW, the first portion (one container size) corresponding to the removed FAB is dropped and one more segment of recipe (also one container size) is appended to the end of LAW. Then, the restore engine starts the next assembly cycle to fill in the new first FAB of the current FAA.

During the assembling cycle, the following is the procedure of restoring the data chunk pointed by the restore pointer. If the data chunk at the restore pointer has already been stored by previous assembling operations, the pointer will be directly moved to the next one. If the data chunk has not been stored at its location in the FAB, the chunk cache is checked first. If the chunk is found in the cache, the data chunk is copied to the locations where this data chunk appears in all the FABs including the location pointed by the restore pointer. Also, the priority of the chunk in the cache is adjusted accordingly. If the chunk is not in the chunk cache, the container that holds the data chunk will be read in from the storage. Then, each data chunk in this container is checked with LAW to identify all the locations it appears in the FAA. Then, the data chunk is copied to the corresponding locations in all FABs if they exist. Next, we have to decide which chunks in this container are to be inserted to the chunk cache according to a caching policy, and it will be described later. After this data chunk is restored, the restore pointer moves to the next data chunk in the recipe. The read-in container will be replaced by the next requested container. An example is shown in Figure 4.3, and it will be described in detail in the last part of this section.

When inserting data chunks from the read-in container to the chunk cache, we need

to identify the potential usage of these chunks in the LAW and treat them accordingly. Based on the second portion of LAW (i.e., the remaining LAW after the size of FAA), the data chunks from the read-in container can be classified into three categories: 1) *U-chunk* (Unused chunk) is a data chunk that does not appear in the current entire LAW, 2) *P-chunk* (Probably used chunk) is a data chunk that appears in the current FAA but does not appear in the second portion of the LAW, and 3) *F-chunk* (Future used chunk) is a data chunk that will be used in the second portion of the LAW. Note that a *F-chunk* may or may not be used in the FAA.

F-chunks are the data chunks that should be cached. If more cache space is still available, we may cache some *P-chunks* according to their priorities. That is, *F-chunks* have priority over *P-chunks* for caching. However, each of them has a different priority policy. The priority of *F-chunks* is defined based on the ordering of their appearance in the second portion of the LAW. That is, an *F-chunk* to be used in the near future in the LAW has a higher priority over another *F-chunk* which will be used later in the LAW. The priority of *P-chunks* is LRU based. That is, the most recently used (MRU) *P-chunk* has a higher priority over the least recently used *P-chunks*. Let us denote the cache space used by *F-chunks* (*P-chunks*) as F-cache (P-cache). The boundary between the F-cache and P-cache is dynamically changing as the number of *F-chunks* and that of *P-chunks* vary.

When the LAW advances, new *F-chunks* are added to the F-cache. An *F-chunk* that has been restored and no longer appeared in the LAW will be moved to the MRU end of the P-cache. That is, this chunk becomes a *P-chunk*. The priorities of some *F-chunks* are adjusted according to their future access sequences. The new *P-chunks* are added to the P-cache based on the LRU order of their last appearance. Some *P-chunks* may become *F-chunks* if they appear in the newly added portion of LAW. When the cache eviction happens, data chunks are evicted from the LRU end of the P-cache first. If there is no *P-chunk* in P-cache, eviction starts from the lowest priority end of the F-cache.

Figure 4.3 is an example to show the entire working process of our design. Suppose one container holds 4 data chunks. The LAW covers the data chunks of 4 containers in the recipe from chunk 18 to chunk 28. The data chunks before the LAW have been assembled and written back to the client. The data chunks beyond the LAW is unknown

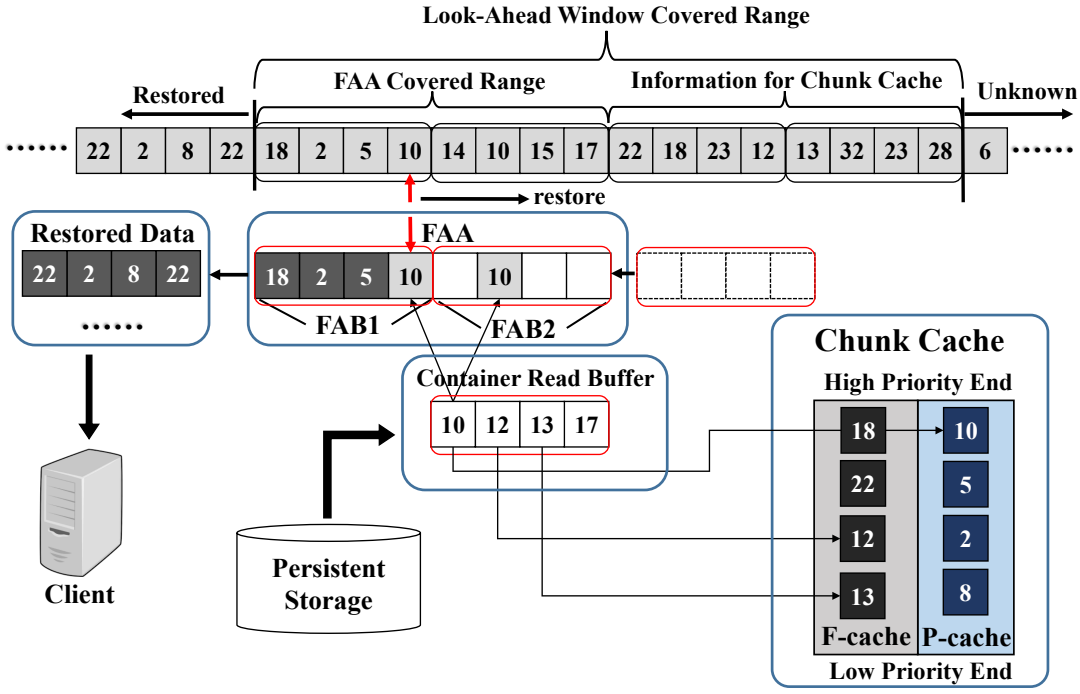


Figure 4.3: An example of look-ahead window assisted chunk-based caching

to the restore engine at this moment. There are two buffers in the FAA denoted as FAB1 and FAB2. Each FAB has a size of one container and also holds 4 data chunks. FAA covers the range from chunk 18 to chunk 17. The rest information of data chunks in the LAW (from chunk 22 to chunk 28) are used for the chunk cache. The red frames in the figure show the separations of data chunks in containers. The labeled chunk number represents the chunk ID and is irrelevant to the order of the data chunk appearing in the recipe.

In FAB1, chunks 18, 2, 5 have already been stored and the current restore pointer is at data chunk 10 (pointed by the red arrow). This data chunk has neither been allocated nor been cached. The container that stores chunk 10, 12, 13 and 17 is read out from the storage to the container read buffer. Then, the data of chunk 10 is copied to the FAB1. At the same time, chunk 10 also appears in the FAB2 and the chunk is stored in the corresponding position too. Next, the restore pointer moves to the chunk 14 at the beginning of the FAB2. Since FAB1 has been fully assembled, its content is written out as restored data and it is removed from FAA. The original FAB2 becomes

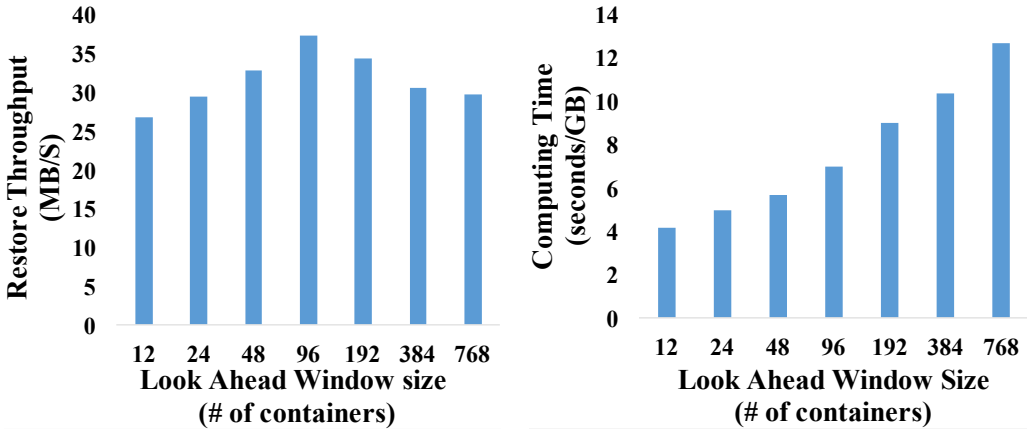
the new FAB1 and a new FAB (represented with dotted frames) is added after FAB1 and becomes the new FAB2.

All the data chunks in the container read buffer are checked with the LAW. Data chunk 10 is used in the FAA but it does not appear again in the rest of LAW. So chunk 10 is a *P-chunk* and it is inserted to the MRU end of the P-cache. Chunk 12 and chunk 13 are not used in the current FAA but they will be used in the next two assembling cycles within the LAW. They are identified as *F-chunks* and added to the F-cache. Notice that chunk 12 appears after chunk 22 and chunk 13 is used after chunk 12 as shown in the recipe. Therefore, chunk 13 is inserted into the low priority end and chunk 12 has the priority higher than chunk 13. Chunk 17 has neither been used in the FAA nor appeared in the LAW. It is a *U-chunk* and it will not be cached. When restoring chunk 14, a new container will be read into the container read buffer and it will replace the current one.

4.5 The Adaptive Algorithm

The performance of the look-ahead window assisted chunk-based caching is better than that of simply combining the forward assembly with LRU-based chunk caching. However, the sizes of FAA, chunk cache and LAW are pre-determined and fixed in this design. As discussed in the previous sections, FAA and chunk cache have their own advantages and disadvantages for different workloads. In fact, in a given workload like a backup trace, most data chunks are unique at the beginning. Later in different sections of the workload, they may have various degrees of duplication and re-usage.

Therefore, the sizes of FAA and chunk cache can be dynamically changed to reflect the locality of the current section of a workload. It is also important to figure out what the appropriate LAW size is to get the best restore performance given a configuration of the FAA and chunk cache. We propose an adaptive algorithm called ALACC that can dynamically adjust the sizes of FAA, chunk cache and LAW according to the workload variation during the restore process. First, we evaluate and analyze the restore performance of using different LAW sizes. Then, we present the details of ALACC.



(a) Restore throughput

(b) Computing time per 1GB restored

Figure 4.4: The restore performance and computing overhead variation as the LAW size increases

4.5.1 Performance Impact of LAW Size

We did an experiment that varies the LAW sizes for a given workload and compared the restore throughput and required computing overhead. We use an 8MB (2 container size) FAA and a 24MB (6 container size) chunk cache as the memory space configuration and increase the LAW size from 12 container size to 768 container size (it covers the original data stream size from 48MB to 3GB). As shown in Figure 4.4, the computing time continuously increases as the LAW size increases due to higher overhead to process and maintain the information in the LAW. When the size of LAW is larger than 96, the restore throughput starts to decrease. The performance degradation is caused by the increase of computing overhead and less efficiency of chunk cache. Thus, using an appropriate LAW size is important to make better trade-offs between cache efficiency and computing overhead.

We can explain the observation by analyzing the chunk cache efficiency. Suppose the LAW size is S_{LAW} chunks, the FAA size is S_{FAA} chunks, and the chunk cache size is S_{cache} chunks. Note that we use a container size as the basic unit for allocation and the number of containers can be easily translated to the number of chunks. Assume one data chunk C_i is used at the beginning of the FAA and it will be reused after D_{C_i} chunks (i.e., the reuse distance of this chunk). If $D_{C_i} < S_{FAA}$, it will be reused in the FAA.

If $S_{FAA} < D_{C_i}$, this chunk should be either an *F-chunk* or a *P-chunk*. However, the chunk category highly depends on the size of S_{LAW}. If the LAW only covers the FAA range ($S_{FAA} = S_{LAW} < D_{C_i}$), the proposed algorithm degenerates to the LRU-based chunk caching algorithm. If $S_{FAA} < D_{C_i} < S_{LAW}$, we can definitely identify this chunk as an *F-chunk* and decide to cache this chunk. However, if $S_{LAW} < D_{C_i}$, this chunk will be identified as a *P-chunk* and it may or may not be cached (this depends on the available space in cache). Therefore, at least we should ensure $S_{FAA} + S_{cache} \leq S_{LAW}$.

If the LAW size is large enough and most of the chunk cache space are occupied by *F-chunks*, the cache efficiency is high. However, once the cache is full of *F-chunks*, the newly identified *F-chunk* may or may not be inserted into the cache. This depends on its reuse order. Thus, continuing increase the LAW size would not further improve the cache efficiency. What is worse, a larger LAW size requires more CPU and memory resources to identify and maintain the future access information. As we discuss before, if S_{LAW} is the same as S_{FAA}, all cached chunks are *P-chunks* and chunk caching becomes an LRU-based algorithm.

Thus, the best trade-off between cache efficiency and overhead is when the total number of *P-chunks* is as low as possible but not 0. However, it may be hard to maintain the number of *P-chunks* low all the time, especially when there is a very limited number of *F-chunks* identified with a large LAW size. In this case, the size of LAW should not be extended or it should be decreased slightly.

4.5.2 ALACC

Based on the previous analysis, we propose an adaptive algorithm, which dynamically adjusts the memory space ratio of FAA and chunk cache, and the size of LAW. We apply the adjustment at the end of each assembling cycle right after the first FAB is restored. Suppose at the end of i_{th} assembling cycle, the sizes of FAA, chunk cache and LAW are S^i_{FAA}, S^i_{cache} and S^i_{LAW} container size respectively. $S^i_{FAA} + S^i_{cache}$ is fixed. To avoid an extremely large LAW size, we set a maximum size of LAW (Max_{LAW}) that is allowed during the restore.

The algorithm of ALACC optimizes the memory allocation to FAA and chunk cache first. On one hand, if the workload has an extremely low locality or its duplicated data chunk re-use distance is small, FAA is preferred. On the other hand, according to the

total number of *P-chunk* in the cache, the chunk cache size is adjusted. Then, according to the memory space allocation changing and the total number of *F-chunk* in the cache, the LAW size is adjusted to optimize the computing overhead. The detailed adjustment conditions and actions are described in the following part of this section.

In our algorithm, the conditions of increasing the FAA size are examined first. If any of these conditions is satisfied, FAA will be increased by 1 container size and the size of chunk cache will be decreased by 1 container accordingly. Otherwise, we will check the conditions of adjusting the chunk cache size. Notice that the size of FAA and chunk cache could remain the same if none of the adjustment conditions are satisfied. Finally, the LAW size will be changed.

FAA and LAW Size Adjustment. As discussed in Section 3.2, FAA performs better than chunk cache when 1) the data chunks in the first FAB are identified mostly as unique data chunks and these chunks are stored in the same or close containers, or 2) the re-use distance of most duplicated data chunks in this FAB is within the FAA range. Regarding the first condition, we consider that the FAB can be filled in by reading in no more than 2 containers and none of the data chunks needed by the FAB is from the chunk cache. When this happens, we consider this assembling cycle FAA effective. For Condition 2, we observed that if the re-use distances of 80% or more of the duplicated chunks in this FAB is smaller than the FAA size, forward assembly performs better.

Therefore, based on the observations, we use either of the following two conditions to increase the FAA size. First, if the number of consecutive FAA effective assembling cycles becomes bigger than a given threshold, we increase the FAA size by 1 container. Here, we use the current FAA size, S^i_{FAA}, as the threshold to measure this condition at the end of i_{th} assembling cycle. When S^i_{FAA} size is small, the condition is easier to satisfy and the size of FAA can be increased faster. When S^i_{FAA} is large, the condition is more difficult to satisfy. After increasing the FAA size by one, the count of consecutive FAA effective assembling cycles is reset to 0. Second, the data chunks used during the i_{th} assembling cycle to fill up the first FAB in FAA are examined. If the re-use distances of more than 80% of these examined chunks during the i_{th} assembling cycle are smaller than $S^i_{FAA} + 1$ container size, the size of FAA will be increased by 1 container. That is, $S^{i+1}_{FAA} = S^i_{FAA} + 1$.

If the FAA size is increased by 1 container, the size of chunk cache will decrease by 1

container accordingly. Originally, $S^i_{LAW} - S^i_{FAA}$ container size LAW information is used by S^i_{cache} container size cache. After the FAA adjustment, $S^i_{LAW} - S^i_{FAA} + 1$ container size LAW is used by $S^i_{cache} - 1$ container size cache, which wastes the information in the LAW. Thus, the LAW size is decreased by 1 container size to avoid the same size LAW used by a now smaller size chunk cache. After the new sizes of FAA, chunk cache and LAW are decided, two empty FABs (one to replace the re-stored FAB and the other reflects the increasing size of FAA) will be added to the end of FAA and the chunk cache starts to evict data chunks. Then, the $(i+1)_{th}$ assembling cycle will start.

Chunk Cache and LAW Size Adjustment. If there is no adjustment to the FAA size, we now consider the adjustment of chunk cache size. After finished the i_{th} assembling cycle, the total number ($N_{F-chunk}$) of *F-chunks* in the F-cache and the total number ($N_{P-chunk}$) of *P-chunks* in the P-cache are counted. Also, the number of *F-chunks* that are newly added to the F-cache during the i_{th} assembling cycle is denoted by $N_{F-added}$. These newly added *F-chunks* either come from the read-in containers in the i_{th} assembling cycle or are transformed from *P-chunks* due to the extending of the LAW. We examine the following three conditions.

First, if $N_{P-chunk}$ becomes 0, it indicates that all the cache space is occupied by *F-chunks*. The current LAW size is too large and the number of *F-chunks* based on the current LAW is larger than the chunk cache capacity. Therefore, the chunk cache size will be increased by 1 container. Meanwhile, the size of LAW will decrease by 1 container to reduce the unnecessary overhead. Second, if the total size of $N_{F-added}$ is bigger than 1 container, it indicates that the total number of *F-chunks* increases very quickly. Thus, we increase the chunk cache size by 1 container and decrease LAW by 1 container. Notice that a large $N_{F-added}$ can happen when $N_{P-chunk}$ is either small or large. This condition will make our algorithm quickly react to the changing of the workload.

Third, if $N_{P-chunk}$ is very large (i.e., $N_{F-chunk}$ is very small), the chunk cache size will be decreased by 1 container. In this situation, the LAW size is adjusted differently according to either of the following two conditions: 1) In the current workload, there are few data chunks in the FAB that are reused in the future, and 2) The size of LAW is too small, it cannot look ahead far enough to find more *F-chunks* for the current workload. For Condition 1, we decrease the LAW size by 1 container. For

Condition 2, we increase the LAW size by K containers. Here, K is calculated by $K = (Max_{LAW} - S^i_{LAW})/(S^i_{FAA} + S^i_{cache})$. If LAW is small, its size is increased by a larger amount. If LAW is big, its size will be increased slowly.

LAW Size Independent Adjustment If none of the aforementioned conditions are satisfied (the sizes of FAA and chunk cache remain the same), the LAW size will be adjusted independently. Here, we use the $N_{F-chunk}$ to decide the adjustment. If $N_{F-chunk}$ is smaller than a given threshold (e.g., 20% of the total chunk cache size), the LAW size will be slightly increased by 1 container to process more future information. If $N_{F-chunk}$ is higher than the threshold, the LAW size will be decreased by 1 to reduce the computing overhead.

ALACC makes the trade-offs between computing overhead and the reduction of container-reads, such that a higher restore throughput can be achieved. Instead of using a fixed value as the threshold, we tend to dynamically change FAA size and LAW size. It can slow down the adjustments when the overhead is big (when the LAW size is large) and it can speed up the adjustment when the overhead is small to quickly reduce the container-reads (when FAA size is small).

4.6 Prototype Implementation

We implemented a prototype of a deduplication system (a C program with 11k LoC) with several restore designs: FAA, container-based caching, chunk-based caching, LAW assisted chunk-based caching, and ALACC. For the deduplication process, it can be configured to use different container size and chunk size (fixed size chunking and variable size chunking) to process the real world data or to generate deduplication traces.

To satisfy the flexibility and efficiency requirements of ALACC, we implemented several data structures. All the data chunks in the chunk cache are indexed by a hash-map to speed up the searching operation. *F-chunks* are ordered by their future access sequence provided by the LAW and *P-chunks* are indexed by an LRU list. The LAW maintains the data chunk metadata (chunk ID, size, container ID, address and offset in the container) in the same order as they are in the recipe. Although the insertion and deletion in the LAW is $\mathcal{O}(1)$ by using the hashmap, identifying the *F-chunk* priority is $\mathcal{O}(\log(N))$, where N is the LAW size.

A Restore Recovery Log (RRL) is maintained to ensure reliability. When one FAB is full and flushed out, the chunk ID of the last chunk in the FAB, the chunk location in the recipe, the restored data address, FAA, chunk cache and LAW configurations are logged to the RRL. If the system is down, by using the information in the RRL, the restore process can be recovered to the latest restored cycle. The FAA, chunk cache and LAW will be initiated and reconstructed.

4.7 Performance Evaluation

To comprehensively evaluate our design, we implement five restore engines including ALACC, LRU-based container caching (Container_LRU), LRU-based chunk caching (Chunk_LRU), forward assembly (FAA), and fixed combination of forward assembly and chunk-based caching. However, in the last case, we show only the Optimal Fix Configuration (Fix_Opt). Fix_Opt is obtained by exhausting all possible fixed combinations of FAA, chunk cache, and LAW sizes.

4.7.1 Experimental Setup and Data Sets

The prototype is deployed on a Dell PowerEdge R430 server with a 2.40GHz Intel Xeon with 24 cores and 32GB of memory using Seagate ST1000NM0033-9ZM173 SATA hard disk with 1TB capacity as the storage. The container size is configured as 4MB. All five implementations are configured with one container size space as container read buffer and memory size of S containers. Container_LRU and Chunk_LRU use one container for FAA and $S-1$ for container or chunk cache. FAA uses LAW of S container size and all S memory space as the forward assembly area. The specific configuration of Fix_Opt is given in Section 4.7.2. In all experiments, ALACC is initiated with $S/2$ container size FAA, $S/2$ container size chunk cache and $2S$ container size LAW before the execution. For the reason that ALACC requires to maintain at least one container size space as FAB, the FAA size varies from 1 to S and the chunk cache size varies from $(S-1)$ to 0 accordingly. After one version of backup is restored, the cache is cleaned.

We use four deduplication traces in the experiments as shown in Table 5.1. FSL_1 and FSL_2 are two different backup traces from FSL /home directory snapshots of the year 2014 [145]. Each trace has 6 full snapshot backups and the average chunk size is

Table 4.2: Characteristics of datasets

Dataset	FSL_1	FSL_2	EMC_1	EMC_2
Size	103.5GB	317.4GB	29.2GB	28.6GB
ACS	4KB	4KB	8KB	8KB
DR	3.82	4.88	1.04	4.8
CFL	13.3	3.3	14.7	19.3

ACS stands for Average Chunk Size
DR stands for the Deduplication Ratio, which is the original data size divided by the deduplicated data size.
CFL stands for the Chunk Fragmentation Level, which is the average number of containers that stores the data chunks from one container size of original data stream. High CFL value leads to low restore performance.

4KB. EMC_1 and EMC_2 are the weekly full-backup traces from EMC and each trace has 6 versions and 8KB average chunk size [146]. EMC_1 was collected from an exchange server and EMC_2 was the */var* directory backup from a revision control system.

To measure the restore performance, we use the *speed factor*, *computing cost factor* and *restore throughput* as the metrics. The speed factor (MB/container-read) is defined as the mean data size restored per container read. Higher speed factors indicate higher restore performance. The computing cost factor (second/GB) is defined as the time spent on computing operations (subtracting the storage I/O time from the restore time) per GB data restored and the smaller value is preferred. The restore throughput (MB/second) is calculated from the original data stream size divided by the total restore time. We run each test 5 times and present the mean value. Notice that, for the same trace using the same caching policy, if the restore machine and the storage are different, the computing cost factor and restore throughput can be different while the speed factor is the same.

4.7.2 Optimal Performance of Fixed Configurations

In LAW assisted chunk-based caching design, the sizes of FAA, chunk cache and LAW are fixed and are not changed during the restore process. To find out the best performance of a configuration for a specific trace with a given memory size, in our experiments we run all possible configurations for each trace to discover the optimal throughput. This optimal configuration is indicated by Fix_Opt. For example, for 64MB memory, we vary

Table 4.3: The FAA/chunk cache/LAW configuration (# of containers) of Fix_Opt for each deduplication trace

	FSL_1	FSL_2	EMC_1	EMC_2
Size	4/12/56	6/10/72	2/14/92	4/12/64

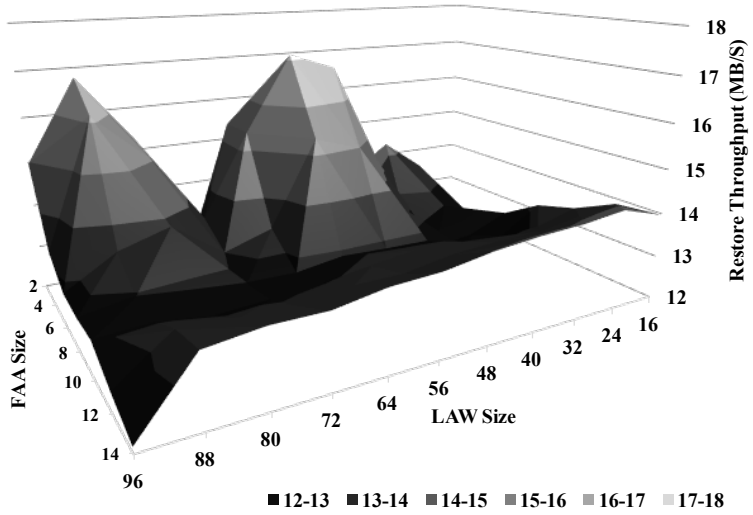
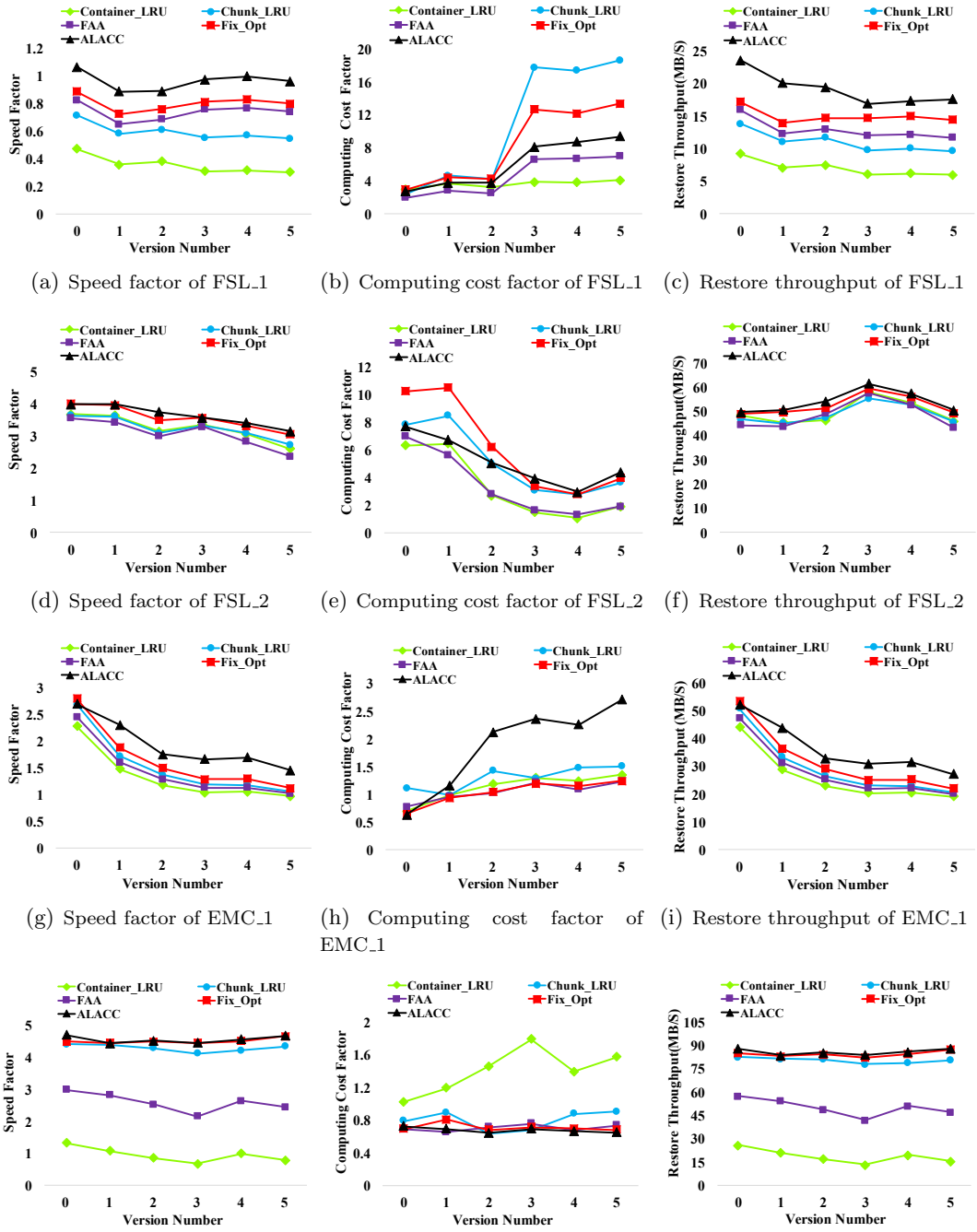


Figure 4.5: The restore throughput of different FAA, chunk cache and LAW size configurations

the FAA size from 4MB to 60MB and the chunk cache size from 60MB to 4MB. At the same time, the LAW size increases from 16 containers to 96 containers. Each test tries one set of fixed configuration and finally, we draw a three-dimensional figure to find out the optimal results.

An example is shown in Figure 4.5, for FSL_1, the optimal configuration has 4 containers of FAA, 12 containers of chunk cache and a LAW size of 56 containers. One throughput peak is when the LAW is small. The computing cost is low while the container-reads are slightly higher. The other throughput peak is when the LAW is relatively large. With more future information, the container-reads are lower but the computing cost is higher. The optimal configuration of each trace is shown in Table 4.3, the sizes of FAA and chunk cache can be calculated by the number of containers times 4MB.



(a) Speed factor of FSL_1 (b) Computing cost factor of FSL_1 (c) Restore throughput of FSL_1

(d) Speed factor of FSL_2 (e) Computing cost factor of FSL_2 (f) Restore throughput of FSL_2

(g) Speed factor of EMC_1 (h) Computing cost factor of EMC_1 (i) Restore throughput of EMC_1

(j) Speed factor of EMC_2 (k) Computing cost factor of EMC_2 (l) Restore throughput of EMC_2

Figure 4.6: The restore performance results comparison of Container_LRU, Chunk_LRU, FAA, Fix_Opt and ALACC. **Notice** that the speed factor, computing cost factor and restore throughput vary largely in different traces, we use different scales among sub-figures to show the relative improvement or difference of the five designs in the same trace.

Table 4.4: The percentage of memory size occupied by FAA of ALACC in each restore testing case

Version #	0	1	2	3	4	5
FSL_1	50%	38%	39%	38%	40%	38%
FSL_2	67%	67%	64%	69%	64%	57%
EMC_1	26%	24%	14%	17%	16%	16%
EMC_2	7%	8%	8%	8%	8%	7%

4.7.3 Restore Performance Comparison

Using the same restore engine and storage as discovering the Fix_Opt, we evaluate and compare the speed factor, computing cost and restore performance (throughput) of the four traces with 64MB total memory. The evaluation results are shown in Figure 4.6. As indicated in Table 5.1, the CFL of FSL_2 is much lower than the others, which leads to a very close restore performance of all the restore designs. However, when the CFL is high, ALACC is able to adjust the sizes of FAA, chunk cache, and LAW to adapt to the highly fragmented chunk storing, and achieves higher restore performance.

The computing cost varies in different traces and versions. In most cases, the computing cost of FAA is relatively lower than the others, because it avoids the cache insertion, look-up, and eviction operations. As expected, the computing overhead of Chunk_LRU is usually higher than that of Container_LRU due to more management operations for smaller caching granularity. However, in EMC_2, the computing overhead of Container_LRU is much higher than the other four designs. The overhead is caused by the high frequent cache replacement and extremely high cache miss ratio. The cache miss ratio of Container_LRU is about $3X$ higher than that of Chunk_LRU. The time of reading containers from storage to the read-in buffer dominates the restore time. Thus, the speed factor can nearly determine the restore performance. Comparing the speed factor and restore throughput of the same trace, for example, trace FS_1 in Figure 4.6(a) and 4.6(c), the curves of the same cache policy are very similar.

For all 4 traces, the overall average speed factor of ALACC is 83% higher than Container_LRU, 37% higher than FAA, 12% higher than Chunk_LRU and 2% higher than Fix_Opt. The average computing cost of ALACC is 27% higher than that of Container_LRU, 23% higher than FAA, 33% lower than Chunk_LRU and 26% lower than Fix_Opt. The average restore throughput of ALACC is 89% higher than that of

Table 4.5: The average LAW size (# of containers) of ALACC in each restore testing case

Version #	0	1	2	3	4	5
FSL_1	31.2	30.5	31.4	32.2	32.0	30.7
FSL_2	44.6	44.1	40.1	32.3	32.8	36.9
EMC_1	77.1	83.1	88.7	84.2	76.1	82.6
EMC_2	95.3	95.2	95.1	95.3	94.8	95.2

Container_LRU, 38% higher than FAA, 14% higher than Chunk_LRU and 4% higher than Fix_Opt. In our experiments, the speed factor of ALACC is higher than those of Container_LRU, Chunk_LRU, and FAA. More importantly, ALACC achieves at least a similar or better performance as Fix_Opt. By dynamically adjusting the sizes of FAA, chunk cache and LAW, the improvement of the restore throughput is higher than the speed factor. Notice that we need tens of experiments to find out the optimal configurations of Fix_Opt which is almost impossible to carry out in a real-world production scenario.

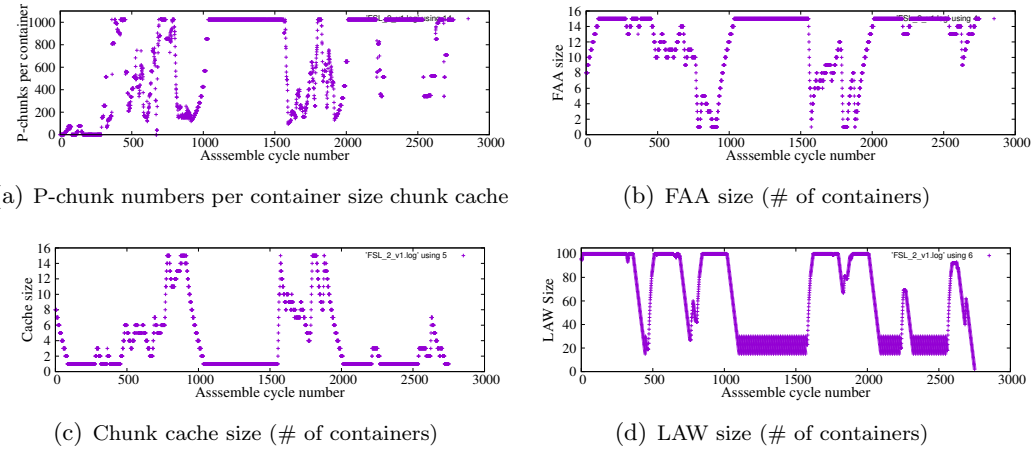
The main goal of ALACC is to make better trade-offs between the number of container-reads and the required computing overhead. The average percentage of the memory space occupied by FAA of ALACC is shown in Table 4.4 and the average LAW size is shown in Table 4.5. The percentage of chunk cache in the memory can be calculated by $(1 - FAA$ percentage). The mean FAA size and LAW size vary largely in different workloads. The restore data with larger data chunk re-use distance usually needs smaller FAA size, larger cache size, and larger LAW size, like in traces FSL_1, FSL_2, and EMC_2. One exception is trace EMC_1. This trace is very special, only about 4% data chunks are duplicated chunks and they are scattered over many containers. The performances of Container_LRU, Chunk_LRU and FAA are thus very close since extra container-reads will always happen when restoring the duplicated chunks. By adaptively extending the LAW to a larger size (about 80 containers and 5 times larger than FAA cover range) and using larger chunk cache space, ALACC successfully identifies the data chunks that will be used in far future and caches them. Therefore, ALACC can outperform others in such an extreme workload. Assuredly, ALACC has the highest computing overhead (about 90% higher than others in average) as shown in Figure 4.6(h).

Comparing the trend of varying FAA and LAW sizes of Fix_Opt (shown in Table 4.3) with that of ALACC (shown in Tables 4.4 and 4.5), we can find that ALACC usually applies smaller LAW and larger cache size than Fix_Opt. Thus, ALACC achieves lower computing cost and improves the restore throughput as shown in Figures 4.6(b), 4.6(e) and 4.6(k). In EMC_2, ALACC has a larger cache size and a larger LAW size than those of Fix_Opt. After we exam the restore log, we find that the *P-chunks* occupied 95% the cache space in more than 90% of the assembling cycles. A very small portion of data chunks is duplicated many times, which can explain why the Chunk_LRU performs close to Fix_Opt. In such an extreme case, ALACC makes the decision to use a larger cache space and a larger LAW size such that it can still adapt to the workload and maintain a high speed factor and high restore throughput as shown in Figures 4.6(j) and 4.6(l). In general, ALACC successfully adapts to the locality changing and delivers high restore throughput for a given workload.

4.7.4 The Adaptive Adjustment in ALACC

To verify the adaptive adjustment process of ALACC, we write the log at the end of each assembling cycle and using *P-chunk* as an example to show the size adjustment of FAA, chunk cache and LAW by ALACC. The log records the *P-chunk* numbers per container size cache , the sizes of FAA, chunk cache, and LAW. We use FSL_2 version 1 as an example and the results are shown in Figure 4.7. The number of *P-chunks* per container size cache is very low at the beginning and varies sharply as assembly cycle increases as shown in Figure 4.7(a). One container size cache can store about 1000 data chunks in average. During the assembling cycle range (1000–1500), most of the chunk cache space is occupied by the *P-chunks* and there are few duplicated data chunks. Thus, ALACC uses a larger FAA and a smaller LAW.

If the number of *P-chunk* is relatively low, more caching space is preferred. For example, in the assembling cycle range (700–900), the number of *P-chunks* is lower than 200 (i.e., more than 80% of the chunk cache space is used for *F-chunks*). As expected, the FAA size drops quickly and the chunk cache size increases sharply and stays at a high level. Meanwhile, since the cache space is increased, the LAW size is also increased to cover larger recipe range and to identify more *F-chunks*. In general, ALACC successfully monitors the workload variation and self-adaptively reacts to the



(a) P-chunk numbers per container size chunk cache

(b) FAA size (# of containers)

(c) Chunk cache size (# of containers)

(d) LAW size (# of containers)

Figure 4.7: The variation of P-chunk numbers per container size chunk cache, FAA size, chunk cache size, and LAW size during the restore of FSL_1 in version 0

number of *P-chunks* variation as expected, and thus, delivers higher restore throughput without manual adjustments.

4.8 Conclusion and Future Work

Improving restore performance of deduplication system is very important. In this research, we studied the effectiveness and the efficiency of different caching mechanisms applied to the restore process. Based on the observations of the caching efficiency experiments, we design an adaptive algorithm called ALACC which is able to adaptively adjust the sizes of the FAA, chunk cache and LAW according to the workload changes. By making better trade-offs between the number of container-reads and computing overhead, ALACC achieves much better restore performance than container-based caching, chunk-based caching and forward assembly. In our experiments, the restore performance of ALACC is slightly better than the best performance of restore engine with all possible configurations of fixed sizes of FAA, chunk cache and LAW. In our future work, duplicated data chunk rewriting will be investigated and integrated with ALACC to further improve the restore performance of data deduplication systems for both primary and secondary storage systems.

Chapter 5

Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance

5.1 Introduction

With the fast development of new eco-systems such as social media, cloud computing, artificial intelligence (AI), and Internet of Things (IoT), the volume of data created increases exponentially. However, the storage density and capacity increase in main storage devices like disk drives (HDD) and solid-state drives (SSD) cannot match with the explosion speed of data creation [147]. Data deduplication is an efficient way to improve the storage space utilization such that the cost of storing data can be reduced. Data deduplication is a widely used technique to reduce the data amount to be transferred or stored in today's computing and communication infrastructure. It has been applied to primary and secondary storage systems, embedded systems and other systems that host and transfer a huge amount of data.

Data deduplication partitions the *byte stream* formed by original data into data chunks (e.g., 4KB size in average). Each chunk is represented by a *chunk ID* which is a

hashed fingerprint of the chunk content. By representing the original data stream with a sequence of the chunk IDs as well as the metadata to find the data chunks (called file recipe) and storing only the unique data chunks (not a duplication of existing data chunks) into the storage system, the amount of data to be stored can be greatly reduced. Since the average size of data chunks is rather small, the deduplication process typically accumulates a number of unique data chunks (say 1000 data chunks) in a container before writing them out together to storage to increase I/O efficiency. Due to the same reason, when reading a data chunk for restore, the whole container is read in since we are expecting many data chunks in the same container will be accessed soon.

The data restore process is to access and write out data chunks based on the order and locations in the recipe to re-create the original byte stream. Since the unique data chunks are writing to storage based on the order of their first appearance in the byte stream, a duplicated data chunk may request to read a container that was written long time ago. The result of deduplication can be measured by the deduplication ratio which is the total amount of data in the original byte stream divided by the total size of the stored unique data chunks. However, a higher deduplication ratio means more container reads have to be issued when restoring the original byte stream and it leads to a lower restore performance.

To improve the restore performance, several techniques have been proposed including caching schemes (e.g., container-based caching [100, 101, 102, 103], chunk-based caching [55, 104, 59], and forward assembly [55, 59]) and duplicated data chunk rewrite schemes [100, 101, 102, 103, 55]. Since the unique data chunks are stored in the same order as they first appeared, unique data chunks in the same container are located closely in the original byte stream. If these chunks in a container will appear in the near future in the byte stream as duplicated data chunks, caching schemes can be very useful to reduce the number of container reads for duplicated data chunks.

When a deduplication system runs for a long time, the data chunks in a segment of the original byte stream can be highly fragmented. That is, the duplicated data chunks in this segment are stored in a large number of containers that have already existed (old containers). When restoring this segment, each read-in container consists of only a few data chunks that are used in this segment. Therefore, the cached data chunks may be evicted before they are used again in the restore process. In this scenario, the number

of container reads cannot be reduced by applying caching schemes only.

In this highly fragmented situation, each container read only contributes to a very small number of data chunks for restore. This type of container reads is inefficient and expensive. To address this issue, rewriting some of the duplicated data chunks in a highly fragmented segment and storing them together with the nearby unique data chunks in the same container can effectively reduce the number of container reads required for restoring these chunks. However, the rewrite decisions made during the deduplication process are not sufficient and efficient due to the limited information of the deduplication process known by the rewrite schemes.

Several data chunk rewriting selection policies are introduced in previous studies. Among these studies, Lillibridge *et al.* [55] proposed a simple but effective data chunk rewriting selection policy, named *capping*. Capping partitions the byte stream into fixed size segments and sorts the old containers by the number of data chunks contributed by the old containers in the segment. Data chunks in the old containers which are ranked out of a pre-defined threshold (i.g, *capping level*) are stored again (called *rewrite*). Therefore, when restoring the data chunks in this segment, the total number of container reads is capped by the capping level plus the number of new containers created with the unique data chunks.

Capping needs a cache space to buffer the segment before a rewrite decision is made. It ensures a higher bound on the number of container reads to restore the data chunks in the segment. However, its design has several limitations: 1) A fixed value of capping level may cause unnecessary rewrites or leaving out the data chunks that should be rewritten (false negative rewrite); 2) Since each segment is a fixed size partition, the number of data chunks of the containers that locate near the cutting boundary is usually counted inaccurately. It causes unnecessary rewrites (false positive rewrite); 3) The caching effect of the restore process has not been considered. Data chunks that can be covered by the cache should not be rewritten; 4) The deduplication ratio cannot be guaranteed with capping.

In this research, we explore ways of further improving the restore performance while maintaining low resource consumption during the deduplication process. We first propose a flexible container referenced count based scheme which is an improvement of the capping scheme. This approach can achieve a good tradeoff between the number of

container reads and the deduplication ratio. Second, we propose a new rewrite scheme called Sliding Look-Back Window Rewrite to more efficiently decide the data chunks to be rewritten. A sliding look-back window can remedy the false positive issue caused by a fixed size cut. Based on the sliding look-back window design, a container read efficiency based rewrite is proposed which takes the caching effect into consideration.

To fairly and comprehensively evaluate our rewriting designs, we implemented a system with both deduplication and restore engines of normal deduplication, capping scheme, the two schemes we proposed, Forward Assembly (FAA) [55], and a chunk-based caching scheme called ALACC [59]. The performance of different combinations of deduplication and restore engines are compared and evaluated. According to our evaluation by using real world deduplication traces, with the same deduplication ratio and the same restore engine, our proposed sliding look-back window based design always achieves the best restore performance. We use *speed factor* (MB/container-read), which is defined as the mean size data being restored (MB) per container read [55], to indicate the amount of data that can be restored by one container read in average. A higher speed factor indicates a higher restore performance. Our design can improve the speed factor up to 97% compared with that of normal deduplication and it can improve the speed factor up to 41% compared with capping.

The rest of the chapter is presented as follows. Section 2 reviews the background of data deduplication and the related work of caching and rewrite schemes. Section 3 describes a flexible container referenced count based rewrite scheme which is improved from capping. To further reduce the number of container reads, a sliding look-back window based rewrite scheme is proposed and presented in Section 4. Based on the sliding look-back window, the rewrite candidate selecting policy is discussed in Section 5. We present the evaluation results and analysis in Section 6. Finally, we conclude our work and discuss the future work in Section 7.

5.2 Background and Related Work

In this section, we first briefly describe the deduplication and restore processes. Then, the related studies of improving the restore performance are presented and discussed.

5.2.1 Deduplication and Restore Process

Data deduplication is widely used in secondary storage systems such as archiving and backup systems to improve the storage space utilization [36, 88, 35, 89, 90, 91, 142, 143]. Recently, data deduplication is also deployed in primary storage systems like SSD arrays to make better trade-offs between cost and performance [92, 93, 94, 95, 96, 86, 97, 98]. After the original data is deduplicated, only the unique data chunks and the file recipe are stored. When the original data is requested, the recipe is used to read out the corresponding data chunks for assembling the data back. That is, based on the recipe to access each data chunk and restore these data chunks back to the original byte stream. From the beginning of the recipe, the restore engine uses the data chunk metadata to access the corresponding data chunks one by one and assembles the data chunks in the memory buffer. Once the engine accumulates a buffer worth of data chunks, it will write out the restored data in the buffer to persistent storage. To get a data chunk from storage to memory in an efficient way, the restore engine will access the entire container holding the data chunk. The accessed data chunks can be temporarily held in memory by a specifically designed cache to reduce the future container reads.

In the worst case, to assemble N data chunks, we may need N container reads. A straightforward way to reduce the number of container reads is to cache some of the containers or data chunks. Since some data chunks will be used very shortly after they are read into memory, these cached chunks can be directly copied from the cache to the assembling buffer which can reduce the number of container reads. Even with caching schemes like chunk-based caching and forward assembly [55], an accessed data chunk may trigger a container read. This is especially true for accessing a duplicated data chunk since this data chunk was stored in a container created in a distance from the current location in the recipe. Intuitively, the higher a deduplication ratio (i.e., more duplicated data chunks) is, the more container reads are required to restore [101]. Another way to reduce the number of container reads is to store (rewrite) some of the duplicated data chunks together with the unique chunks in the same container during the deduplication process. The decision of rewriting a duplicated data chunk has to be decided during the deduplication process instead of being done at restore process like caching. The duplicated chunks and unique chunks will be read out together in the same container and thus the needs of accessing these duplicated chunks from other containers

are avoided. However, this approach reduces the effectiveness of data deduplication (i.e., reduces the deduplication ratio).

5.2.2 Related Work on Restore Performance Improvement

We first review the different caching policies such as container-based caching, chunk-based caching, and forward assembly. Then, we will focus on the studies of storing duplicated chunks to improve the restore performance.

Different caching policies are studied in [100, 101, 102, 55, 104, 59, 141]. To improve the cache hit ratio, Kaczmarczyk *et al.* [100], Nam *et al.* [101, 102], and Park *et al.* [141] used container-based caching. Container-based caching schemes can achieve Belady's optimal replacement policy [103]. To achieve a higher cache hit ratio, some studies directly cache data chunks [55, 104]. For container-based and chunk-based caching, the former has lower cache management overhead (e.g., fewer memory copies), while the latter has a higher cache hit ratio. Therefore, caching chunks is preferred in the restore process, especially when the cache space is limited. Lillibridge *et al.* proposed the forward assembly scheme, which reads ahead of the recipe and pre-fetches some chunks into a Forward Assembly Area (FAA) [55]. This scheme ensures that one container will be only read out once when restoring the current FAA. The management overhead of FAA is much lower than that of the chunk-based caching, but the performance of these two schemes depends on the workloads. Therefore, Cao *et al.* proposed a combined FAA and chunk-based caching called Adaptive Look-Ahead Chunk Caching (ALACC) which can potentially adapt to various workloads and achieve a better restore performance [59].

Nam *et al.* [101, 102] first introduced the Chunk Fragmentation Level (CFL) to estimate degraded read performance of deduplication storage. CFL is defined as a ratio of the optimal number of containers with respect to the real number of containers required to store all the unique and duplicated chunks of a backup data stream. When the current CFL becomes worse than a predefined threshold, data chunks will be rewritten. Kaczmarczyk *et al.* [100, 148] utilized stream context and disk context of a duplicate block to rewrite highly-fragmented duplicates. Data chunk whose stream context in the current backup is significantly different from its disk context will be rewritten. Fu *et al.* [103, 149] proposed a History-Aware Rewriting algorithm (HAR) which accurately

identifies and rewrites sparse containers according to the historical information of the previous backup. The container capping was proposed by Lillibridge *et al.* [55] which uses a fixed-size segment to identify the data chunks to be rewritten.

Since each container read involves a large fixed number of data chunks, Tan *et al.* [150] proposed a Fine-Grained defragmentation approach (FGDefrag) that uses variable-size data groups to more accurately identify and effectively remove fragmented data. FGDefrag rewrites the fragmental chunks and the new unique chunks of each segment into a single group. After some of the data chunks are rewritten, the same data chunks can appear in different containers, how to choose one container to be referenced is challenging. Wu *et al.* [151, 152] proposed a cost-efficient rewriting scheme (SMR). SMR first formulates the defragmentation as an optimization problem of selecting suitable containers and then builds a sub-modular maximization model to address this problem by selecting containers with more distinct referenced data chunks.

The caching schemes can be effective if recently accessed data chunks (unique or duplicated) will be needed again in the near future during the restore process. If duplicated data chunks are stored in the containers created and scattered over different time, they will not be effective. Only rewrite some of the duplicated data chunks together with unique chunks can reduce the container reads. However, it is difficult to make a decision which duplicate chunks need to be rewritten at the minimum cost of the deduplication ratio. This is the focus of our study in this research.

5.3 Flexible Container Referenced Count based Design

In this section, we first discuss the strengths and limitations of the capping design. Then, we propose a flexible container referenced count based scheme improved from capping which achieves a good tradeoff between the number of container reads and the deduplication ratio.

5.3.1 Capping and Its Limitations

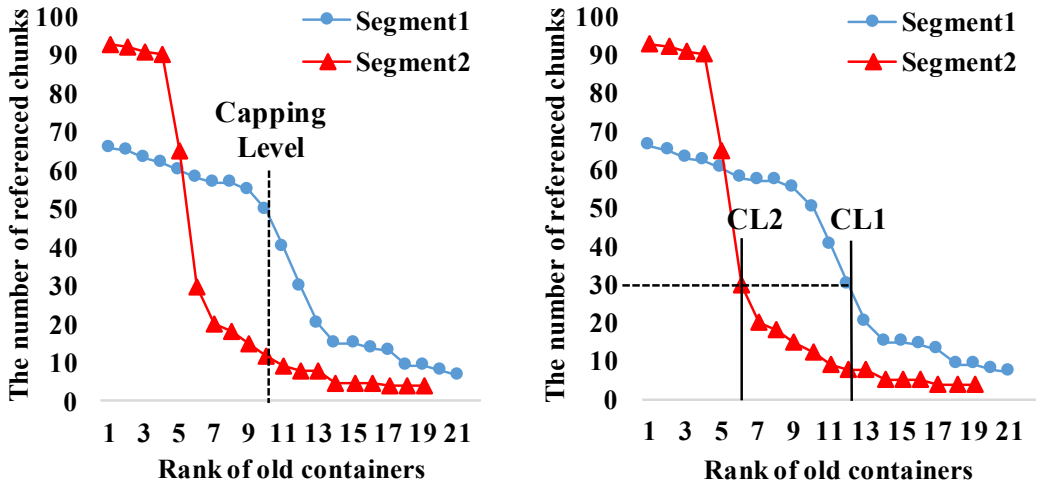
The container capping is proposed by Lillibridge *et al.* [55]. The original data stream is partitioned into fixed size segments and each segment is N container size (e.g., $N \cdot 4MB$ size). The number of data chunks (duplicates are counted) in the segment that belonging

to an old container is counted, called *CoNtainer Referenced Count* (CNRC). These old containers containing at least one duplicated data chunk in the segment are sorted with their CNRCs in descending order. A fixed threshold T called *capping level* is used to decide whether the duplicated data chunks of an old container will be rewritten or not. If the container is ranked lower than T, the duplicated data chunks in this container are written together with unique data chunks into the active container. In this way, when restoring this segment in the future, there will be at most $T + C$ container reads, where C is the number of new containers that are generated for storing unique data chunks when deduplicating this segment. Thus, the number of container reads of the segment is capped by $T + C$. However, the deduplication ratio will be lowered too and it is hard to predict the deduplication ratio reduction.

Capping uses the same fixed threshold to control the capping level for all the segments such that a higher bound on the number of container reads can be obtained. However, the deduplication ratio is not considered as an optimization objective in the capping scheme. We believe that with some changes to capping, a better tradeoff between the number of container reads and the data deduplication ratio can be achieved. Since the distribution of CNRCs of involved containers varies from segment to segment, some segments can keep fewer container reads than the targeted capping level while other segments may have to go beyond the targeted capping level if we wish to bound the reduction of deduplication ratio.

For example, in a segment, after we sort the old containers according to their CNRCs in descending order, we can plot out the CNRC distribution of these containers as shown in Figure 5.1(a). The X-axis is the rank of containers and Y-axis is the number of referenced data chunks of an involved container. Different segments will have different distribution shapes. Consider Segment 1 and Segment 2 in Figure 5.1(a). Capping level is a fixed cutting point at X-axis which is 10 and all duplicated data chunks in the containers to the right of the capping level partition have to be rewritten.

In this example, the number of container reads of these two segments are capped by 20 plus the number of newly generated containers. For Segment 1, containers ranked from 10 to 12 have a relatively large number of referenced data chunks. Rewrite these containers will cause more reduction of deduplication ratio. The ideal capping level for Segment 1 should be either 12 or 13 according to the distribution. For Segment



(a) The CNRCs distribution and the fixed capping level

(b) The CNRCs distribution and the specified capping level for different containers

Figure 5.1: An example of the relationship between the CNRCs distribution and the capping level

2, since data chunks are more concentrated to the higher ranking containers, rewriting the containers ranked beyond container 6 or 7 instead of container 10 will reduce more container reads while the increased amount of duplicated data chunks being rewritten is limited. Therefore, as shown in Figure 5.1(b), if we use 12 as the new capping level for Segments 1 (CL1 in the figure) and use 6 as the new capping level for segment 2 (CL2 in the figure), the number of container reads will be 1 fewer and the number of duplicated data chunks being rewritten will be even lower.

5.3.2 Flexible Container Reference Count Scheme

To address the aforementioned limitations of using a fixed capping level, we proposed the Flexible Container Referenced Count based scheme (FCRC), which is an improvement of capping based on the CNRC instead of the capping level. If we set a value as the CNRC threshold, data chunks in one old container that has CNRC lower than the threshold are rewritten. In this way, different segments will have different actual capping levels which are decided by the threshold and the distribution of CNRCs of these segments. The CNRC threshold T_{cnrc} can be estimated by the targeted capping level. That is, the

total number of duplicated data chunks (duplicates are counted) in a segment divided by the targeted capping level. Statistically, if each read-in container in a segment can contribute more than T_{cnrc} number of duplicated data chunks, the total number of container reads for the segment will be bounded by the targeted capping level. Thus, by rewriting duplicated data chunks in a container with a referenced count lower than T_{cnrc} can achieve a similar number of container reads as using the targeted capping level.

Although using a fixed value of T_{cnrc} as a threshold can make a tradeoff between the number of container reads and the deduplication ratio, it cannot guarantee either an upper bound on the number of container reads like capping or a lower bound on the deduplication ratio. To solve this issue, we use a targeted deduplication ratio reduction limit $x\%$ and the targeted number of container reads Cap of a segment to generate these two bounds for T_{cnrc}. In the following, we will discuss how these two bounds are decided.

Bound for Deduplication Ratio Reduction We first calculate the targeted number of data chunks that can be rewritten (N_{rw}) in a segment according to the targeted deduplication ratio reduction limit $x\%$. Let us consider backup as an example. Suppose in the current backup version, the total number of data chunks is N_{dc} and the targeted deduplication ratio reduction limit is x% (i.e., after rewrite, deduplication ratio is at most x% lower than that of no rewrite case). For one backup version, we use the number of unique data chunks generated in the previous deduplicated version as the estimation of the current version, which is N_{unique}. If we do not rewrite duplicated data chunks, the deduplication ratio is $DR = \frac{N_{dc}}{N_{unique}}$. To have at most x% of deduplication ratio reduction, we need to rewrite N_{rw} data chunks. We can calculate the new deduplication ratio after rewrites by $DR \cdot (1 - x\%) = \frac{N_{dc}}{N_{unique}+N_{rw}}$. Finally, we have $N_{rw} = \frac{N_{unique} \cdot x\%}{1-x\%}$. By dividing N_{rw} by the total number of segments of this backup version, we can calculate the average data chunks that can be rewritten in each segment, which is H_{rw}. For other applications, we can run deduplication for a short period of time. Then , we use the current total number of data chunks as N_{dc} and the current total number of unique chunks as N_{unique} to estimate H_{rw}.

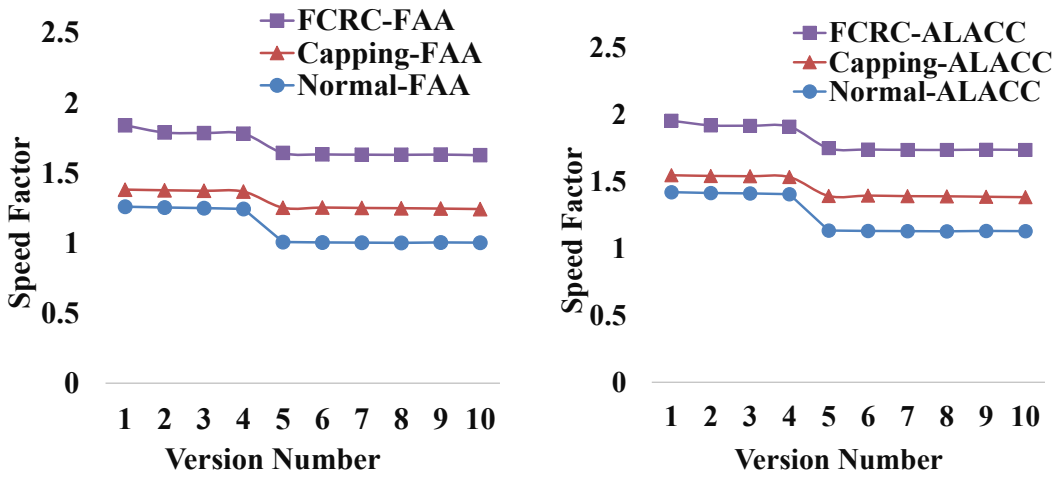
Since the actual rewrite number can be smaller than H_{rw} in some segments, we can accumulate and distribute the saving as credits to the rest of the segments such that

some of the segments can rewrite more than H_{rw} duplicated data chunks. However, the overall deduplication reduction limit x% is still satisfied. For the i_{th} segment, suppose the accumulated actual rewrites before i_{th} segment is N_{rw}^{i-1}. We can rewrite at most $H_{rw} \cdot i - N_{rw}^{i-1}$ duplicated data chunks in the i_{th} segment. If $N_{rw}^{i-1} = H_{rw} \cdot (i-1)$, we still use H_{rw} as the rewrite limit for the i_{th} segment. This means rewriting more than the limit is not allowed in any segment. In this way, we will get the referenced count bound of the i_{th} segment RC_{rw}^{i} by adding the CNRCs of the old containers from low to high until the accumulated number reaches the rewrite limit.

Bound for Container Reads We use a cap on the number of container reads Cap and the CNRC distribution to get the bound on the number of container reads of the i_{th} segment which is RC_{reads}^{i}. Suppose the accumulated number of old containers used before the i_{th} segment is N_{reads}^{i-1}. For the i_{th} Segment, we can tolerate at most $Cap \cdot i - N_{reads}^{i-1}$ containers to be used. Counting from the high ranked container to low, we will find the RC_{reads}^{i} of the container at the bound.

If $RC_{rw}^{i} < RC_{reads}^{i}$ in the i_{th} segment, we are unable to satisfy both of the requirements. We will try to satisfy the deduplication ratio reduction limit first. Therefore, the threshold T_{cnrc} we used in this segment is RC_{rw}^{i}. The bound on the number of container reads will be violated and the credits of the container reads of this segment will be negative. If $RC_{rw}^{i} \geq RC_{reads}^{i}$, the threshold T_{cnrc} is able to be adjusted between RC_{rw}^{i} and RC_{reads}^{i}. If T_{cnrc} of the previous segment is in between, we directly use the same T_{cnrc}. Otherwise, we will use $\frac{RC_{rw}^{i}+RC_{reads}^{i}}{2}$ as the new T_{cnrc} which can accumulate the credit for both rewrite number and the number of container reads.

The performance comparison between the FCRC scheme and the capping scheme is shown in Figure 5.2. We use the trace MAC2 shown in Table 5.1 which is from FSL [145, 153, 154] as an example and the details about the trace are described in Section 6.1. We use two different restore caching schemes which are FAA (the FAA size is 10 containers) and ALACC of the fixed algorithm (4 containers for FAA and 4 containers for chunk cache). To comprehensively compare the performance, we implemented the normal deduplication which has no rewrite (Normal), the capping scheme (Capping), and the FCRC. Capping and FCRC are configured with the same segment size (5 containers suggested by [55]). By configuring the targeted capping level and deduplication reduction ratio, both Capping and FCRC achieve the same deduplication ratio (12.8 in



(a) The performance comparison with FAA as the caching scheme

(b) The performance comparison with ALACC as the caching scheme

Figure 5.2: The performance comparison between Regular, Capping, and FCRC

the last version).

As shown in Figure 5.2, the FCRC scheme can achieve about 33% of higher speed factor than that of Capping on average when the same restore caching scheme is used. Specifically, if we use FAA as the restore caching scheme, FCRC can achieve about 31% of higher speed factor than that of Capping and 55% higher than that of Normal. If we use ALACC as the restore caching scheme, the speed factor is improved about 10% than that of FAA in average and FCRC can achieve about 47% of higher speed factor than that of Normal. In general, by using a flexible referenced count based scheme, the container reads are effectively reduced compared to the Capping scheme with a fixed value of capping level when the same deduplication reduction ratio is achieved.

5.4 Sliding Look-Back Window

Although our proposed FCRC scheme can address the tradeoff between the number of container reads and the deduplication ratio, using a fixed size segment partition causes another problem. In this section, we will first discuss the problem and then a new rewrite framework called Sliding Look-Back Window (LBW) is presented in detail.

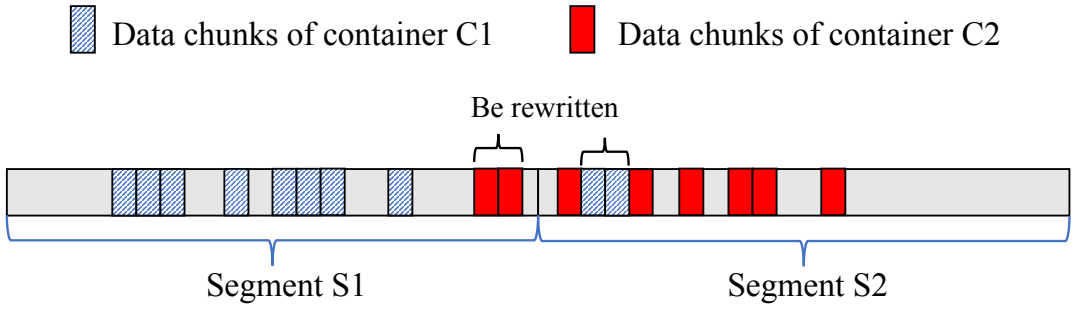


Figure 5.3: An example to show the fixed size segment partition issue in capping

5.4.1 Issues of Fixed Size Segment Partition

In either capping or FCRC designs, the byte stream is partitioned into fixed size segments and the rewrite decision is made based on the CNRC statistics of a segment. The container and data chunk information based on a fixed size segment may not take into consideration of the information in the past and future segments. On one hand, data chunks at the front of a segment are evaluated with more information of near future while the data chunk referencing situation and rewrite decisions made in the previous segment cannot be used by the data chunks at the front. On the other hand, data chunks at the end of the segment can be evaluated with the information from the near past while there is less future information can be identified. The data chunks at the end of the segment have higher probability to be rewritten if the majority data chunks of their containers are in the subsequent few segments. As a result, the rewrite decisions made with the statistics only from the current segment may not be accurate.

As shown in Figure 5.3, there are two consecutive segments $S1$ and $S2$. The majority of duplicated data chunks belonging to container $C1$ appear at the front of $S1$ with blue color and the majority data chunks of $C2$ are at the head of $S2$ with red color. Due to the segment cut point, a few duplicated chunks of container $C1$ are at the head of $S2$ and a few duplicated chunks of container $C2$ are at the end of $S1$. According to the capping design, the number of data chunks of $C1$ in $S1$ and that of $C2$ in $S2$ are ranked higher than the capping level. These chunks will not be rewritten. However, the ranking of container $C1$ in $S2$ and container $C2$ in $S1$ are out of the capping level. Since we do not know the past information about $C1$ in $S1$ when deduplicating $S2$ and

the future information about $C2$ in $S2$ when deduplicating $S1$, these chunks (i.e., a few data chunks from $C2$ in $S1$ and a few data chunks from $C1$ in $S2$) will be rewritten. When restoring $S1$, $C1$ and $C2$ will be read in the cache. When restoring $S2$, $C2$ is already in the cache and the container read will not be triggered. Therefore, rewriting the data chunks of $C2$ appearing in $S1$ and data chunks of $C1$ in $S2$ is wasted.

To more precisely evaluate each data chunk in a segment, a "sliding" window design can potentially solve this issue. Assuming a sliding window consists of N containers. In the deduplication process, a new data chunk is identified and this data chunk will be a part of the most recent container in the sliding window. Each duplicated data chunk will be evaluated with the past information for rewrite when it is in the sliding window. When the most recent container is filled up, the window will move forward to include a new container (i.e., the window is moved with one container size data chunks). In the meantime, the last (oldest) container will be removed from the window. Before a container is moved out the window, the decision to rewrite the duplicated data chunks in this container has to be made. In fact, a rewrite decision can be made earlier for some data chunks to reduce the memory utilization. We will describe this process in details below and present how a rewrite decision for a duplicated data chunk is made in Section 5.

5.4.2 Sliding Look-Back Window Assisted Rewrite

Motivated by the discussion in Section 5.4.1, we designed and implemented the sliding look-back window assisted rewrite scheme. The overall architecture is shown in Figure 5.4. A fixed size window of N container size is used to cover a range of a byte stream that has been deduplicated. The window moves forward to process the newly identified data chunks while these data chunks are evaluated backward through the window to make the initial rewrite decision. This is why we call it a *Look-Back Window* (LBW). The newly identified data chunk is located in the most recent container of the LBW. Unique data chunks will be buffered first in the active container. Once the active container is full, it will be written to storage. The LBW acts as a recipe cache which maintains mostly the data chunk metadata entries in the recipe of N containers in the order of the byte stream. Each metadata entry consists of the information of the data chunk as in the recipe including chunk ID, chunk size, offset in the byte stream,

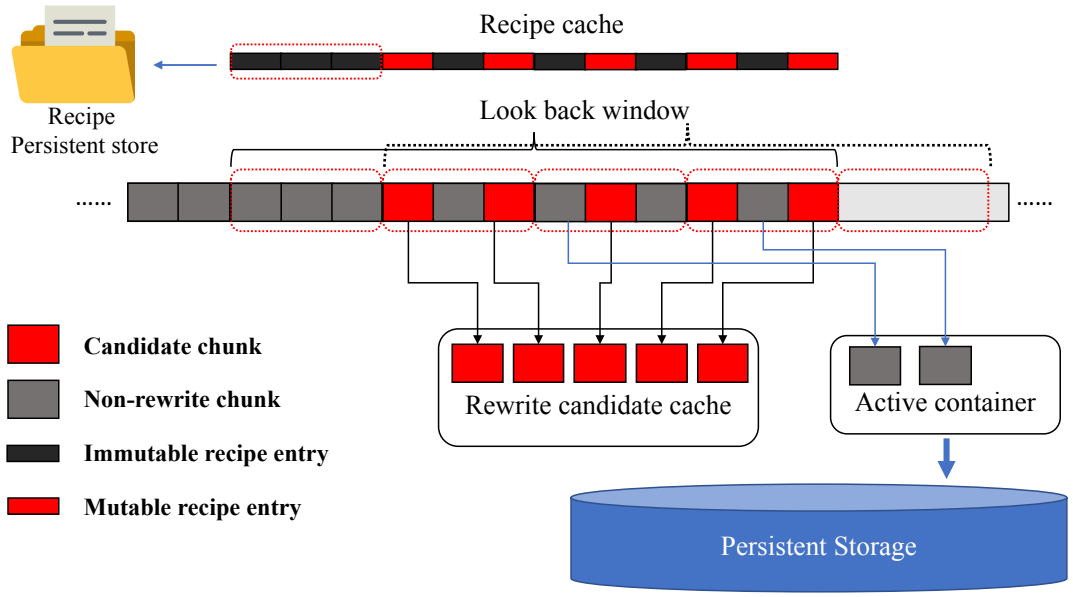


Figure 5.4: The overview of the sliding look-back window assisted rewrite design

container ID/address, and the offset in the container. Besides, the LBW maintains the information of referenced containers (i.e., containers holding duplicated data chunks) covered by the LBW. This information will be used to decide which duplicated data chunks are rewritten.

As shown in Figure 5.4, the window is 4 container size. The newly identified data chunks are part of the most recent container in the window. If one data chunk is unique, it will be put into the active container and its corresponding metadata entry is stored in the recipe cache of the LBW. If it is a duplicated data chunk, its metadata information of the original data chunk is also included in the current metadata entry in the recipe cache. If a duplicated data chunk is decided not to be rewritten (unique chunk or duplicated chunk satisfying the non-rewrite condition described later), it will be marked as a *non-rewrite chunk* and not be added to the rewrite candidate cache. If the rewrite decision cannot be given right after a duplicated data chunk is identified, this chunk will be added to the rewrite candidate cache as one *candidate chunk*. When the most recent container in the LBW fills up with data chunks (either unique or duplicated), the oldest (least recent) container in the LBW is moved out and its corresponding metadata entries are written to the recipe on storage. Before the oldest container is moved out,

the rewrite decisions of all candidate chunks in that container have to be made. Then, the LBW will move forward to cover a new container size of data chunks.

The effect of delaying rewrite decision using a look-back scheme is as follows. When a data chunk is identified as a candidate chunk, we will have the information of all data chunks in the several previous (N-1) containers including their rewrite decisions. If we cannot make a decision for this candidate chunk immediately due to the limited information, we can wait to make a decision when this chunk is moved out of the window. Until then, we will know the data chunks subsequent to this data chunks in the rest of N-1 containers. Therefore, we believe a more accurate decision can be made.

To implement such a look back mechanism, some data needs to be temporarily cached in memory. In our design, we reserve three types of data in memory as shown in Figure 5.4. First, as mentioned before, the LBW maintains a recipe cache. Second, similar to the most container based deduplication engines, we maintain an active container as chunk buffer in memory to store the unique or rewritten data chunks. The buffer is one container size. Third, we maintain a rewrite candidate cache in memory to temporally cache the candidate chunks. The same chunk might appear in different positions in the window, but only one copy is cached in the rewrite candidate cache. As the window moves, data chunks in the rewrite candidate cache will be gradually evicted according to the rewrite selection policy and these chunks become non-rewrite chunks. Finally, before the last container is moved out, the remaining candidate chunks in this container are rewritten to the active container. The metadata entry of a candidate chunk is mutable while the metadata entry of a non-rewrite chunk is immutable. The details of the rewrite selection policy will be presented in Section 5.5.

Let us consider a simple example as shown in Figure 5.5. Suppose we use a fixed value of reference count as the threshold and set the value as 4. That is, once an old container has more than 4 duplicated data chunks in the window, the duplicated data chunks in this container will not be rewritten. When the LBW is at Position 1, the number of data chunks from Containers 1, 2, and 3 are 8, 2, and 2 respectively. The newly identified data chunks from Containers 2 and 3 cannot satisfy the non-rewrite condition, we add these data chunks to the rewrite candidate cache. When the LBW is moved to Position 2, the number of data chunks from Container 2 is already 5 which satisfy the non-rewrite condition. The data chunks from Container 2 will not be rewritten. Therefore, the data

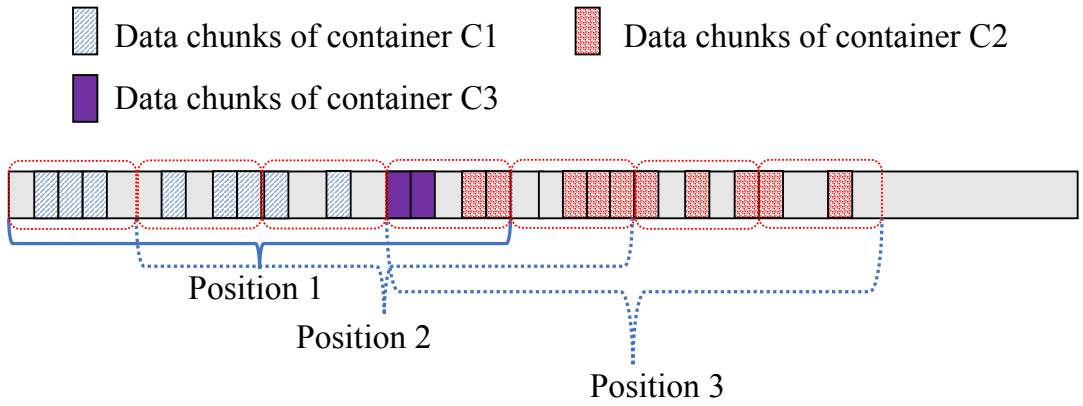


Figure 5.5: An example of the sliding look-window assisted rewrite scheme

chunks of Container 2 in the rewrite candidate cache will be dropped. For the data chunks from Container 3, we still need to delay the decision. When the LBW is moved to Position 3 and the data chunks from Container 3 are already at the end of the LBW. At this moment, the number of data chunks from Container 3 is still lower than the threshold. Thus, the two data chunks will be rewritten to the active container. At the same time, the corresponding metadata entries in the recipe cache are updated with the active container information.

5.5 Rewrite Selection Policy for LBW

In this section, we present the details of the rewrite selection policy for LBW. We first discuss the two criteria which are closely related to our proposed policy. Then, the details of the policy are described.

5.5.1 Two Criteria

For data chunks in the LBW, we base on the following two criteria to design and adjust the rewrite selection policy:

Cache-effective Range In most data deduplication systems, caching is used to improve the restore performance. Different caching policies (e.g., chunk-based caching, container-based caching, or FAA) have different eviction priorities. However, in a very short period of time, the data chunks of the read-in container will not be evicted. In

other words, if a container is just read into the memory, the data chunks of this container will be cached for at least next S container size restore, we consider it as a cache-effective range. After restoring S containers, these cached data chunks will be gradually evicted. For FAA, S is the size of FAA in terms of the number of containers. For chunk-based caching, S can be greater than the size of the caching space. However, for the container-based caching, S can be much smaller than the cached size. Note that S will also be affected by the workload. Although it may be hard to precisely know when a data chunk will be evicted, we can still estimate a lower bound of the cache effective range of each container which is used as S.

Importantly, the size of the LBW is related to the cache-effective range S. On one hand, if the size of the LBW is much smaller than S, some data chunks identified to be rewritten have a high probability to be in the cache, which can not be identified from a small size LBW. It will cause unnecessary deduplication ratio reduction. On the other hand, if LBW is much larger than S, the data chunk out of the cache-effective range cannot be quickly decided. Maintaining a huge amount of metadata in memory will increase the computing overhead of identifying the data chunks to be rewritten, which will slow down the deduplication performance. Therefore, maintaining an LBW with the size compatible with the cache-effective range is a good tradeoff.

Container Read Efficiency A container read efficiency can be evaluated by the number of data chunks being used in a short duration after it is read-in and the concentration level of these chunks in a given restore range. Please note that at different moments of the restore, the same old container may have a different read efficiency. If one container read can restore more data chunks, this container read is more efficient. With the same number of data chunks being restored from one old container, if these data chunks are more concentrated in a small range of the byte stream, the cache space can be more efficiently used such that container reads can be potentially reduced. In this case, the container read has higher efficiency. Please note in the restore process, the information of future data chunks to be accessed has already been recorded in the recipe and a look-ahead window can be used to identify them [55, 59]. Therefore, caching schemes in the restore process can be more accurate and efficient than the typical application memory caching schemes.

To quantitatively define a container read efficiency, we use two measurements of the

old container in an LBW: *container referenced count* and *referenced chunk closeness*. The container referenced count (CNRC) has been used in FCRC to decide the rewrite candidates in segments. Similarly, we define the container referenced count of an old container as the number of times that data chunks in this container appear in the LBW (duplications are counted) in a moment, which is $CNRC_{lbw}$. $CNRC_{lbw}$ of an old container can change in each movement of the LBW as data chunks are added in and moved out. In fact, the nature of capping is to rank the $CNRC_{lbw}$ of the old containers appearing in a segment and to rewrite the data chunks that belong to the old containers with low CNRC. The FCRC scheme rewrites the containers with CNRC lower than a certain value (the CNRC threshold). However, both schemes are based on a fixed size segment partition.

We define the referenced chunk closeness L_c as: the average distance (measured by the number of data chunks) between the chunk that will potentially trigger a container read and the rest of the data chunks of this container in the same LBW divided by the size of LBW (the total number of chunks in the LBW), where $L_c < 1$. Note that if one data chunk appears in multiple positions, only the first one is counted. Smaller L_c means the data chunks are closer.

5.5.2 Rewrite Selection Policy for LBW

The rewrite selection policy is designed based on the information covered by the LBW. Suppose the LBW size is S_{LBW} which is indicated by the number of containers. When the LBW moves forward for one container size, a container size of data chunks (added container) will be added to the head of the LBW and one container size of data chunks (evicted container) will be removed from the LBW. There will be five steps: 1) process the added container, classify the data chunks into three categories: unique chunks, non-rewrite chunks (duplicated data chunks that will not be rewritten), and candidate chunks (duplicated data chunks that might be rewritten); 2) update the metadata entries of data chunks in the added container and add the identified candidate chunks to the rewrite candidate cache; 3) recalculate the $CNRC_{lbw}$ of old containers that are referenced by all the candidate chunks and reclassify them to the non-rewrite chunks according to the updated $CNRC_{lbw}$; 4) rewrite the remaining candidate chunks in the evicted container to the active container and update their metadata entries; 5) adjust the CNRC threshold

T_{dc} which is used to decide the data chunks once the LBW moves S_{LBW} containers. The details of each steps are explained in the following.

In **Step 1**, by searching the indexing table, the unique data chunks and duplicated data chunks are identified. Unique data chunks are written to the active container. Next, each duplicated data chunk is searched backward in the LBW. If there are already some data chunks from the same old container appearing ahead of it and these chunks are non-rewrite chunks, this chunk is directly marked as a non-rewrite chunk. Otherwise, the duplicated chunk becomes a candidate chunk. For a candidate chunk, if it is the first chunk of its container appearing in the most current S containers, it means this container has not been used at least S containers range. This chunk will potentially trigger one container read and thus we call it the leading chunk of this container. It is also possible that there will be few or even no other data chunks from the same container appearing after the leading chunk in a large range. Rewrite data chunks from this container will be a better choice.

In **Step 2**, the corresponding metadata entires of data chunks in the added container are updated with the metadata information of their chunks and containers. Only one copy of the candidate chunks is added to the rewrite candidate cache.

In **Step 3**, we follow the definition of $CNRC_{lbw}$ described in Section 5.5.1 to calculate the latest $CNRC_{lbw}$ of each involved old containers referenced by the candidate chunks. The candidate chunks from the old containers whose $CNRC_{lbw}$ are higher than the current T_{dc} are removed from the rewrite candidate cache. They become the non-rewrite chunks and their metadata entries in the recipe cache still reference to the old containers.

In **Step 4**, the remaining candidate chunks in the evicted container are rewritten to the active container. Their metadata entires in the recipe cache are updated to reference the active container. Also, if a leading chunk is rewritten, which means the $CNRC_{lbw}$ of its container is still lower than T_{dc}, other data chunks from the same container appearing in the current LBW will be rewritten too. Note that the same data chunk can appear in multiple old containers after being rewritten. One of the old containers should to be referenced in the metadata entry. There are some studies that optimize the selection of container, for example, an approximately optimal solution is proposed by [151, 152]. Here, we use the greedy algorithm which selects the container that has most chunks in

the current LBW. Other algorithms can be applied to our proposed policy easily.

In **Step 5**, T_{dc} is adjusted during the deduplication process to make better tradeoffs between deduplication ratio reduction and container reads. We use the similar algorithm as FCRC to bound the container referenced chunk threshold T_{dc}. Every S_{lbw} size movement is tread as one LBW moving cycle and the T_{dc} is adjusted at the end of each cycle. Similar to the FCRC in Section 5.3.2, at the end of each cycle, we use the accumulated credit of the container reads and the number of rewritten data chunks to get the two bounds of the CNRC in this LBW. If the adjustment range does not exist, we use the bound generated by the deduplication ratio reduction limit as the new T_{dc}. If the adjustment can be made, we will first generate an adjustment *starting value* of T_{dc}. Then, the new T_{dc} is adjusted based on the starting value and the variation of the referenced chunk closeness L_c. If the adjustment range exists and T_{dc} of the last cycle is in the range, we use the previous threshold as the starting value. If the T_{dc} of the last cycle is already out of the adjustment range, we use the middle of the range as the starting value. Next, according to the variation of L_c in the previous cycle, we increase or decrease the starting value value. If the average L_c of old containers in the previous cycle increases (data chunks are more scattered), we increase the starting value by 1 as the new T_{dc}. Otherwise, we decrease the starting value by 1 as the new T_{dc}.

5.6 Performance Evaluation

To comprehensively evaluate our design, we implemented four deduplicate engines including normal deduplication with no rewrite (Normal), capping scheme (Capping), flexible container referenced count based scheme (FCRC), and Sliding Look-back Window scheme (LBW). For the restore engine, we implemented two state-of-the-art caching designs which are Forward Assembly Area (FAA) [55] and Adaptive Look-Ahead window Chunk based Caching (ALACC) [59]. Since any restore experiments will include one deduplication engine and one restore engine, there are 8 combinations.

5.6.1 Experimental Setup and Data Sets

The prototype is deployed on a Dell PowerEdge R430 server with a Seagate ST1000NM0033-9ZM173 SATA hard disk of 1TB capacity as the storage. The server has a 2.40GHz

Table 5.1: Characteristics of datasets

Dataset	MAC1	MAC2	MAC3	FSL1	FSL2	FSL3
TS(TB)[1]	2.04	1.97	1.97	2.78	2.93	0.63
US(GB)[2]	121.8	134.5	142.3	183.7	202	71
ASR(%)[3]	45.99	46.50	46.80	42.49	39.96	33.60
SV(%)[4]	99.28	96.60	95.48	97.87	98.64	93.00
IV[5]	5	20	60	5	20	60

[1] TS stands for the total size of the trace.
[2] US stands for the unique size of the trace.
[3] ASR stands for the average self-referenced chunk ratio in each version.
[4] SV stands for the average similarity between each version.
[5] IV stands for the interval(days-based) between each version.

Intel Xeon with 24 cores and 64GB of memory. In our experiments, the container size is set as 4MB. To fairly compare the performance of these four deduplication engines, we try to use the same amount memory in each engine. That is, for Capping and FCRC, the size of each segment is fixed with 5 containers. For LBW, the total size of the recipe cache and the rewrite candidate cache size are also 5 containers. For FAA, the look-ahead window size is 8 containers and the forward assembly area size is also 8 containers. For ALACC, it has 4 container size FAA, 4 container size chunk cache, 8 container size look-ahead window.

In the experiments, we use six deduplication traces from FSL [145, 153, 154] as shown in Table 5.1. MAC1, MAC2, and MAC3 are three different backup traces from MacOS. FSL1, FSL2, and FSL3 are from FSL /home directory snapshots from the year 2014 to 2015. Each trace has 10 full backup snapshots and we choose the version with the 4KB average chunk size. As shown in SV (average similarity between versions) of Table 5.1, each trace has more than 93% of the data chunks appeared in the previous versions. It indicates that most data chunks are duplicated chunks and they are fragmented. The time interval (IV) between versions of each trace is also shown in Table 5.1. We select different IV for each trace such that the fragmentation and version similarity are varied

Table 5.2: The change of the deduplication ratio (DR) and the speed factor (SF) with different LBW size and a fixed mean T_{dc} (mean T_{dc} =120)

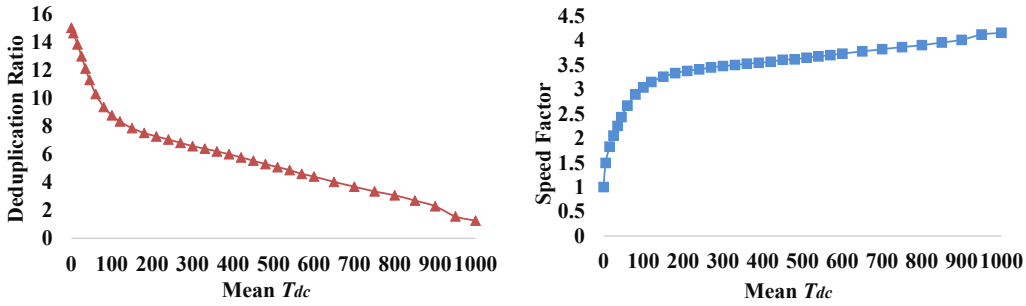
LBW size	16	14	12	10	8	6	4	2
DR	8.53	8.53	8.53	8.50	8.45	8.37	8.27	8.07
SF	2.80	2.84	2.87	2.92	3.01	3.14	3.16	3.18

in different traces. Before replaying a trace, all data in the cache is cleaned. For each experiment, we run the traces 3 times and present the average value.

5.6.2 Influence of LBW Size and Mean Value of T_{dc}

In the LBW design, a different configuration of LBW size and a different mean value of CNRC threshold T_{dc} will influence both of the deduplication ratio and the speed factor. We use the MAC2 trace as an example and design a set of experiments to investigate the relationships. In the restore process, we use FAA as the restore caching scheme and set the FAA size as 8 containers. To investigate the influence of LBW size, we fix the mean T_{dc} to 120 by setting the two bounds while the size of the LBW changes from 16 to 2 containers. The results are shown in Table 5.2. As the LBW size decreases, the deduplication ratio decreases slightly. It indicates that, when the threshold configuration is the same, if we reduce the LBW size, less information is maintained in the window and more data chunks are selected to be rewritten. The deduplication ratio decreases even faster when the LBW size is smaller than the FAA size, which is caused by a large number of unnecessary rewrites due to a small LBW size. Differently, when the LBW size is close to or even smaller than the FAA size as shown in Table 5.2, there is a relatively large increase of the speed factor. Therefore, to achieve a good tradeoff between the deduplication ratio and the speed factor, the LBW size should be compatible with the FAA size, which is the cache-effective range in this case.

In another experiment, we fix the LBW size to 8 containers, FAA size to 8 containers, and vary T_{dc} from 0 to 1000. With a higher T_{dc}, more old containers will be rewritten since it is harder for each old container to have more than T_{dc} duplicated data chunks



(a) The deduplication ratio variation

(b) The speed factor variation

Figure 5.6: The change of the deduplication ratio and the speed factor with different mean T_{dc} and a fixed LBW size (LBW=5)

in the LBW. Therefore, as shown in Figure 5.6(a), the deduplication ratio decreases as the T_{dc} increases. Also, the speed factor increases since more data chunks are rewritten as shown in Figure 5.6(b). When T_{dc} is zero, there is no rewrite and the deduplication regresses to the Normal deduplication case. On the other hand, when T_{dc}= 1000 (one container stores 1000 data chunks in average), almost every duplicated chunk is rewritten. Thus, no deduplication is performed (i.e., deduplication ratio is 1) and the speed factor is 4. Note that, the decreasing of the deduplication ratio as well as the increasing of the speed factor are at a faster pace when T_{dc} is less than 100. When T_{dc} is larger than 100, the two curves vary linearly in a slower pace. It shows that we can achieve a good tradeoff between deduplication ratio reduction and restore performance improvement when T_{dc} is set smaller than 100 for this trace.

5.6.3 Deduplication Ratio & Speed Factor Comparison

In this subsection, we compare the performance of Capping with LBW in terms of deduplication ratio and speed factor. In order to get the speed factor, we choose FAA as the restore engine. In both cases, we set the FAA size to 8 containers. For Capping, the segment size is set to 5 containers and we run the Capping with capping level varying from 0 to 70. For LBW, the LBW size is fixed to 8 containers and we vary the T_{dc} from 0 to 1000 by adjusting the targeted deduplication ratio and the targeted number of container reads. To fairly compare Capping and LBW, we need to compare the speed factor of the two designs when they have the same deduplication ratio.

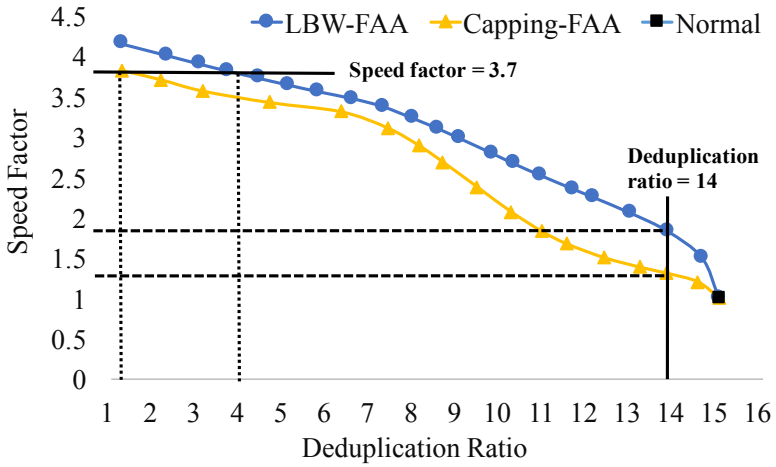


Figure 5.7: The speed factor of Capping and LBW with different deduplication ratios when deduplicating the last version of MAC2. The Normal scheme is a black point in the figure

The results are shown in Figure 5.7. We can clearly conclude that when the deduplication ratio is the same, the speed factor of LBW is always higher than that of Capping. For examples, when the speed factor = 3.7, Capping has a deduplication ratio = 1.2 which indicates a very high rewrite number (close to no deduplication). With the same speed factor, LBW has a deduplication ratio = 4. With the same deduplication ratio =14, Capping has a speed factor = 1.27 and LBW has a speed factor = 1.75 which is about 38% higher than that of Capping. If no rewrite is performed (Normal deduplication process), the deduplication ratio is 15 while the speed factor is close to 1. When the deduplication ratio increases from 8 to 14 which indicates fewer data chunks are rewritten and close to no rewrite case (Normal), compared with Capping, the speed factor of LBW is about from 20% to 40% higher. In general, LBW can achieve a better tradeoff between the deduplication ratio reduction and speed factor improvement especially when the deduplication ratio is high.

5.6.4 Restore Performance Comparison

To obtain the performance of Normal, Capping, FCRC, and LBW as deduplication engines, we pair them with two restore engines (FAA and ALACC). Since a small reduction of deduplication ratio (i.e., fewer data chunk rewrites) is preferred, we configure

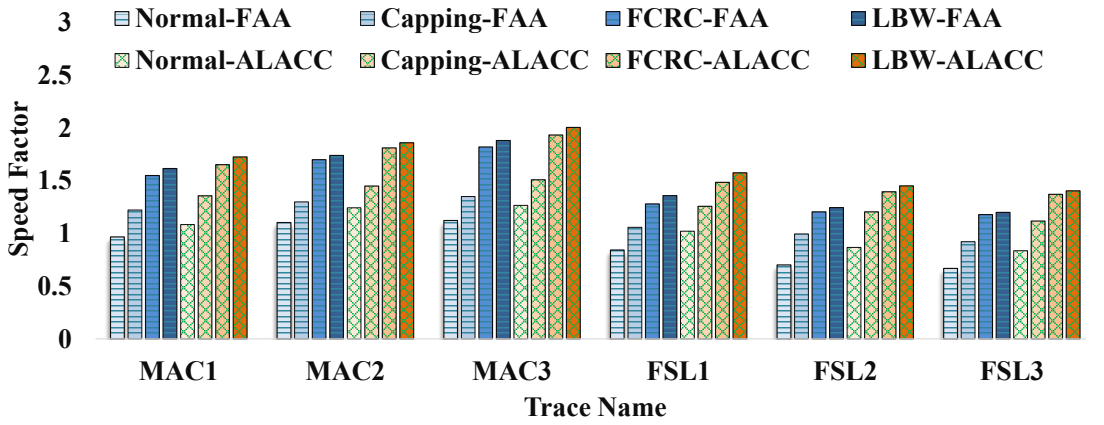
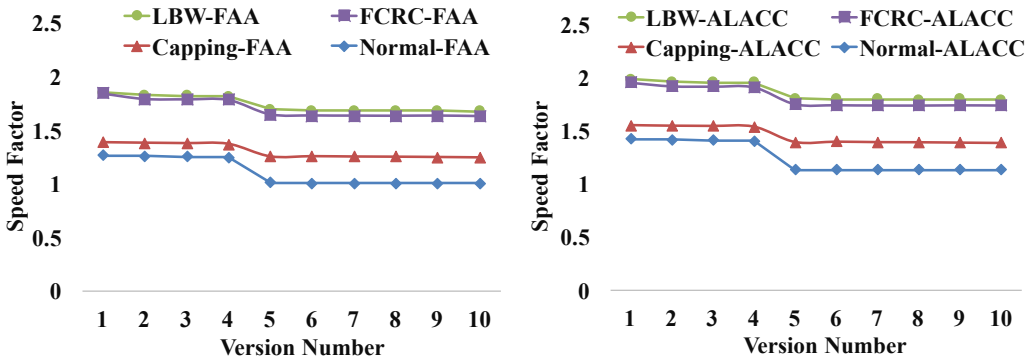


Figure 5.8: The mean speed factor of Normal, Capping, FCRC, and LBW for the 10 versions of six traces when restoring with FAA and ALACC

the parameters to target the reduction of deduplication ratio of Capping, FCRC, and LBW in the following experiments to 7%.

Each trace has 10 versions of backup and the mean speed factors achieved by all 8 combination schemes are shown in Figure 5.8. If data chunk rewrite is not applied (Normal design) in deduplication process, even with FAA or ALACC as the restore caching schemes, the speed factor is always lower than 1.3. In three FSL traces, the speed factor is even lower than 1. That means reading out a 4MB size container cannot even restore 1MB size of data. It indicates that the data chunks are fragmented and the two caching schemes still cannot reduce the large number of required container reads. By rewriting duplicated data chunks, the speed factor has improved. For all 6 traces, we can find that with the same deduplication ratio, LBW always achieves the highest speed factor (best restore performance) when the same restore engine is used. The speed factor of FCRC is lower than that of LBW, but higher than that of Capping. Due to a lower averaged self-reference count ratio (ASR) shown in Table 5.1, the duplicated data chunks of FSL traces are more fragmented than that of MAC traces. Thus, the overall speed factor of FSL traces is lower than that of MAC traces. In general, when using the same restore caching scheme, the speed factor of LBW is up to 97% higher than that of Normal, 41% higher than that of Capping, and 7% higher than that of FCRC.

Replaying the trace MAC2, the speed factors of using FAA and ALACC as the restore caching scheme are shown in Figure 5.9(a) and Figure 5.9(b) respectively. Among



(a) The speed factor of using FAA as the restore caching scheme

(b) The speed factor of using ALACC as the restore caching scheme

Figure 5.9: The speed factor comparison of 8 combinations of deduplication-restore experiments when deduplicating and restoring the trace MAC2

the 10 versions, the duplicated data chunks are more fragmented after version 4. Therefore, there is a clear speed factor drop at version 5. As shown in both figures, the speed factor of LBW is always the highest in all 10 versions. By using the FCRC, the speed factor is improved a lot comparing with the Normal and Capping schemes. LBW further improves the speed factor when compared with FCRC by addressing the rewrite accuracy issues caused by a fixed segment partition. In general, with the save deduplication ratio, LBW can effectively reduce the number of container reads such that its restore performance is always the best.

5.7 Conclusion and Future Work

Rewriting duplicated data chunks during the deduplication process is an important and effective way to reduce container reads, which cannot be achieved by caching schemes during the restore. In this research, we discuss the limitations of capping design and propose the FCRC scheme based on capping to improve the rewrite selection accuracy. To further reduce the inaccurate rewrite decisions caused by the fixed segment partition used by capping, we propose the sliding look-back window based rewrite approach. By combining the look-back mechanism and the caching effects of the restore process, our approach makes better tradeoffs between the reduction of deduplication ratio and container reads. In our experiments, the speed factor of LBW is always better than that

of Capping and FCRC when the deduplication ratio is the same. In our future work, the adaptive LBW size and more intelligent rewrite policies will be investigated.

Chapter 6

IS-HBase: A In-Storage Computing based HBase Architecture in Big Data Infrastructure

6.1 Introduction

During the past half a century, persistent storage devices have evolved for keeping/storing the desired data. Among them, the magnetic recording based disk drives (i.e., HDDs) were introduced over 60 years ago and have been widely used in persistent storage systems, especially for large scale and low cost storage systems [155]. An HDD provides a set of simple and basic block based storage interfaces and more sophisticated management is achieved in the management of upper layers like a storage controller or a software stack in storage servers. During the past 20 years, flash memory based storage devices become one of the most important storage media [19]. Due to the high performance of both sequential and random accesses of flash memory based storage devices like solid state drives (SSDs), SSDs are widely used in the primary storage systems [156, 18] to achieve higher performance.

Since flash pages in an SSD cannot be directly updated, an SSD typically has an

indirect addressing and a complicated process to garbage collects (GC) dirty (invalid) flash pages [19, 157]. To support the GC and other management services inside an SSD, SSD is equipped with GB-level DRAM and multiple CPU cores [158, 159]. Therefore, the computing and memory resources of a single SSD can potentially support a small operating system and execute some simple applications. Although the performance of an SSD is higher than that of an HDD, the performance and bandwidth gaps between main memory and the storage still create a large performance penalty, especially for I/O intensive applications when a large amount of data is read out from or written to storage devices[160].

One of the possible solutions is to pre-process the stored data in storage devices and thus reduce the amount of traffic between storage devices and application servers, i.e., using the concept of in-storage computing or computational storage. In recent years, with the help of spare CPU and memory resources in a storage device like HDD or SSD, the approaches that enable the storage device with simple computing functions are introduced, for examples, in active disk drives [161, 162, 163], smart SSD [158, 164, 165, 166], data processing based SSD [167, 168, 169, 170], and the KVSSD proposed by Samsung [171]. In KVSSD, I/O intensive jobs like searching and scanning the key-value pairs (KV-pairs) are executed in SSD instead of at the server side, which effectively reduces the amount of data being read out from SSD to memory. Similarly, the Kinetic drives developed by Seagate runs a key-value store (LevelDB) in an HDD and provides the KV interfaces through Ethernet connections [172, 173, 174].

The concept of in-storage computing has been extended to other scenarios to improve the application performance, especially when the size of data that really requested by an application is much smaller than the data being read out from storage. YouSQL [175] offloads several I/O intensive functions of MySQL like range queries to SSD such that when “Select” or other range queries are called, the overall query per second (QPS) can be effectively improved. With the help of in-storage computing, not only the required I/O is effectively reduced, but also it can achieve better parallelism from multiple storage devices. If a computing process of an application is not bottlenecked by the required I/O, it can continue the execution of computing such that the throughput and overall performance can be greatly improved.

In the current and most popular IT infrastructure, the compute cluster is disaggregated from the storage cluster to achieve higher flexibility, availability, manageability, and scalability. A compute node in the compute cluster has a relatively powerful CPU and a large memory space which supports several applications like various stateless services, caching, and compute services. The storage cluster consists of one or multiple large storage pools with different storage devices and it provides several storage interfaces including block, file system, Key-Value, and Object for different services running in the compute cluster. Compute and storage servers are connected via high speed networks. Cloud service providers [176] (e.g., Amazon S3 [177], Microsoft Windows Azure [178], and Huawei Cloud [179]) and many large Internet companies (e.g, Facebook and Google) have deployed and implemented their IT infrastructures with this type of compute-storage disaggregation.

However, the performance gap existing between memory and storage also appears in the aforementioned compute-storage disaggregated IT infrastructures. All the desired data are required to transfer through the network connections between the compute cluster and the storage cluster. When the workloads are light or stable, there are fewer performance issues caused by the heavy connection traffic. However, when the spiked I/O workloads are generated by applications with intensive reads/writes, the available connection bandwidth can be easily over-saturated. The performance of the applications requiring high storage I/Os will be potentially impacted. To address these issues, the in-storage computing concept can also be applied to reduce the amount of traffic between compute and storage clusters. Since a storage server, which may connect to multiple storage devices, has a relatively powerful CPU and a larger memory space than those in a storage device, more complicated functions/services can be executed in the storage servers to pre-process the stored data instead of only processing them in application servers.

HBase [180, 44, 181, 182] (derived from BigTable [183]) is an important and widely used column-oriented distributed key-value store and its performance can be severally impacted when HBase is deployed in a compute-storage disaggregated IT infrastructure. We explore the potential of applying in-storage computing to improve the performance of HBase by reducing the traffics between compute servers and storage servers.

RegionServers of HBase are responsible to receive and respond to client queries. Typically, a RegionServer caches the KV-pairs from write queries (e.g., Put and Delete) in Memtable and appends them to a write-ahead-log (WAL) to ensure data reliability. When a Memtable is full, KV-pairs are sorted and written to storage as a single file called HFile which consists of tens of data blocks. Read queries (e.g, Get and Scan) usually trigger many block reads and cause a serious read amplification (i.e., the data being sent to the client is much smaller than the data being read out from storage).

In the legacy deployment of HBase, for one RegionServer, its HFiles are usually stored in the same host via HDFS interfaces [184, 185, 186] (i.e., the host acts as both RegionServer and HDFS DataNode). However, in the compute-storage disaggregated infrastructure, RegionServers are running in the compute nodes of the compute cluster. HFiles are stored in the storage nodes of the storage cluster and all the I/Os will have to go through the connecting network between compute and storage clusters. To investigate how serious the performance (especially read performance) of HBase is influenced by the network condition, we evaluated the performance of HBase with different available network bandwidths. The major observations are: 1) For the key-value workload generated by YCSB with either a Zipfian or Uniform distribution [57], Get queries can cause an extremely high network traffic (e.g., the data being read out is about 40-67 times higher than that being sent to a client); 2) Scan queries with a shorter scan length usually have a higher read amplification; 3) For the scan queries with filters, the more data can be filtered out, the larger the read amplification is; 4) When the available network bandwidth is low, the throughput of Get and Scan queries become very low due to the long network transmission time of data blocks.

To address the performance issue caused by the insufficient network bandwidth, we adopt the in-storage computing approach to redesign the read paths and compactions of HBase (called *IS-HBase*). IS-HBase offloads part of the read logic and compaction routine from HBase RegionServers to the storage nodes as a service module called In-Storage ScanNer (ISSN). For Get and some Scan queries (depends on the Scan query requirements and network conditions), data blocks are pre-processed by ISSNs in the storage nodes and only the useful results (the requested KV-pairs) are sent through the network and combined in the RegionServer. Since only the requested KV-pairs are

transferred through the network, it effectively reduces the network bandwidth consumption. Therefore, a higher performance can still be achieved when the available network bandwidth is limited. Moreover, if the HFiles related to one query are scattered to different storage nodes, IS-HBase can search the KV-pairs in parallel. This further improves the performance. For a compaction, one ISSN is selected to execute the compaction of a set of HFiles such that it avoids the round trip transmission of KV-pairs to RegionServers and back during a compaction.

In-storage computing usually brings in a challenging caching issue to applications. In IS-HBase, since only the requested KV-pairs are sent to RegionServers for read queries instead of sending all relevant data blocks, the original block cache design at a RegionServer cannot be used directly. To address this caching issue, we propose a self-adaptive caching scheme for IS-HBase. First, the cache space is dynamically allocated between complete data blocks and partial blocks. The partial blocks cache the KV-pairs from the same data blocks and transmitted by ISSNs in response to read queries. Second, since the KV workloads may have strong key-space localities [61], IS-HBase upgrades some of the partial blocks into complete blocks to reduce the cache misses. Third, the cache space for complete blocks and partially blocks are dynamically changed to better adapt to the workloads and network bandwidth variations.

According to our experiments, the IS-HBase can achieve up to 95% and 97% of network traffic reduction for GET with Zipfian and Uniform distributions respectively, and the QPS (Queries Per Second) is barely affected by network bandwidth variation. Also, there is up to 94% data transmission reduction for Scan queries with column filters. Moreover, the IS-HBase with self-adaptive block cache can further improve the average QPS by 26.31% compared with the IS-HBase with KV-pair cache only.

This chapter is organized as follows. We briefly introduce the system background of HBase and compute-storage disaggregated infrastructure in Section 2. Related work on active storage devices is discussed in Section 3. We evaluate and discuss the performance of HBase in a compute-storage disaggregated infrastructure in Section 4. The results motivate us to offload the I/O intensive module from HBase to storage nodes. In Section 5 we introduce IS-HBase which offloads the HFile scanner module originally in RegionServers to storage nodes as the in-storage processing unit. The proposed system architecture, the flow of data read logic, and the data consistency/correctness are also

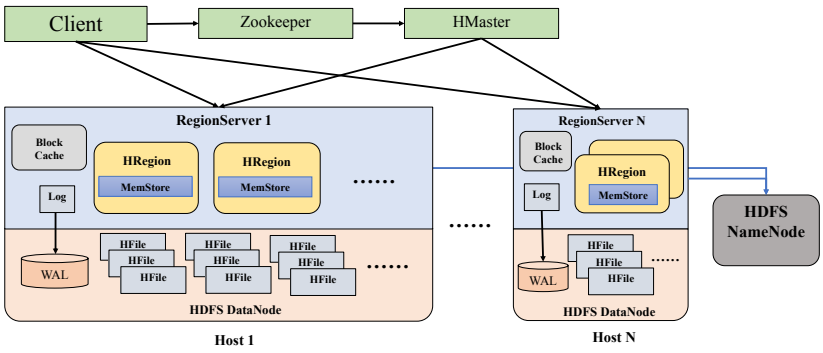


Figure 6.1: The legacy architecture of HBase

discussed in this section. A self-adaptive caching scheme is described in Section 6 and we present the evaluation results in Section 7. Finally, we conclude the research and discuss future work in Section 8.

6.2 Background

In this section, we first introduce the architecture, basic operations, and data management of HBase. Then, we will introduce the compute-storage disaggregated infrastructures.

6.2.1 HBase Preliminary

HBase [180], a distributed column-oriented key-value (KV) database, is an open-source implementation of Google BigTable [183]. The architecture of a typical HBase is shown in Figure 6.1. It consists of three components: 1) HBase RegionServers receive and respond to the queries from clients; 2) A HBase Master decides the key range assignment, manages tables, and handles high level services; and 3) Zookeeper monitors and maintains the states of the clusters. Since HBase is a KV based NoSQL database, it provides simple interfaces to achieve data read/write queries including Get, Scan, Put, and Delete. Get is a point-lookup query, which requests the KV-pair with a given key. Scan returns a set of KV-pairs that satisfy the scan requirement (i.e., start from a certain key, scan a range of KV-pairs, and filter out some KV-pairs). Put inserts new KV-pairs and Delete removes the existing KV-pairs from HBase.

HBase applies an LSM-tree [187] indexing scheme to persistently store the KV-pairs in a backend file system to achieve high write performance. KV-pairs are sorted and stored in files called HFiles. Different from LevelDB or RocksDB which are also LSM-tree based KV-stores, HBase does not have the "level" concept. All HFiles are maintained at the same level. Therefore, the key range of one HFile may overlap with the key ranges of other HFiles. In a RegionServer, new KV-pairs or the special KV-pairs to indicate the deletions are first temporally cached in a memory buffer called MemStore. At the same time, to ensure data persistency, these KV-pairs are appended to a Write-Ahead-Log (WAL)in storage. One HRegion maintains one MemStore. When a MemStore is full, the KV-pairs in the MemStore are sorted in a pre-defined order and written to storage as a new HFile. When a client wants to read a KV-pair (e.g., Get query), the MemStore and all HFiles that may cover the key of this KV-pair are searched. Since deleting and updating a certain KV-pair is achieved by inserting a new KV-pair with the same key, there may be multiple KV-pairs with the same key being stored in Memstore and HFiles. Therefore, to respond to a Get request, HBase needs to find all these KV-pairs and a RegionServer combines the results before responding to a client. Only the KV-pairs that satisfying the read requirement (e.g., the latest KV-pairs) are returned to the client.

As more and more new KV-pairs are inserted, the deleted KV-pairs and invalid KV-pairs (e.g., KV-pairs that have been updated) should be cleaned to save space. Also, as the number of HFiles increases, more HFiles will be searched to respond to Get or Scan queries such that the read performance is impacted. Therefore, a compaction is introduced to combine some HFiles into a single larger HFile. When a compaction is triggered in one RegionServer, some HFiles are selected to carry out the compaction based on some pre-defined criteria. Then, the RegionServer sequentially reads the KV-pairs from these HFiles, compares their keys, and removes the invalid KV-pairs. All the valid KV-pairs are sorted and written back as a new HFile.

HFile is organized as a tree structure similar to a B+ tree. KV-pairs are stored in data blocks (e.g., 64KB per data block) in a sorted order. Data blocks are appended from the beginning of an HFile. Then, other metadata blocks including indexing blocks, filter blocks, and other blocks containing the metadata information are stored after the data blocks. An indexing block maintains the key-ranges, the sizes, and the offsets of the

data blocks. If a single level indexing is not enough to index the data blocks, a second or third level indexing blocks are created automatically. A RegionServer maintains a manifest including the mapping of HFiles to their key-ranges.

To look up a key for Get query or find a start KV-pair of a Scan query, the query is first sent to one of the RegionServers which maintains the key-range containing that key. Then, based on the key-ranges of the HFiles, the HFiles whose key-ranges overlapped with the key range are selected. The RegionServer starts a HFile scanner for each selected HFile to search KV-pairs with the given key and each HFile scanner will return the requested KV-pairs or Null if none exists.

An HFile scanner first searches an indexing block and locates the data blocks whose key-ranges cover the key. Then, the HFile scanner reads out the data blocks to a block cache, applies a binary search the KV-pairs of the data blocks to check the existence of the requested KV-pair. Since there may be multiple KV-pairs that have the same key and store in different HFiles, the RegionServer builds a heap to sort the KV-pairs returned from all the HFile scanners and picks the top/latest one to return to the client.

For Scan queries, a client specifies the start position of the scan via providing a start-key. The process of finding a start-key of each HFile is the same as that of Get. To continuously get the consecutive KV-pairs after the start-key, a client can call Next() once the real start-key (i.e., an existing key equals to or greater than the start-key) identified. When Next() is called, the top KV-pair is popped out from the heap and returned to the client. Then, the HFile scanner whose KV-pair is popped out moves to get the next KV-pair if it exists and inserts the new KV-pair to the heap. A new top KV-pair is then generated from the heap. One example is shown in Figure 6.2.

Each RegionServer maintains a read cache for the accessed blocks (called *block cache*) including both data blocks and metadata blocks being read out from HFiles. Each cached block is indexed by its block sequence number which is a unique ID for each block being created in the HBase. For read queries and also compactions, metadata blocks are first read to the block cache if there are cache misses and then searched by RegionServers. Based on the outcomes of metadata block searches, the corresponding data blocks are fetched to the cache. Blocks in block cache are evicted based on an LRU policy. Since metadata blocks have much higher frequency to be accessed when responding to the read queries, these metadata blocks are usually cached in the block cache for a long

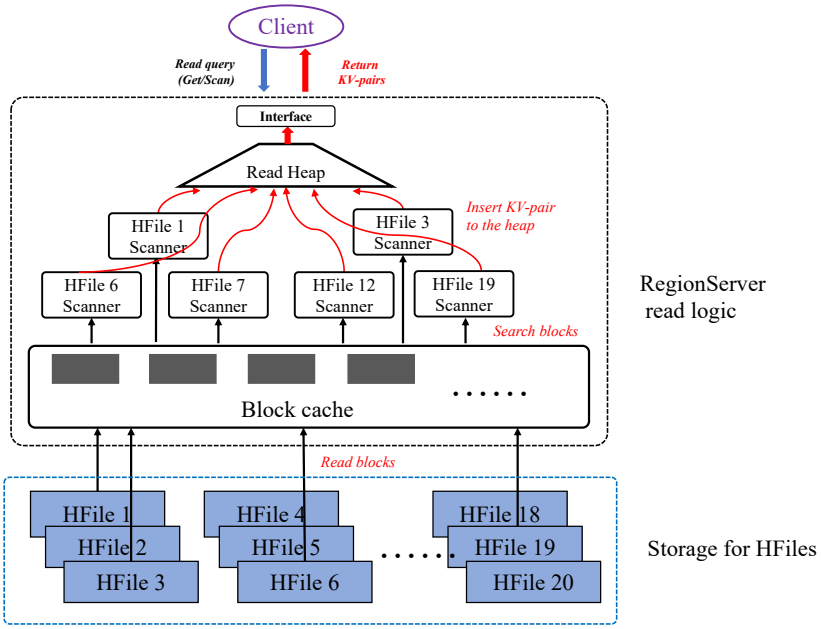


Figure 6.2: One example of the read logic in legacy HBase. When a client issues a Get or Scan request with a given key, the RegionServer first determines which HFiles may be related to the current query (e.g., HFiles 1, 3, 6, 7, 12, and 19). Then, the RegionServer starts a HFile scanner for each HFile involved. An HFile scanner reads the data blocks from HFiles to the block cache and applies a binary search to locate the first key that satisfies the read requirement. Then, the relevant KV-pair is inserted to a read heap. The read heap returns the top KV-pair to the client.

period of time. During a compaction, some existing HFiles are compacted to a new HFile and these old HFiles are discarded and these deleted blocks in the block cache are also removed.

HBase itself is not responsible for its data persistency. In the legacy HBase deployment, HFiles are stored in HDFS. A RegionServer and an HDFS DataNode can be deployed in the same host. If the HFiles that are managed by one RegionServer are also stored in the DataNode at the same host, a RegionServer does not need to read data from other hosts. However, if HFiles are not stored in the host where the RegionServer is deployed, it will create some network traffic to read and write data blocks. Due to the LSM-tree based design of HBase, it may have both serious read and write amplifications. One data block usually consists of hundreds of KV-pairs. When one KV-pair

is requested, the whole data block needs to be sent to the RegionServer through the network which creates a read amplification. Moreover, one key may exist in multiple HFiles (e.g., a KV-pair has been updated multiple times and stored in different HFiles), one Get query will trigger multiple data block reads which makes the read amplification even worse. The size mismatch between the requested KV-pair and the data being read out not only consumes the available network bandwidth but also causes a performance degradation especially when the available network bandwidth is low.

6.2.2 Compute-Storage Disaggregated Infrastructure

During the past 10 years, the IT infrastructure has a trend to separate the compute cluster from the storage cluster to improve its flexibility and scalability. Applications and services are running in virtual machines (VMs) and containers in a distributed environment to achieve better resources management. In such a setup, storage devices can no longer be tightly coupled with one compute node. Also, applications have various demands on different data I/O interfaces including block, file system, blob, object, and key-value. Therefore, organizing the storage clusters into one or more shared storage nodes/pools and providing different I/O interfaces further enhance the availability of the storage services. Therefore, compute-storage disaggregation becomes an important architecture for large scale IT infrastructures.

As shown in Figure 6.3, in a compute-storage disaggregated infrastructure, the compute cluster focuses on computing intensive applications. At the same time, the storage cluster provides data persistence capabilities to the applications and supports various I/O interfaces. A high-performance network is used to connect the storage and compute clusters. However, the disaggregation of storage and computation may lead to a performance penalty for data-intensive applications. When many read/write requests are issued intensively by some applications at the same time, the available network bandwidth for other applications reduces tremendously. This will cause a serious performance issue especially for the applications that have high I/O demands.

HBase is a typical application that has high I/O demands and it can easily encounter the performance penalty in a compute-storage disaggregated infrastructure. RegionServers are deployed in the compute nodes and all HFiles are stored in the storage nodes. For read requests, once the block cache misses happen, data blocks are accessed

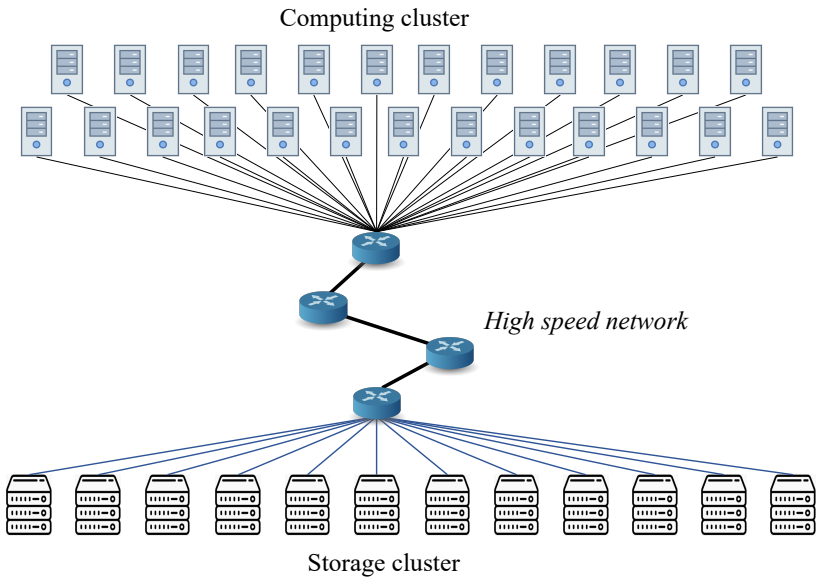


Figure 6.3: Compute-storage disaggregated infrastructure.

from storage nodes through the network to RegionServers. If client requests are spiked, a huge number of data blocks have to read out from storage nodes during a short period of time. If the available network bandwidth is insufficient, read performance will be extremely low (i.e., high latency and low QPS). Second, a compaction also creates large amounts of read and write traffics through the network. If a compaction is slowed down due to the low network bandwidth, both read and write performance will be heavily influenced. The details will be explored in the following sections.

6.3 Related Work

In this section, we first review the studies on active storage devices. Then, we will introduce the related work on near-data processing.

6.3.1 Active Storage Devices

During the past 60 years, persistent storage devices are mainly used in a passive away. That is, the host decides where and when to read and write data to a storage device. Data is read out from a storage device to memory and being processed. When it is

necessary (e.g., the data in memory is new or updated), data is written to the storage to ensure persistency. In most cases, a block interface is used. However, due to the lower I/O throughput of storage devices comparing with memory, storage I/O becomes the performance bottleneck for I/O intensive applications.

The storage devices produced in recent years, including HDD and SSD, are equipped with additional CPU cores and DRAM to support the management services in the device. It makes pre-processing data in storage devices feasible. The concept of the active disk was early introduced in [161, 162, 163]. The design of the active disk demonstrated the possibility and benefits of using the processing power of storage devices. Moving the list intersection operation (i.e., the operation to find the common elements between two lists) from memory to SSD is one example to avoid the heavy I/O between memory and SSD [167]. No matter which intersection exists, the two lists must be read out from SSD to memory originally to identify the intersections. If the list size is extremely large, it will cause a large number of SSD reads and the performance will be low. By benefiting from the relatively high search speed inside the SSD, processing the list intersection inside an SSD can improve the throughput and reduce the energy consuming. Similarly, more complicated searching functions can be implemented in an SSD to reduce the I/O accesses[170].

Applications like Hadoop and data mining are also I/O intensive. The benefits and tradeoffs of moving some Hadoop tasks to SSD are introduced in [164] and [168]. The interfaces of defining a job to be executed in SSD are provided and it can also be used in a large scale distributed environment. The in-storage processing based SSD is also used to accelerate data mining jobs [165]. The global merging operations are executed inside SSD which can reduce the required number of data reads and writes.

Database is a widely used I/O intensive application. Several research studies using active devices to speed up the performance [171, 172, 175, 158]. KVSSD [171] and Kinetic Drives[172] are two examples that implement a NoSQL database (key-value store) inside a storage device. KVSSD integrates the key-value store function with the SSD firmware and provides object-store like interfaces to applications. KVSSD can provide a high I/O efficiency and cost-effectiveness. Similar to KVSSD, a Kinetic drive runs a mini operating system and a key-value store in an HDD. Clients no longer need to read and write data based on fixed-size blocks in a Kinetic drive. It supports

the key-value interface including Get, Put, Delete, and Scan. Moreover, clients can access the disk via Ethernet connections. This makes the usage of storage device more flexible. Active devices are also used to speed up SQL databases. By executing the Select query in SSD, the performance and energy consuming are both improved [158]. Moreover, YourSQL [175] offloads the execution of a Scan query to SSD. It can improve the throughput from 3.6X to 15X by reducing the data being read out from SSD to memory.

In general, active storage device is a way of making tradeoffs between reducing I/O overhead and increasing processing time due to limited CPU and memory resources inside the device. By carefully selecting an application and its functions to be offloaded to the device, the overall performance and energy saving can be effectively improved.

6.3.2 Near-Data Processing

To bridge the performance gap between {memory and storage} and {CPU and memory}, the concept of near-data processing was proposed. Active storage device and Processing-In Memory (PIM) are two typical approaches of near-data processing. Co-KV [188] and DStore [189] proposed to execute compactions for LSM-tree based key-value store in the storage device. In an LSM-tree based key-value store (e.g., LevelDB or RocksDB), the files being flushed out from memory needs to be compacted such that the deleted and updated key-value pairs can be removed. Compactions can benefit both space needed and read performance. However, a large amount of data needs to be read out and write back during a compaction. This brings a high overhead. Co-KV and DStore offload the compaction operation from host to the storage device like SSD such that it can avoid the required compaction traffic between the host and its storage devices. A compaction can be executed inside the storage devices as a background job.

Near-data processing is also used to speed up the processing of big data workloads. Biscuit [67] proposed a framework to apply near-data processing in SSD for big data workloads such as doing wordcount, string searching, DB scanning and filtering, and pointer chasing in graph processing. Biscuit provides a library for clients to program and deploy their data processing modules in the host and also in SSDs such that some of the I/O intensive jobs can be executed in SSDs. In this way, it can effectively reduce the traffic between storage devices and the host to improve the overall throughput.

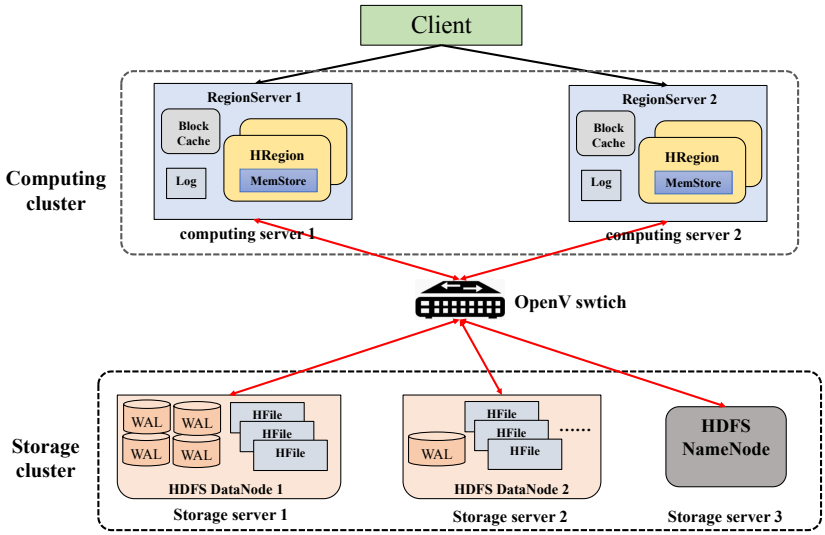


Figure 6.4: The HBase deployment in a compute-storage disaggregated setup.

6.4 Performance Degradation of HBase in Compute-Storage Disaggregated Infrastructures

To comprehensively investigate the performance issue of HBase when it is deployed in an infrastructure that compute and storage clusters are disaggregated, we conduct a series of experiments to explore the relationship between network condition and HBase performance. We use the following measurement metrics: 1) network amplification (NET-AM): the data being transmitted from storage nodes through the network when responding to a client request divided by the amount of data being actually needed by the client; and 2) Queries Per Second (QPS). To read/write the same number of KV-pairs, a higher NET-AM indicates more data are required to transfer through the network. Thus, the performance is more sensitive to the available network bandwidth. QPS measures the overall performance of HBase. We use an Open Vswitch [190] to control the available network bandwidth between compute servers (where RegionServers are deployed) and storage servers (where HDFS DataNodes are deployed). We focus on evaluating the NET-AM of Put, Get, and Scan queries and the performance correlation to the available network bandwidth.

6.4.1 Experimental Setup

Figure 6.4 depicts the deployment of HBase in our test environment. In a compute-storage disaggregated setup, there are two servers deployed with HBase RegionServers (i.e., compute servers) and two others deployed as the HDFS DataNodes (i.e., storage servers). An HDFS NameNode is deployed in one of the storage servers. Compute servers and storage servers are connected via an Open Vswitch [190], such that we can dynamically adjust the available network bandwidth. The HDFS version is 2.7.3 and the HBase version is 1.2.6. Figure 6.5 shows the legacy deployment of HBase. We adopt the pseudo-distribution mode of HBase such that we can collect the network traffic more precisely. We keep one replicate of HFiles and the size of HFile data block is configured as 64KB.

We use YCSB [57] *workloada* to issue Get, Put, and Scan queries. We use two different hotness distributions of workloads: 1) Zipfian; and 2) Uniform. Zipfian issues more queries to some of the "hot" KV-pairs and the hotness follows a Zipfian distribution. It has a better key-space locality. Under Zipfian workloads, the block cache has a higher cache hit probability and it can effectively reduce the number of block reads for Get and Scan queries. In the workloads of Uniform distribution, the keys of queries are randomly generated. Therefore, there are no hot KV-pairs. We set the size of each KV-pair as 1KB and thus one data block has about 64 KV-pairs. Since YCSB does not support the Scan query with filters, we developed a new benchmark tool that can generate Scan queries with column filters. The network status including the available bandwidth and traffic amount is collected by the Linux Iptables tool [191].

6.4.2 NET-AM Evaluation

In this subsection, we evaluate the NET-AM of Put, Get, Scan, and Scan with filter.

Put In this experiment, YCSB writes KV-pairs into HBase in the same table and the same column family. We write different numbers of KV-pairs to HBase to find their corresponding NET-AMs. During the write process, HFiles are compacted and thus data blocks are read out from these HFiles. After compaction, valid KV-pairs are written back to storage as a new HFile. These compactions make NET-AM higher. As shown in Figure 6.6, the X-axis is the total number of records written to HBase, and

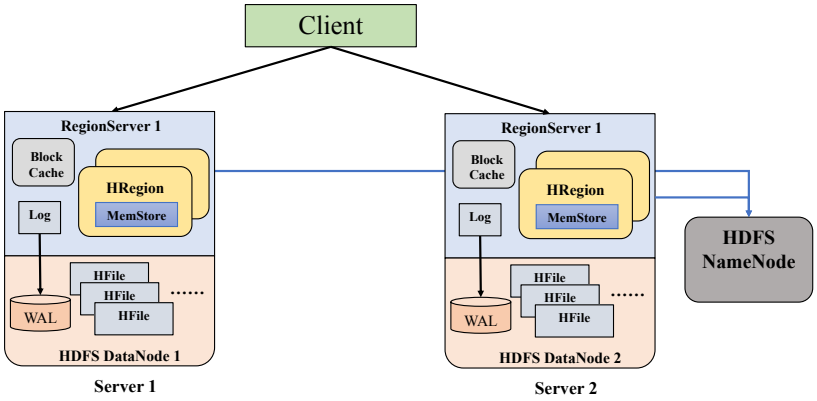


Figure 6.5: The legacy deployment of HBase in our experiment

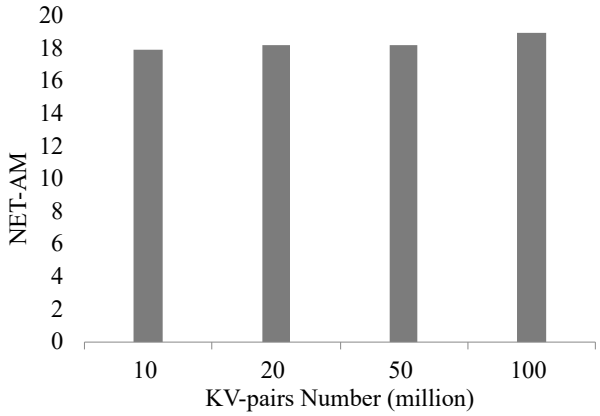


Figure 6.6: NET-AM of write queries

the Y-axis is the measured NET-AM of writes (data being accessed by storage servers divided by data being sent to the RegionServers). The NET-AM becomes higher due to HBase compactions. For example, if we insert 10 million KV-pairs (about 10 GB data in total), the total data accessed by the storage server is about 18X (about 180 GB in total) after HBase is stabilized (i.e., compactions no longer happen). During the compaction, the data has to be read from several HFiles on HDFS servers and transferred to the RegionServer through the network. After the compaction, a new HFile has to be written back to the HDFS server again. The same data will be read and written several times which leads to a higher NET-AM for write.

As the total number of KV-pair increases, the NET-AM only increases slightly. For example, the NET-AM of inserting 10 million KV-pairs and 100 million KV-pairs are 18 and 18.7 respectively. As more KV-pairs inserted, more HFiles need to be combined during a compaction. If deletion and updating queries are also called by clients, NET-AM can be a bit lower because some KV-pairs are cleaned during compactions. In general, due to the LSM-tree based design, it always has a write amplification issue and creates a higher NET-AM.

Get To evaluate the NET-AM and the QPS of Get queries, we first insert 100 million KV-pairs to the HBase table as the basic dataset for read via YCSB LOAD operations. Then, we use YCSB to generate Get queries with a uniform distribution (Read-Uniform) or Zipfian distribution (Read-Zipfian). The former randomly access the 100 million KV-pairs while the later access more on some hot KV-pairs and follows a Zipfian distribution. At the same time, we vary the ratio of KV-pairs being read from HBase among the 100 million KV-pairs. As shown in Figure 6.7, the X-axis represents the ratio of the requested data against the total dataset, and the Y-axis represents the NET-AM of reads (the size of data being read from a storage server to the RegionServer divided by the requested data size). When the ratio of the requested data is less than 0.1%, the NET-AM is about 90 in Read-Uniform, which is about 30% higher than those when the requested data ratios is higher than 1%. Similarly, for Read-Zipfian, when the ratio of the requested data is less than 0.1%, the NET-AM is about 80 and it is about 85% higher than those when the requested data ratios is higher than 1%.

The evaluation results verify that when the requested data set size is extremely small (i.e., less than 0.1% KV-pairs are read), the read amplification is more serious. The amount of metadata (e.g., HFile metadata and metadata blocks) being read out is relatively constant because once those data is read to the RegionServer, they are most likely to be cached. When the requested data size ratio is larger, the metadata overhead is amortized and thus the NET-AM decreases. Note that, the NET-AM of read intensive workloads with a Uniform distribution is always higher than that of a Zipfian distribution. The difference is caused by the variations of data locality and cache efficiency. If the requested data locality is very low (e.g., randomly gets KV-pairs in the entire key range), theoretically, the read amplification can be higher than 64 if the data block cache hit ratio is very low. That is, when a client reads out one KV-pair

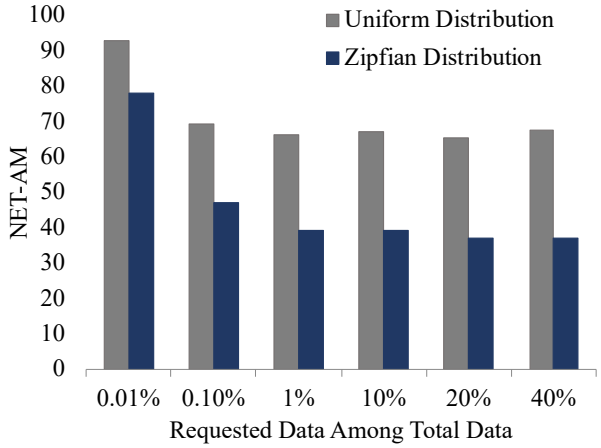


Figure 6.7: NET-AM of Get queries with Uniform and Zipfian KV-pair access hotness distributions

which is 1KB, one data block of 64 KB size is sent to the RegionServer and the data block is rarely used again before it is evicted. If the KV-pair size is smaller, the read amplification can be even worse and it creates even a higher NET-AM.

A heavy network traffic caused by the high NET-AM makes the Get performance sensitive to the available network bandwidth. As shown in Figure 6.8, when network bandwidth between a RegionServer and a storage server decreases, the QPS of Get decreases. We normalized the QPS based on the QPS when the network bandwidth is 2 Mbps. If the available network bandwidth is higher than 100 Mbps, the QPS of Read-Zipfian is explicitly higher than that of Read-Random. With a better read locality, Read-Zipfian has a higher cache hit ratio and thus its performance is better. However, as the bandwidth decreases to lower than 64 Mbps, the QPS of the two workloads are nearly the same and drops drastically. Under certain network conditions, the data being read from the storage server can saturate the network bandwidth and the read performance is decreased dramatically. Therefore, in a compute-storage disaggregated infrastructure, the read performance of HBase can be easily impacted by the available network bandwidth due to the extremely high demand on the I/Os.

Scan In this experiment, we first insert 100 million KV-pairs to an HBase table as the basic dataset for read via YCSB LOAD operations. We use YCSB *workloadb* to generate only Scan requests (the ratio of Get and Put are sets to 0). In each Scan query,

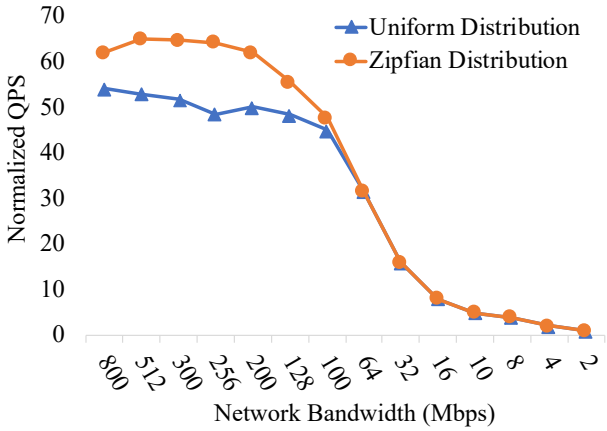


Figure 6.8: Normalized QPS of Get queries based on the QPS when the network bandwidth is 2 Mbps. KV-pairs are accessed with Uniform and Zipfian KV-pair access hotness distributions

it requires N consecutive records as the scan-length. N is configured as 10, 100 and 1000 in different tests. For example, scan-10 refers to the test that, YCSB randomly selects a key as the start-key of a Scan iterator. Then, HBase calls 10 *Next()* to get the next 10 consecutive KV-pairs. We vary the total number of KV-pairs being read by Scan from 0.01% to 40% of the KV-pairs stored in the HBase.

As shown in Figure 6.9, the X-axis is the requested data ratio which varying from 0.01% to 40%, and the Y-axis is the NET-AM of Scan (the total data being read through the network divided by the data being returned to clients). When the scan length is smaller, the NET-AM is higher. The NET-AM of Scan-10 is about 4X and 30x higher than that of Scan-100 and Scan-1000 respectively. With a shorter scan length, the more KV-pairs in the data blocks being read out for Scan queries are not used. For example, if the Scan length is 10 and all 10 KV-pairs are in the same data block, the rest of 54 KV-pairs are not used by the current Scan. It causes a high read amplification. Therefore, the lower the ratio of requested data is, the higher the network traffic will be. Similar to Get, as the requested data ratio increases from 0.01% to 40%, the NET-AM of Scan decreases due to a higher cache hit ratio. In general, with a smaller scan length and lower requested data ratio, the NET-AM is higher.

Scan with filter In some HBase use cases, a client may want to exclude some of the KV-pairs while scanning. A client can set the filter parameter of Scan to specify

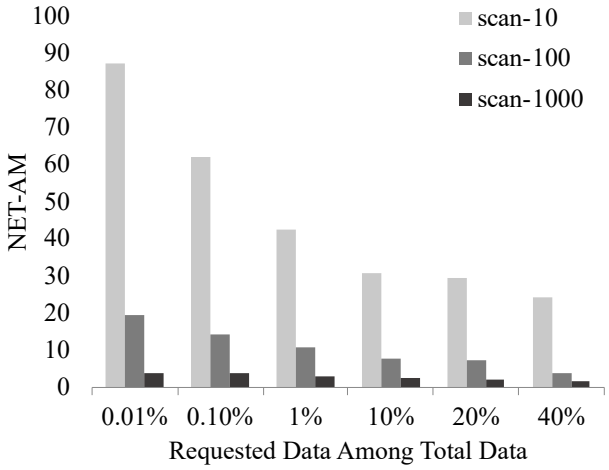
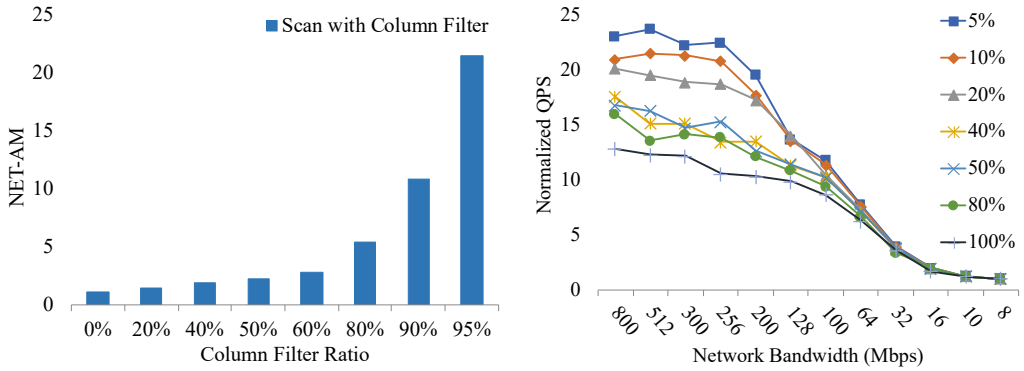


Figure 6.9: NET-AM of scan operation

the filtering condition. For a KV-pair, its key composes of a raw key, column family, column, and time stamp. A client can use the Scan filter to effectively skip some of the KV-pairs and get only the KV-pairs needed. For example, in an HBase table, it has 100 different columns. A client wants to scan the KV-pairs from key A to key B. HBase returns all the KV-pairs that consist a key in the key range [A, B]. If the client only focuses on the KV-pairs from a certain column, for example, Column 2, the client can set the column name as the filter condition. In this situation, only the KV-pairs that consist a key in key range [A, B] and have Column 2 are returned. When using the Scan with filter, the requested data ratio is even smaller. However, all the data blocks that are in the key-range [A, B] need to be transferred from storage servers to the RegionServer. The NET-AM can be more serious than that of a simple scan.

Since YCSB cannot directly generate Scan queries with filter, we developed a new benchmark to issue the scan requests with simple filter conditions. First, 200,000 records with 100 columns are written into HBase. Therefore, the total data size is 20 GB. The column filter ratio varies from 0% to 95% for each test with the same 50 MB requested data. For example, with 90% filter ratio, at least, there will be 500 MB data read out from HDFS and only 50 MB data is requested (i.e., 90% of the KV-pairs are filtered out before returning to the client). Figure 6.10(a) shows that the higher the filter ratio is, the higher the network amplification will be. The reason is that all column data has to



(a) NET-AM of Scan with different column filter ratio

(b) Normalized QPS (based on the QPS when network bandwidth is 8 Mbps) of experiments with different column filter ratios and with different available network bandwidths

Figure 6.10: The NET-AM and QPS results of Scan with column filter

be read out together in the same data blocks to the RegionServer in spite of the filter ratio. The filtering work is done on the RegionServer other than HDFS servers.

Figure 6.10(b) shows the normalized QPS of experiments with different column filter ratios and with different available network bandwidths. When network condition is the same (e.g., network bandwidth is 256 Mbps), Scan with higher filter ratio achieves higher QPS since fewer data will be transmitted. When the network becomes congested (e.g., the network bandwidth drops from 512 Mbps to 8 Mbps), the QPS of Scan with low filter ratios drops quickly.

Conclusion: The physical disaggregation of RegionServers and HDFS servers results in heavy network traffics between them. The read amplification makes it more severe due to the size mismatch of KV-pairs and data blocks. Therefore, the throughput especially the read performance is tremendously affected.

6.5 IS-HBase Design

As discussed in Section 6.4, the read performance of HBase is very sensitive to the available network bandwidth due to its large number of block reads caused by the high read amplification. In this section, we present a new HBase architecture that offloads the function of HFile scanner, called *in-storage scanner*, from RegionServers to storage

nodes. An in-storage scanner follows the concept of in-storage computing, processes the data blocks of HFiles in storage nodes, and only returns the required KV-pairs to the RegionServer. By filtering out the irrelevant data by the in-storage scanner, the new HBase design effectively reduces the network traffic required especially for Get and Scan with filter.

However, several challenges need to be addressed when we deploy the new in-storage scanner based HBase (called *IS-HBase*): 1) How to design a new protocol to achieve the communication between RegionServers and in-storage scanners in different storage nodes? 2) How to handle Scan and Get with in-storage scanner? 3) How to achieve compactions in the new architecture? and 4) How to ensure data consistency and correctness?

In the following subsections, we will first describe the new architecture of IS-HBase. Then, the communication between RegionServers and in-storage scanners is discussed. Next, we will demonstrate the processes to respond to Get and Scan queries as well as how compactions are done. We also address the issue of how to ensure data correctness and consistency.

6.5.1 System Overview

To generalize the compute-storage disaggregated infrastructure for in-storage computing based applications, we have the following assumptions:

- There are two clusters: 1) a compute cluster, and 2) a storage cluster. Each server in the compute cluster has a big memory and powerful CPU. Its local storage is fast (e.g., SSD based) but small in terms of capacity. These compute servers are connected with a high speed network and the network bandwidth of each server is the same. The storage cluster consists of a number of storage servers. Each storage server has enough memory and CPU resources to handle I/O requests. Comparing with the compute server, the memory space of a storage server is limited and the storage servers' CPU is less powerful. Each storage server maintains a number of storage devices (e.g., HDDs or SSDs). The compute cluster and storage cluster are also connected with a high speed network which has a fixed available network bandwidth.

- The storage cluster is deployed similarly to an object storage system which is very popular in distributed file systems like XFS, Lustre, and HDFS. The servers providing the capability of storing data objects are the *storage nodes*. Also, a small number of servers provide metadata service. Metadata servers receive the read and write requests from clients and respond to with the addresses of the storage nodes that store the data (for read requests) or the storage node addresses to write the data (for write requests). Then, the client can directly contact the storage nodes to read and write the objects. When a read/write session is finished, the storage node updates the status of the objects with the metadata servers.

- The compute cluster may run multiple services and HBase is only one of the services. To better utilize the memory/CPU resources, RegionServers of HBase are deployed as virtual machines on a number of physical servers. Therefore, each RegionServer has being allocated its own memory space, network bandwidth, and CPU resources.

- Storage nodes are also connected via networks and the available bandwidth is fixed. Different from in-storage computing modules which are offloaded for different services, in-storage scanners are running in storage nodes as backend demons. Each in-storage scanner has its memory and CPU utilization limits such that the performance of other primary storage I/O requests can be guaranteed.

The architecture of IS-HBase in a compute-storage disaggregated infrastructure is shown in Figure 6.11. In IS-HBase, in order to reduce the network traffic, a RegionServer does not actively read data blocks from storage nodes to its memory. Instead, in most cases, only the requested KV-pairs are sent through the network. To achieve that, each storage node that stores HFiles is deployed with a *in-storage scanner* (ISSN) as the local processing module. Each ISSN maintains the information of the HFile key-range and caches some of the metadata/data blocks which are read out to respond the requests from RegionServers. If there is no read request from any RegionServer, an ISSN uses little CPU resource.

When a RegionServer receives read requests based on the mapping from key-range to HFiles, the RegionServer generates a list of the HFile names that are relevant to this query. Then, the RegionServer queries the information of the storage nodes that store

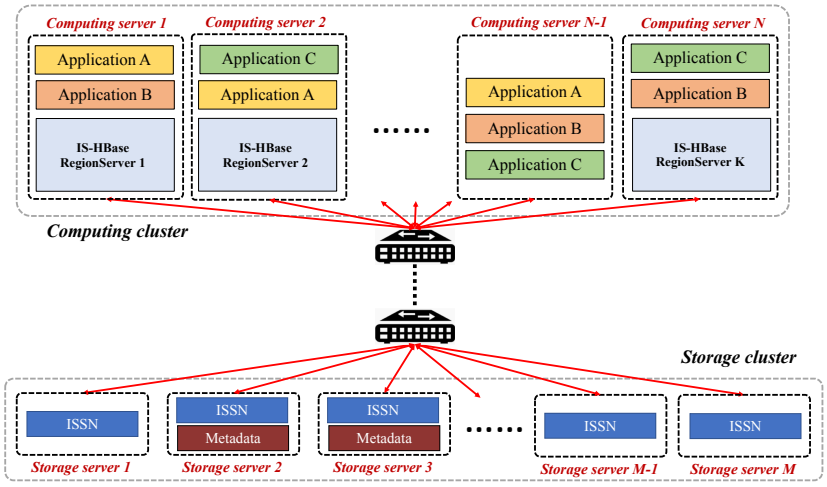


Figure 6.11: The architecture of IS-HBase deployed in a compute-storage disaggregated infrastructure. In this example, there are N compute nodes and M storage nodes. In each compute server, different applications are running, and data are exchanged via a high-speed network. The IS-HBase RegionServers are running as one application in the compute cluster. Each storage node that stores HFiles is deployed with one ISSN, and it manages the HFiles stored in that node. The metadata service of the object storage is running as one service in some of the storage nodes. RegionServer controls the ISSNs to achieve KV-pair reads and compaction.

these HFiles from the metadata service. Next, the RegionServer set up communications with ISSNs whose storage nodes manage the relevant HFiles to search the HFiles in the storage nodes via ISSNs. Afterward, the RegionServer receives the processed results from ISSNs, combines these results, and then returns the final KV-pairs to the client.

For write queries (e.g., Put or Delete), new KV-pairs are accumulated in a Memtable and logged to the WAL. The WAL is stored as objects in the storage nodes. When a Memtable of one column family is full, the KV-pairs are sorted and written out as an immutable HFile to the storage cluster. During this process, a RegionServer generates the HFile name and queries the metadata server of the storage node that stores the HFile. Metadata server selects a storage node to store this HFile based on the load balancing algorithm (e.g., round-robin, or hashing). Then, the RegionServer sends the new HFile to the selected storage node. Next, the storage node confirms the success of HFile write with the metadata server and the metadata server responds to the RegionServer for this successful HFile write. Finally, the RegionServer updates the HFile

information including the HFile name, key-ranges, and other metadata information to the ISSN in that storage node.

In the following subsections, we will introduce how the read (Get and Scan) queries and compaction are handled in IS-HBase in detail. Then, we will discuss the mechanisms to ensure data correctness and consistency.

6.5.2 Scan and Get

In HBase, Get is a special case of Scan which is to only scan the first KV-pair that matches the key being searched. Therefore, we first introduce the process of Scan and then the description of Get.

Scan:

When using Scan API, a client specifies the start-key and a scan filter (if it is needed). HBase returns an Iterator handler to the client and the client calls Next() to scan through the consecutive KV-pairs with keys equal to or larger than the start-key. The scan length is determined by the client. For short scans (e.g., fewer than 10 Next() are called before the client releases the Iterator handler), it is efficient for the RegionServer to collect the KV-pairs from ISSNs and merge the results. Sending a data block (assuming a data block consists of more than 10 KV-pairs) to the RegionServer is wasting in this scenario because a large portion of the KV-pairs in the data block are irrelevant to the current Scan query.

However, when the scan length is long enough (e.g., scan through multiple data blocks), it is more efficient to send data blocks to the RegionServer and construct a two-level heap in the RegionServer. On one hand, a RegionServer can quickly read out the next KV-pair in memory instead of waiting for the round trip network latency (i.e., from the RegionServer to one ISSN and back). On the other hand, if the network condition is good, the latency of sending a data block is close to sending a single KV-pair.

Also, the scan filter condition influences the scan performance. If most of the KV-pairs being scanned are filtered out, it is more efficient to apply the filter in ISSN and only returns the valid KV-pairs to the RegionServer. In this case, filtering is processed in parallel in all relevant ISSNs and the overall performance will be better than filtering

out the KV-pairs at the RegionServer. For example, a client requests the KV-pairs from a specific column and there are hundreds of columns. When Scan is called, in the worst case, more than 99% of the KV-pairs are filtered out. In this case, reading data blocks to the RegionServer can cause a large amount of irrelevant data reads.

Therefore, it is necessary to maintain two different scan mechanisms: 1) legacy scan, and 2) in-storage scan. IS-HBase uses different scan logics based on network condition, scan length, and filter conditions. Supposedly, each data block has N KV-pairs in average, the scan length is L, network available bandwidth is W (bits/s), and the data block size is D bytes. The latency of sending a data block is about $D(N-1)*8/W$. We can base on these parameters and other conditions to design a policy which will decide if it is efficient to send data blocks to the RegionServer for Scan queries or not (discussed in Section 6.6 in detail).

For in-storage scan, we also maintain a two-level heap design which is similar to that of the legacy HBase. Differently, the Level-2 heap is maintained in the RegionServer to combine and sort the KV-pairs from different ISSNs. A RegionServer uses RPC calls to communicate with the ISSNs in different storage nodes. Based on the start-key (the key to begin the scan) and key-range information of HFiles in different storage nodes, the RegionServer selects the storage nodes whose HFile skey-ranges may overlap with the start-key. Then, the RegionServer starts the RPC calls and communicates with the ISSNs in those storage nodes. The start-key, scan filter, and other scan related information are sent to the relevant ISSNs.

At a storage node, when ISSN receives the Scan RPC call from one RegionServer, it starts one HFile scanner for each HFile that satisfies one of the following conditions: 1) The start-key is within the key-range of the HFile, or 2) The start-key is smaller than the first key of the HFile. At the same time, an ISSN creates a Level-1 heap to combine the scanning results from all HFile scanners. The function of an HFile scanner is the same as the HFile scanner in legacy HBase. It reads out the metadata blocks from HFile and searches the data blocks based on the start-key. If the KV-pair exists, it returns the KV-pair to the Level-1 heap. At the same time, the HFile scanner constructs an Iterator so the following KV-pairs can be returned by calling Next(). When the top KV-pair in Level-1 heap is sent to the Level-2 heap at the RegionServer, an ISSN calls Next() of an HFile scanner Iterator whose KV-pair is popped out and insert the new

KV-pair returned by Next() to the Level-1 heap. If the HFile scanner is at the end of the HFile, the scan process for this HFile stops.

Get:

Get is a special case of Scan: return the first KV-pair to the client if the key is the start-key or return Null. Therefore, IS-HBase reuses the processing logic of Scan. Differently, an ISSN checks the top KV-pair in its Level-1 heap. If the key is the same as the start-key, the ISSN returns the top KV-pair to the RegionServer and then clean the HFile scanner and Level-1 heap. Otherwise, ISSN returns Null to the RegionServer. After the RegionServer receives all the results from relevant ISSNs, it compares the timestamp of all valid KV-pairs and returns the latest KV-pair in the Level-2 heap to the client. If all ISSNs return Null, the RegionServer sends Null to the client as the response of the Get query.

Figure 6.12 is one example showing how Get and Scan work in IS-HBase. Assume RergionServer 1 receives a Get query. Since the cache does not have the requested KV-pair, RegionServer 1 issues the request to the ISSNs who manage the HFiles overlapping with the key of this Get. Then, each relevant ISSN constructs a number of HFile scanners in the storage node and searches the key. A KV-pair that matches the key is sent back to the RegionServer 1 and combined in the Level-2 heap. Finally, the KV-pair at the heap top is returned to the client. In the second case, RegionServer 1 finds the KV-pair in the cache, so it directly returns the KV-pair to the client. There is no communication happens between RegionServer 1 and any ISSNs. In the third case, a client issues a Scan query to RegionServer K. RegionServer K locates the HFiles that may contain the KV-pairs whose key equals to or larger than the start-key. Then, it issues the requests to the relevant ISSNs that manage those HFiles. A relevant ISSN constructs a local scanner for each HFile which maintains a Level-1 heap in the scanner. At the same time, RegionServer K constructs a Level-2 heap at compute server and sorts the KV-pairs from different ISSNs in the heap. By calling Next(), the ISSN can continuously get the ordered sequential KV-pairs whose key is equal or larger than the start key.

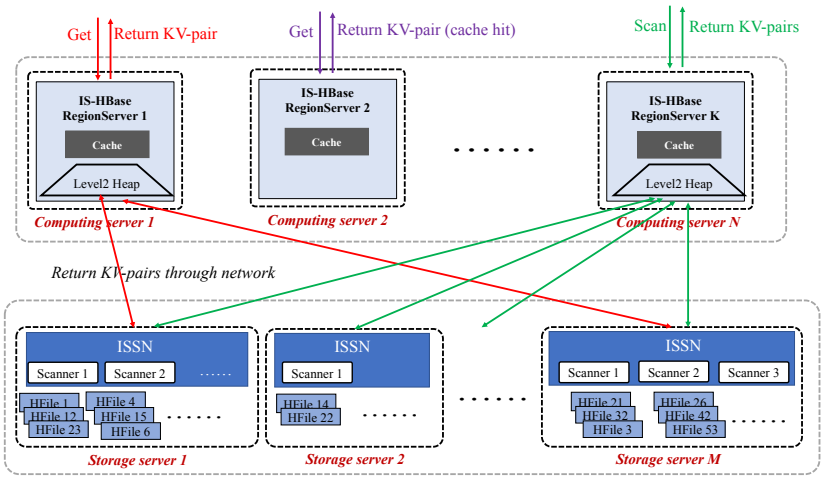


Figure 6.12: One example that should how IS-HBase supports Get and Scan.

6.5.3 In-Storage Compaction

Compaction is the process of merging multiple HFiles into one. During a compaction, deleted KV-pairs are cleaned. In legacy HBase, one RegionServer selects a set of HFiles and combine them into one large HFile. The selected RegionServer relies on its Scan module to read out the KV-pairs from HFiles in order, cleans the invalid (deleted) KV-pairs, and finally sequentially writes the remaining valid KV-pairs to a new HFile. If we do not offload the compaction process in IS-HBase, a compaction may create heavy network traffic to read and write HFiles and cause performance degradation.

In IS-HBase, we design an in-storage compaction mechanism which avoids the heavy traffic by performing the compaction in one of the storage nodes with the help from relevant ISSNs. However, several challenges need to be addressed: 1) Since the HFiles of one RegionServer can be stored in different storage nodes, we need a new communication mechanism for RegionServers and ISSNs; 2) How to achieve compaction without reading data to RegionServers? 3) For load balancing and managing purposes, the storage system has its own policy to allocate the new HFiles. How to decide where the new HFile to be written to?

We propose the following design for in-storage compaction to address the aforementioned challenges. First, if one RegionServer decides to compact several HFiles, based on the compaction selection policy, the RegionServer selects a number of HFiles as the

candidates (called candidate HFiles). Then, the RegionServer queries the storage system metadata services to get the addresses of the storage nodes of the candidate HFiles. Next, the RegionServer creates an empty HFile and writes it to the storage system. The metadata server will return the address of the newly created empty HFile to the RegionServer. At this moment, the RegionServer starts an RPC call to the ISSN (called *compaction-ISSN* during the compaction process) of the storage node where the new empty HFile is stored. The compaction-ISSN will receive the address information of candidate HFiles and the permission to contact all relevant ISSNs in the storage nodes which stores those candidate HFiles.

After the preparation is finished, the compaction-ISSN starts the RPC calls to all the relevant ISSNs (scan-ISSNs) that manage the candidate HFiles. The compaction-ISSN starts a Level-2 heap to aggregate the results from scan-ISSNs and all the scan-ISSNs will construct a Level-1 heap for the candidate HFiles. The compaction-ISSN continues to pick the KV-pairs from the Level-1 heaps in all relevant scan-ISSNs, filters out the invalid KV-pairs, and appends the valid KV-pairs to the pre-stored new HFile. After the compaction is finished. The compaction-ISSN updates the metadata information of the new HFile in the metadata server, and then informs RegionServer the completion of compaction. The RegionServer now can delete all the candidate HFiles from the storage system. To ensure data correctness, during the compaction, if any of the connection is lost or any of the read/write fails, the RegionServer terminates the compaction and restarts the compaction later. One example of in-storage compaction is shown in Figure 6.13.

6.5.4 Data Correctness and Consistency

In legacy HBase, KV-pairs are sorted and stored in the read-only HFiles. For read requests, it checks the KV-pairs in Memtable and cache first. Then, it reads out the data blocks from HFiles to further search the KV-pairs if a cache miss happens in Memtable and cache. If the requested KV-pair does not exist, the RegionServer cannot locate the KV-pair with the key. If the KV-pair has been deleted, the RegionServer will find the key with a special tombstone or the KV-pair is deleted together with the tombstone during a compaction. The read process is executed by the RegionServer. By creating version views (similar to a snapshot of the database when the query is called)

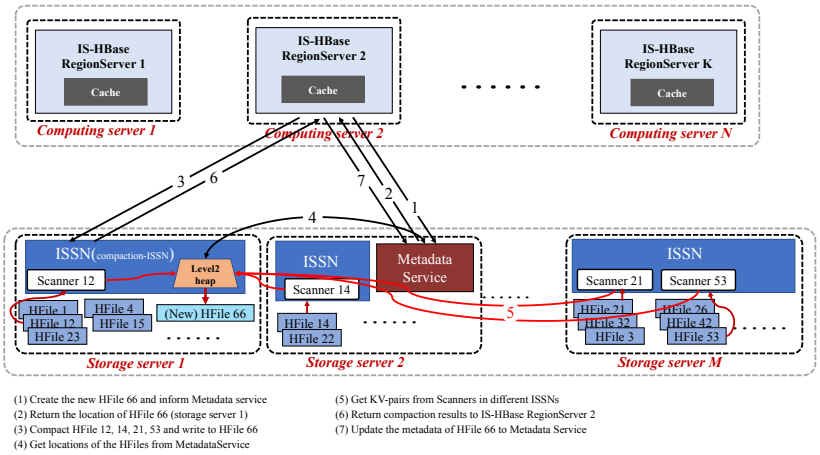


Figure 6.13: One example of in-storage compaction achieved by IS-HBase. RegionServer 2 selects HFile 12, 14, 21, and 53 as candidates to be compacted and writes the valid KV-pairs to HFile 66. The ISSN on Storage Server 2 is selected to execute the compaction which is called compaction-ISSN.

for each read query, it ensures data consistency. By querying the heap in a RegionServer, the RegionServer can always get the correct KV-pairs.

Differently, in IS-HBase, only Level-2 heap is maintained in the RegionServer. Level-1 heaps are created by different ISSNs according to the locations of HFiles. Therefore, several special mechanisms are needed to ensure data correctness and consistency. First, to ensure the correctness of KV-pairs being read out, a RegionServer needs to make sure the RPC connections to the relevant ISSNs are in correct status. Otherwise, the RegionServer returns an exception to the client. Second, it is possible that an HFile is corrupted or an HFile scanner cannot perform correctly. An ISSN will have to report the exception to the RegionServer and the RegionServer needs to decide whether to ignoring the exception, restarting the read, or terminating the read process. Third, it is possible that due to a bad network connection, the RPC process between the RegionServer and an ISSN may be time out. If the RegionServer fails to reconnect to a relevant ISSN for a number of times due to a failed ISSN, the RegionServer needs to fall back to the legacy mode. That is, directly reading out data blocks from the storage nodes and processing the data blocks in the RegionServer without help from ISSNs. If the read queries also fail in legacy mode, the RegionServer returns exceptions to the client.

6.5.5 KV-Pair based Cache

In IS-HBase, since a RegionServer receives only KV-pairs from relevant ISSNs instead of receiving data blocks containing KV-pairs for Get and some Scan queries, the RegionServer can directly cache the KV-pairs in memory. However, this may cause two challenges: 1) How to index and maintain the KV-pairs in cache such that it can support both future Get and Scan queries? and 2) How to identify invalid KV-pairs during a compaction and evict these KV-pairs from cache? To address these two challenges, we propose a new KV-pair based cache for RegionServers.

In legacy HBase, a RegionServer only caches blocks (both metadata blocks and data blocks). During a KV-pair search, a RegionServer reads the correlated metadata blocks (e.g., indexing blocks) to memory such that the RegionServer can identify the sequence numbers of data blocks that are needed to response the query. By searching the sequence numbers in the cache, the RegionServer can decide whether the data block should be read out from the corresponding HFile. With the read-in data blocks, the scan Iterator can find the start key in the data block and get the Next KV-pairs by reading the consecutive KV-pairs.

However, in IS-Hbase, since a RegionServer only receives KV-pairs from relevant ISSNs, the RegionServer does not have the information of data blocks and HFiles. For Get queries, by comparing the KV-pairs in Memtable, KV-pair cache, and the KV-pairs from ISSNs with the same key, the RegionServer can decide which KV-pair is valid. However, for Scan queries, the RegionServer cannot decide if the KV-pair in the cache is the next KV-pair to be read out without comparing the KV-pairs from all Level-1 heaps in relevant ISSNs. Therefore, KV-pair cache is useless for Scan queries if a RegionServer relies on ISSNs to read out KV-pairs directly.

Based on the design of Scan described in Section 6.5.2, a RegionServer reads data blocks from storage nodes to find the KV-pairs for some of the Scan queries and others are all processed by ISSNs via sending KV-pairs. To make the cache consistent, we do not cache data blocks read out by Scan in IS-HBase. A RegionServer always needs to read out the KV-pairs or data blocks from ISSNs during a Scan. The performance of Scan will be lower for some hot key-ranges whose KV-pairs are requested by other Scan queries. The performance of IS-Hbase can be further improved by designing a self-adaptive cache which keeps both KV-pairs and data blocks in memory. We will

discuss this self-adaptive cache in detail in Section 6.6.

The KV-pair cache is achieved as a hash table based LRU cache. The name of the HFile that holding a KV-pair is kept in the value of that KV-pair. For Get queries, by checking the cache, a RegionServer can skip the HFile whose KV-pair is already in the cache. Therefore, the RegionServer only sends the requests to the ISSNs whose HFiles have a cache miss for certain key. This reduces the number of reads at an ISSN side and the required network traffic if these are cache hits. When processing a Scan, the RegionServer bypasses the KV-pair cache and starts the requests to relevant ISSNs as discussed in 6.5.2. During a compaction, a RegionServer evicts the KV-pairs whose HFiles are selected to be compacted. It ensures that the cached KV-pairs are always valid.

6.6 Self-Adaptive Block Cache

One big challenge of in-storage computing design is how to handle the caching issue. The place and the logic of processing data are changed. A new caching scheme is needed to handle different data types at the storage server as well as at the application server.

As we discussed in Section 6.5.5, since IS-HBase gets KV-pairs from ISSNs instead of getting data blocks from storage nodes, maintaining a KV-pair based cache in a RegionServer is a straight forward solution to reduce cache misses. This can reduce the latency of Get queries if there is a cache hit. However, KV cache cannot be used for Scan because a RegionServer cannot decide which KV-pair is the next one. Always reading KV-pairs from storage nodes when responding Scan queries can cause a high performance penalty. First, to fetch the next KV-pair from one storage node, a RegionServer needs to send the request to the storage node instead of directly seeking the next KV-pair in the block cached in memory. Second, if the scan length is long, the overhead of reading data block can be smaller than reading a set of KV-pairs separately. Third, if the workload has very good temporal and key-space locality (i.e., the nearby KV-pairs are accessed in a short period of time later), caching data blocks will have fewer cache misses. Therefore, further optimizing the cache for IS-HBase is essential.

To benefit both Get and Scan queries, we propose a self-adaptive block cache for IS-HBase. It caches both KV-pairs and data blocks. To achieve a high cache space

usage, the cache space boundary between KV-pairs and data blocks are dynamically changing based on the workload variation. We will first introduce the architecture of the proposed self-adaptive block cache. Then, the process of boundary adjusting is discussed.

6.6.1 Architecture of Self-Adaptive Block Cache

In IS-HBase, if a RegionServer only maintains KV-pair cache and always gets KV-pairs from ISSNs, it will have a high overhead when the workloads may have very good key-space locality. For example, when two KV-pairs in the same data block are requested in a short period, in legacy HBase, with the help of block cache, the data block only needs to be read out once. After that, the second access to another KV-pair is a cache hit. While in IS-HBase, two transmissions are needed because only requested KV-pairs are sent to the RegionServer each time. The cache miss will happen for the KV-pairs that are not cached.

Actually, caching has a trade-off between the exploration of key-space locality and the effectiveness of using cache space. Pure block-based caching can maximize potential key-space locality while it will have a high transmission overhead due to the much large data block size and a high cache space usage. Caching only KV-pairs may not be able to take advantage of key-space locality, but uses less caching space. Considering the pros and cons of the two, a design selectively caching blocks or KV-pairs is preferred. However, maintaining a fixed size space of KV cache and block cache cannot adapt to the changing workloads. Therefore, the space partition between KV cache and block cache needs to be dynamically adjusted. There are several challenges that should be addressed in the new cache scheme. First, under which conditions, we need to read data blocks from storage nodes to RegionServers? Second, with limited cache space, how to allocate the cache space to data block caching or KV-pair caching? Third, what the eviction policy is used in the cache?

To address the aforementioned issues, we propose a self-adaptive block cache used in RegionServers to preserve the block key-space locality while trying to consume less cache space and network bandwidth. The architecture of the self-adaptive block cache is shown in Figure 6.14. Instead of caching KV-pairs and blocks independently, it keeps

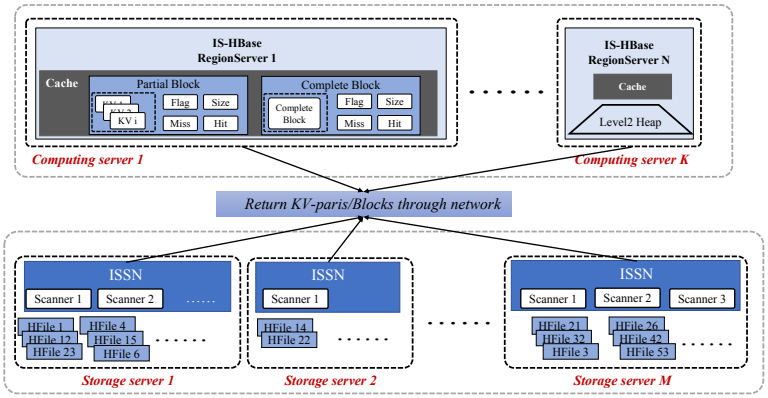


Figure 6.14: The Overview of Self-adaptive Block Cache

linking between cached KV-pairs and blocks with a variable-size block structure. In IS-HBase, a RegionServer maintains a unitary cache space that will be shared by caching both KV-pairs and blocks. Two different formats of blocks are cached, one is a typical complete block and the other is a partial block. The complete block is the data blocks read out from HFiles while the partial block is variable-sized block whose size depends on the cached KV-pairs from the same block. A block sequence number which uniquely identifies the block in IS-HBase is used to index both complete blocks and partial blocks. Besides, we also maintain the following information for each block: 1) a flag to indicate it is a partial block or a complete block, 2) a cache hit counter and a cache miss counter, the sum of them is the total access number to the block, and 3) the size of a block. Cache hit and miss counters are used to measure the locality of a block which will be discussed later.

Although caching KV-pairs and blocks separately is easier to implement, adjusting the cache space boundary between them is challenging. Specifically, if we cache KV-pairs in an independent space without knowing the blocks of the KV-pairs, we are not able to measure the key-space locality of these blocks. It is necessary to maintain the statistics for the partial blocks although only some of its KV-pairs are in the cache. For example, if a partial block has a high number of cache misses, we can allocate more space and accommodate more KV-pairs from the same block to reduce the cache

misses. However, if we cache KV-pairs directly without maintaining the statistics of their corresponding blocks, it is difficult to know if there is such a block suffering from cache misses.

6.6.2 Self-Adaptive Cache Adjustment

We propose a self-upgrade/downgrade adaptive scheme to allocate and reclaim the cache space for partial blocks/complete blocks to achieve a higher caching hit ratio and cache space efficiency. Upgrade is a process used to allocate more cache space for a partial block when it is suffering from cache misses. A large number of cache misses to the same block indicates that the block has a strong key-space locality and need more cache space to accommodate more KV-pairs in the partial block. In certain cases, merely caching individual KV-pairs one by one will may still suffer from compulsory misses (i.e., a miss occurs at the first access). Thus, we make an upgrade by acquiring a complete block to replace that partial block.

Downgrade is a process to evict a complete block and only cache the hot KV-pairs in the block. That is, a complete block contains hundreds of KV-pairs and not all of them need to be kept in cache. In this case, it is more efficient to only cache the hot KV-pairs rather than caching the whole data block. However, identifying hot and cold KV-pairs in a complete block is still an issue. We rely on a hotness sorting method (described in the following subsections) to divide a complete block into two regions: hot region and cold region. During the downgrade process, cold KV-pairs from cold region are evicted, and hot KV-pairs are downgraded to a partial block. In some other cases, if the whole data block is cold, it can be directly evicted from the cache. With this self-upgrade/downgrade scheme, the partition between two kinds of data is dynamically determined. A RegionServer will keep requesting KV-pairs at most times and only acquires a complete block when there is an upgrade.

Self-Upgrade When the miss counter of a partial block reaches to a threshold, this data block will be selected as a candidate to be read out from the storage node and cached in the RegionServer as a complete block. The miss threshold is set to proportional to the amount of network traffic. When the network traffic is heavy, it is better to transmit KV-pairs rather than sending the whole block such that the network traffic can be relieved. In this case, the threshold needs to be high enough to intercept

most dispensable block transmissions. With an upgrade of a partial block, the future cache misses of the data in this block can be effectively reduced. However, since a partial block is smaller than a complete block, some of the other partial blocks or one complete block may need to be evicted to reclaim the space for the new complete block.

We will first check if there is any complete block that is cold enough to be evicted. A cold complete block is a block with a lower total access number (i.e., hit number + miss number) and this number is also lower than the total access counts of some partial blocks which are waiting for an upgrade. If there is no such a cold complete block, a RegionServer will follow an LRU policy to choose some of the partial blocks to evict. The new complete block will inherit the cache hit and miss counters of the original partial block. To exclude the interference of old statistic information (i.e., miss and hit numbers), the access counters of each evicted block will be reset to 0 to ensure the statistic information only reflects a recent trend.

Self-Downgrade This process evicts the cold KV-pairs rom complete blocks and only caches the hot KV-pairs as partial blocks. Downgrade will happen where only a small portion of the KV-pairs in a block is hot and other KV-pairs are rarely accessed. A partial block is created to cache those hot KV-pairs and thus the complete data block is evicted. However, monitoring the hotness of individual KV-pairs has extremely high memory overhead, which cannot be used in IS-HBase. According to [61], hot KV-pairs are usually located closely in the key-space. Therefore, instead of monitoring the hotness of individual KV-pairs, we divide the key-space of KV-pairs in a data block into several regions. KV-pairs in cold regions are evicted from cache.

Monitoring the hotness of KV-pair regions is challenging. In this research, we use a simple method to calculate the hotness of one region, which reflects both recency and frequency. Every T time period, the hotness of one KV-pair region is recalculated. Suppose the hotness of last monitoring period is H^{i-1}, the total access count of this region during the current monitor period is R^i, and the hotness of current monitoring period is H^i. We have $H^i = \alpha \times R^i + (1 - \alpha) \times H^{i-1}$. α is the weight that can reflect the importance of the access count in the current monitoring period. In our design, α is 0.5 and the monitoring period T is 10% of the average data block lifetime in the cache.

When one complete block is selected to be downgraded, we first sort its KV-pair regions based on the hotness in descending order. Counting the hotness from hottest

regions to the cold regions, we select the KV-pair regions that contribute the top 60% of the block hotness to be cached. The rest of the KV-pairs are evicted.

As discussed, both self-upgrade/downgrade are processes used to make a balance between cached KV-pairs (partial blocks) and complete blocks. However, these approaches are only used when there is any partial block or complete block that needs a change. Therefore, a RegionServer still employs an LRU policy as the regular cache replacement policy. Since acquiring a complete block only occurs in upgrade, all regular insertions and evictions are on partial blocks. An insertion will create a new partial block if there is no such a partial block while an eviction may delete a partial block if that block is least recently used.

6.6.3 Get and Scan with Self-Adaptive Block Cache

The logic flows of Get and Scan are shown in Figure 6.15. For a Get, it first checks with the metadata server to find the corresponding block number and then uses this block number to check if there is such a block in the cache. Once there is such a block, no matter a complete block or a partial block, they will use a binary search to find the requested KV-pair. However, unlike a complete block, a partial block may not contain that KV-pair. In this case, the miss counter of that partial block will be increased by 1, and the RegionServer will send a Get request to the relevant ISSN to access that KV-pair as mentioned in Subsection 4.2.2. On the other hand, if there is no such a block in the cache, the RegionServer will also send a Get request with the block number and the requested Key to the relevant ISSN. Once the requested KV-pair has been received, it will be inserted into the cache if there is such a partial block. Otherwise, the RegionServer will first create a partial block with that block number and then insert this KV-pair.

In terms of Scan, as discussed in Subsection 4.2, a Scan can be considered as a sequence of Next() calls. Therefore, a Scan will first do the same operations as Get. If there is no such a block containing the start Key in the cache, it will send a request to the relevant ISSNs with the start Key. After receiving the corresponding KV-pairs, the RegionServer will also insert those KV-pairs into a newly created partial block. However, if that block exists in the cache as a partial block, the partial block cannot serve the Next calls of Scan since a partial block only contains part of the KV-pairs and

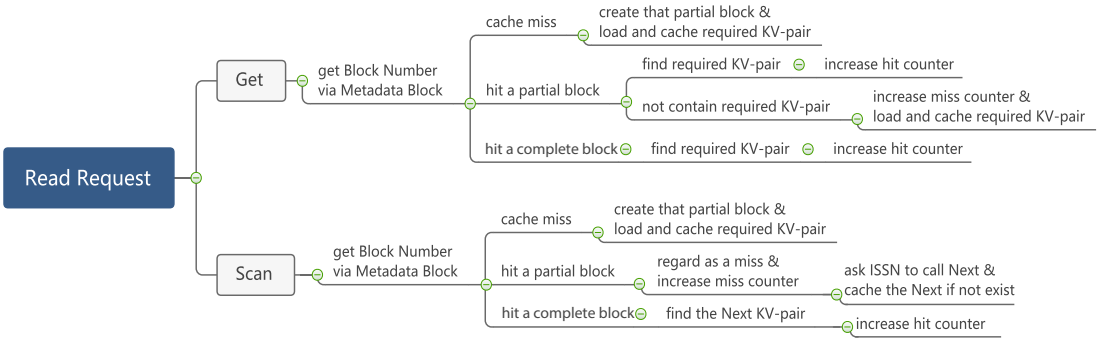


Figure 6.15: Procedure of Read Request

cannot guarantee the next KV-pair is in the partial block as occurs in a complete block.

Thus, if a partial block exists, we do not check that partial block and consider it as a cache miss. If the block which contains the start-key is in the cache, we can scan from the start key and keep requesting the next KV-pair one by one from this block. If this block has been searched through, we need to check the metadata block to get the next block number and do the same operations as the aforementioned.

6.7 Performance Evaluation

In this section, we present a comprehensive evaluation of IS-HBase. To quickly evaluate the concept of IS-HBase, we implement two different systems. We conduct a basic IS-HBbase evaluation in Section 6.7.1, in which we use one RegionServer instance at the storage node as an ISSN to adopt its scanning capability. This can emulate the process of reading KV-pairs and compaction of IS-HBase in a precise way. In these evaluations, we focus on the validation of reducing NET-AM, improving QPS, and reducing latency achieved in IS-HBase. In order to evaluate the design effectiveness of communication protocols of IS-HBase, ISSN, and different cache designs, we develop a simulator called *HBase-sim* and the evaluation results of HBase-sim are discussed in Section 6.7.2. HBase-sim can generate key-value workloads with different features and distributions. Due to the complexity of compaction, the in-storage compaction is not simulated in HBase-sim.

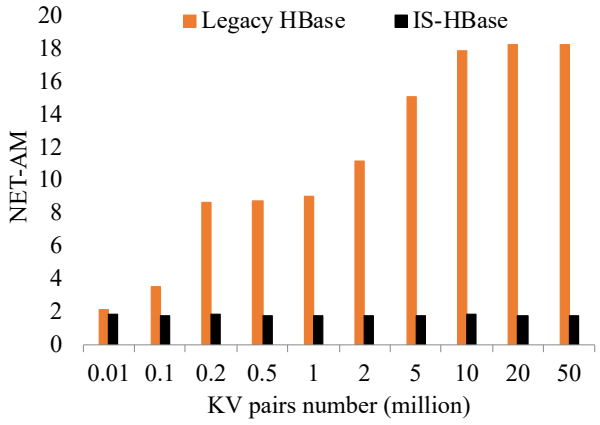


Figure 6.16: The network traffic amplification comparison during loading 50 million KV-pairs between legacy HBase and IS-HBase

6.7.1 Basic IS-HBase Evaluation

In this evaluation, a lightweight instance of RegionServer is deployed at the storage node and it plays like an ISSN performing scanning function, while another copy of RegionServer running on another sever acts as a real RegionServer with its scanning function disabled. The emulated ISSN is responsible for data pre-processing at the storage node, while the latter combines the results and responds to client requests like a RegionServer. We evaluate Get and Scan queries with column filter operations. The size of a KV-pair is 1KB. Note that, the Legacy HBase evaluated with Uniform and Zipfian KV-pair distribution workloads are called Legacy Uniform and Legacy Zipfian respectively. The IS-HBase evaluated with Uniform and Zipfian KV-pair distribution workloads are called IS-HBase Uniform and IS-HBase Zipfian respectively.

First, we evaluate and compare the Get performance of legacy HBase and the proposed IS-HBase. The hotness of the KV-pairs follows either a Uniform or a Zipfian distribution. Before the test, we insert 50 GB KV-pairs (50 million KV-pairs) to the database. We collect the network traffic amplification (i.e., the generated network traffic divided by the total 50 GB data size) of legacy HBase and IS-HBase after finishing the experiments. As shown in Figure 6.16, the amplification of IS-HBase does not change much when the number of KV-pairs being inserted varies from 10,000 to 50 million. The network amplification is caused by flushing and WAL. However, in Legacy HBase,

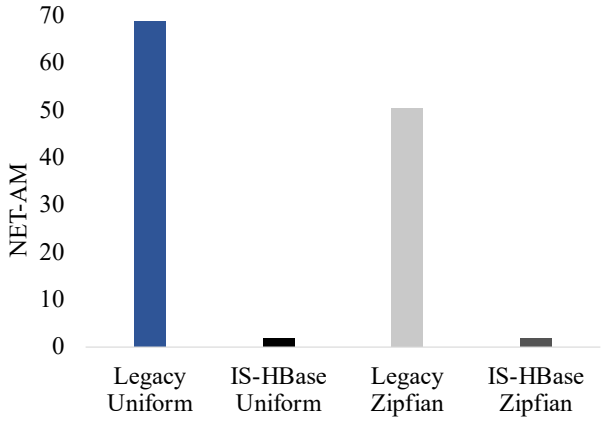
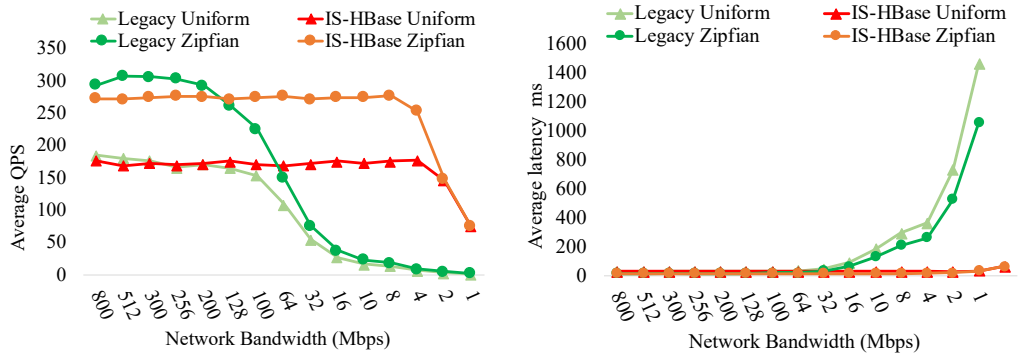


Figure 6.17: The network traffic amplification comparison of Get between legacy HBase and IS-HBase

during insertions, HFiles are continuously read out to a RegionServer to be compacted and written back as new HFiles. It causes a serious amplification. As more KV-pairs are inserted, the amplification varies from 2 to 18. IS-HBase is able to execute the compactions between ISSNs and avoids the additional network traffic.

After 50 million of KV-pairs are inserted to HBase, we use YCSB to issue the Get queries with both Uniform and Zipfian distributions. The read amplification comparison between Legacy HBase and IS-HBase is shown in Figure 6.17. By pre-processing the data blocks locally, IS-HBase can obtain 95% and 97% of network traffic reduction with Uniform and Zipfian distribution workloads respectively. The network traffic reductions are irrelevant to the network condition.

We compare the QPS of Get queries when the network bandwidth varies in Figure 6.18(a). By setting the Open Vswitch, we vary the available network bandwidth between the RegionServer and the Storage node from 1 Mbps to 800 Mbps. As shown in Figure 6.18(a), when the bandwidth becomes lower than 128 Mbps, the QPS of legacy HBase drops quickly and is much lower than that of IS-HBase. We can speculate that, the processing capability of a RegionServer for certain workloads is about 128 Mbps. When network bandwidth is higher than 128 Mbps, the processing capability of RegionServer constrains the performance of HBase. When the bandwidth is lower than 128 Mbps, the network becomes a bottleneck. In this situation, since IS-HBase processes the data blocks in the storage node and only the requested KV-pairs are sent through the network,



(a) The QPS variation as the network bandwidth changes

(b) The query response latency variation as the network bandwidth changes

Figure 6.18: The QPS and latency of Get variations when the network bandwidth varies

the QPS of IS-HBase does not have any explicit degradation. Only when the network is extremely limited (e.g., the network bandwidth is lower than 4 Mbps), the QPS of IS-HBase decreases.

The average latency of Get queries is shown in Figure 6.18(b). As the available network bandwidth decreases from 800 Mbps to 1 Mbps, the average latency of IS-HBase remains at around 24 ms without any significant increases. In legacy HBase cases, however, the average latency starts increasing dramatically from 24 ms after the available bandwidth dropping below 128 Mbps. When the available bandwidth drops from 128 Mbps to 1 Mbps, the average latency of HBase increases to more than 50 times, while the ones of IS-HBase only increase about 3 times. As we can see, processing data blocks in storage nodes can significantly reduce the required network bandwidth and avoid explicitly increased latency under limited network conditions. When the network bandwidth is low, IS-HBase can still achieve a high performance. When the network condition is good, the performance of HBase is close to that of IS-HBase. The results are similar in both Uniform and Zipfian workloads. Differently, the performance of legacy HBase and IS-HBase is usually lower in the Uniform workloads due to fewer cache hits.

To evaluate the performance of Scan queries with different filtering ratios, we first insert a big table to HBase. The total data is 20 GB with 200,000 rows and 100 columns (20 million KV-pairs in total). For one Scan query, it scans from the first KV-pair of a row to its end (100 consecutive KV-pairs in total) and only returns the KV-pairs from

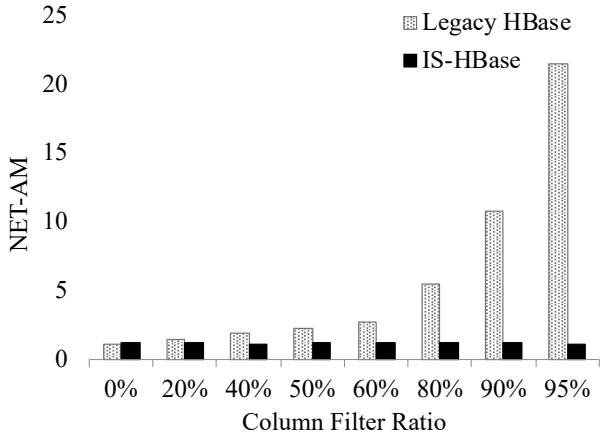


Figure 6.19: The NET-AM variations when different Scan filter ratios are used.

some of the columns according to the filter requirement. We request 20 MB data (20,000 KV-pairs) in each test with the column filter ratio varying from 0% to 95%. If the filter ratio is 95%, only 5% of the KV-pairs being scanned in a Scan query are returned to client and 4,000 Scan queries are issued. If the filter ratio is 0%, all KV-pairs from one row are returned (100 KV-pairs) for one Scan query and 200 Scan queries are issued. We measure the NET-AM in this experiment

As shown in Figure 6.19, the X-axis shows different filter ratios and Y-axis is the obtained NET-AM. Instead of sending all the data blocks to the RegionServer, the ISSN at storage node processes the data blocks locally and only returns the requested KV-pairs from a certain column to HBase. Therefore, the NET-AM of IS-HBase is much lower than that of legacy HBase. Moreover, the network amplification factor is almost stabilized around 1.7 when the column filter varies.

6.7.2 HBase-sim Evaluation

We implemented the HBase-sim that simulates the functionality of a storage node, RegionServer, ISSN, and different cache designs. The simulator is deployed on a Dell PowerEdge R430 server with a 2.40GHz Intel Xeon with 24-core CPU and 64 GB memory. We run each test 3 times and present the average results.

Real-world workloads can be significantly different from each other. To cover different situations, we summarize several factors that lead to these differences, and generate

workloads with different configurations of these factors. The IS-HBase design with self-adaptive block cache consists of multiple components and its performance is influenced by the factors such as network condition and KV access distributions. To test the effectiveness of each component, we run a series of studies that compare the effectiveness of different components. Finally, we compare our IS-HBase with self-adaptive block cache (IS-HBase-SA) with the legacy HBase (HBase-B). Here the legacy HBase refers to the HBase with only a block cache. It can only transmit blocks since it does not have ISSNs. On the other hand, all IS-HBase variations is equipped with ISSNs. Thus, they are able to transmit both KV-pairs and blocks. Now we have proved the NET-AM reduction in the previous section. Thus, we further investigate the QPS improvement with the self-adaptive cache design.

In our experiments, we focus on the interaction between a single RegionServer and a single Storage Node. The reason we use a single RegionServer is that each RegionServer is responsible for only a certain region (i.e. a specific range of keys). Therefore, the processing of requests is independent among RegionServers. Thus, there is no need to simulate all the RegionServers. The HFile stored in the Storage Node is 1 GB which consists of 16,384 blocks. The block size is configured as 64 KB.

Workload Generation

We also generated several workloads as benchmarks in HBase-sim. For Get queries, the key with different access probabilities are generated by the benchmark. For Scan queries, the start key, Scan length, and filter condition follow different distributions. In each test, the benchmark issues 1.5 million queries.

During the generation of a workload, the following factors are taken into account:

- *KV-pair Size*: As the KV-pair size affects the efficiency of sending data from a storage node to the RegionServer, we tested both small and large KV-pair sizes.
- *Network Condition*: Different available network bandwidths lead to different latencies when transmitting the same amount of data. We run the test with both fixed and dynamic available network bandwidths.
- *KV-pair Access Distribution*: KV-pair access distribution affects both KV-pair

hotness and block hotness. To fully evaluate the system performance under different data locality patterns, we test HBase-sim with different distributions of KV-pair access probability. We apply both uniform and Zipfian distributions to different workloads.

- *Scan Length and Filter Ratio*: Both the scan length and filter ratio have an explicit impact on the effectiveness of deploying ISSNs (i.e., transmitting KV-pairs vs. blocks). When the filter ratio is very high (e.g., higher than 99%), only a small number of KV-pairs in the block are accessed when calling Next() by an Iterator. In this case, it is more efficient to filter out the requested KV-pairs directly from ISSNs instead of reading complete data blocks. Moreover, a high filter ratio can lead to "partial hotness" in a block, which means only a small portion of KV-pairs are accessed more frequently.

- *Get-Scan Ratio*: User requests can be Get-intensive or Scan-intensive which can affect KV-pairs/Blocks cache partition. In our simulation, different Get and Scan query ratios are evaluated. Typically, the ratio between Get-Scan is 92:8 [41, 61]. In our simulation, we dynamically adjust the Get-Scan ratio while ensuring the overall ratio remains 92:8.

Each of the five factors above has various configurations. We set a default configuration for each factor. If not specified in the tests, the default configurations are adopted. Specifically, the default KV size is 64 bytes which is considered to be a small one. The default network available bandwidth follows a dynamic pattern. Since 512 Mbps is already high enough as shown in Figure 6.10(a), we generate the network bandwidth using a Sine function whose minimum value is 0 bits/s, the maximum value is 512 Mbps, and minimum positive period is 300 requests. The default KV-pair access distribution is Zipfian, as it is much more often than a uniform distribution. The default Scan length is dynamically changed and its average value is 30. The default scan filter ratio is 0% (no filtering) and the default Get-Scan ratio is 92:8 [41, 61].

To comprehensively study the system performance under different scenarios, we conducted several ablation and independent studies. In the ablation study, we analyze the components in our IS-HBase design with self-adaptive block cache and test its performance with other baselines. After that, we run a set of independent studies using

both the legacy HBase and our IS-HBase with a self-adaptive block cache. In each independent study, we fix four factors listed above to default values while test different settings of the remaining factor.

Ablation Study

We use the legacy HBase as the baseline, in which only blocks are transmitted. We have also included several IS-HBase variations. More specifically, we summarize our design as an integration of the following components:

- *ISSN Component*: An ISSN can carry out the scanning function in a storage node. It has the ability to scan HFiles and return the requested KV-pairs or blocks. If an ISSN is the only component used, it will always transmit the requested KV-pairs. If other components (as described below) are included, an ISSN may send either KV-pairs or data blocks.

- *Network-Bandwidth-Aware Component*: The system switches between "transmitting KV-pairs" and "transmitting blocks" according to the network condition if this component is included. Intuitively, when the available network bandwidth is low, it tends to transmit only KV-pairs.

- *Inter-Block Hotness-Aware Component*: The cache replacement policy of a RegionServer for complete blocks is a priority-based and the priority is based on the access frequencies of blocks.

- *Intra-Block Hotness-Aware Component (a.k.a. complete block down gradation component*): A complete block will be discarded and only its hot KV-pairs be cached to a partial block if only a small portion of KV-pairs in the block is accessed frequently.

- *Partial-Block Upgrade Component*: A partial block will be upgraded to a whole block in the cache if the miss number of this partial block is high. Together with the complete-block downgrade component above, we implemented the proposed adaptive partial/complete block ratio based on the block access locality.

Therefore, in our simulation, we compare the performance of the following 5 different HBase setups based on the implementation of these components in the system.

1. **HBase-B**: This is the legacy **HBase** with only **B**lock-based cache and a LRU cache replacement policy in a RegionServer. For both Get and Scan requests, the RegionServer always reads data blocks to its block cache. This is considered to be the setup of legacy HBase.

2. **IS-HBase-K**: This is **IS-HBase** with only **K**V-pair cache and a LRU cache replacement policy is used in a RegionServer. For both Get and Scan requests, a RegionServer always reads KV-pairs to its cache. The KV-pair cache is only used in Get queries. For Scan queries, it does not check the KV-pair cache, instead, it requests KV-pairs from an ISSN directly. This is a basic version of **IS-HBase** to evaluate the efficiency of an ISSN.

3. **IS-HBase-KB**: This is **IS-HBase** with a separated **K**V-pair cache and a **B**lock cache in each RegionServer. Both caches use the LRU cache replacement policy. The KV-pair and block caches have the same amount of memory space (i.e., fixed with half of the memory capacity). The transmitting decision is based on the query type. For Get queries, the KV-pair cache is used, and it will always transmit KV-pairs. However, for Scan queries, the block-based cache is used, and it will always transmit blocks. This is a baseline version for comparing with the self-adaptive cache design.

4. **IS-HBase-KB-NET**: The cache design of IS-HBase-KB-NET is the same as IS-HBase-KB. The only difference is that IS-HBase-KB-NET takes available network bandwidth into consideration. For Get queries, it will only transmit KV-pairs. But for Scan queries, it will transmit KV-pairs when the available network bandwidth is lower than θ and transmits blocks otherwise.

5. **IS-HBase-SA**: This is **IS-HBase** with **S**elf **A**daptive block cache in a RegionServer. The partial block cache uses an LRU policy while the complete block cache applies a priority-based cache replacement policy. The partition between partial and complete block cache is determined by the partial-block upgrade component and the complete-block downgrade component.

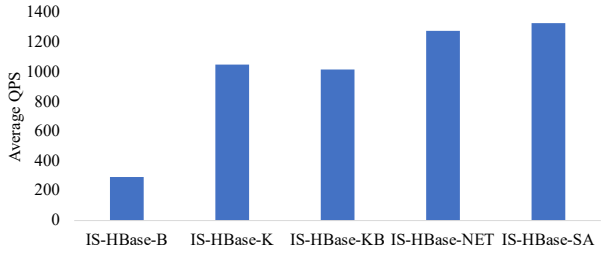


Figure 6.20: The average QPS of different HBase designs

Evaluation Results

We use the aforementioned configurations to evaluate different HBase designs. As Figure 6.20 shows, the average QPS in legacy HBase-B is 292.53 which is the lowest among all the schemes. The reason is that it always transmits blocks instead of KV-pairs. When only a small number of KV-pairs is requested, this can lead to a large overhead, especially when the available network bandwidth is low. When the KV-pair cache is enabled, the transmitting overhead typically decreases significantly. Our results show that the average QPS in IS-HBase-K is 1053.03 which is 3.60 times as large as that of the legacy HBase-B. In terms of IS-HBase-KB, although the self-adaptive cache is adopted, it does not have a smart transmitting strategy. The average QPS in this configuration is 1018.28. Sometimes when the available network bandwidth is low or the scan length is very short, it is more beneficial to transmit only the requested KV-paris rather than the complete blocks for Scan queries. The IS-HBase-KB-NET is a smarter one and it uses both the self-adaptive cache and takes the available network bandwidth into account. Its average QPS is 1280.49. Finally, in our proposed IS-HBase-SA design, all these optimization methods are adopted. Besides, it is self-adaptive and will automatically assign the cache portions of partial and complete blocks. The average QPS is 1330.13 which is slightly better than that of IS-HBase-SA.

Independent Study

We run 5 groups of experiments to compare the impact of each design factor on HBase-B and IS-HBase-SA including: 1) **KV-pair size**: the key size of a KV-pair is configured

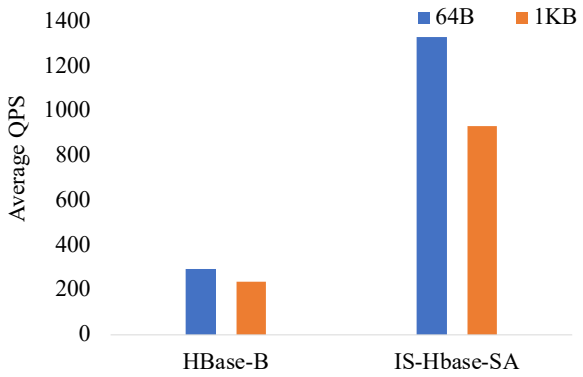


Figure 6.21: The average QPS in the independent study on KV-pair size

as 8 bytes while the value size is set to either 56 bytes or 1016 bytes (i.e., the total KV-pair size is 32 bytes or 1024 bytes respectively); 2) **network condition**: the largest available network bandwidth is set to be 512 Mbps. We test several different available network bandwidths including 1 Mbps, 2 Mbps, 4 Mbps, 8 Mbps, 16 Mbps, 32 Mbps, 64 Mbps, 128 Mbps, 256 Mbps and 512 Mbps; 3) **KV-pair access distribution**: For Get queries, the keys are generated using both uniform and Zipfian distributions. For Scan queries, the start key is generated using the same way as Get queries, while the scan length is set to 30; 4) **Scan filter ratio**: The filter ratio is set to one of the following: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90%; 5) **Get-Scan ratio**: During different periods of time, the ratio changes betwen 0:8 and 184:8. We keep the average Get-Scan ratio as 92:8.

Evaluation Results

Figure 6.21 shows the average QPS of the legacy HBase-B and IS-HBase-SA designs under different KV sizes. As we can see, both of them perform better when the KV-pair size is smaller. This is because ISSNs transmits less amount of data when the number of requested KV-pairs is the same. Suppose that the block size is 64KB, and in a Scan query 10 KV-pairs in a block is needed. The system decides to transmit KV-pairs instead of blocks. If the average KV-pair size is 1KB, 10KB data will be transmitted. However, if the average KV-pair size if 64B, only 640B data will be transmitted.

Figure 6.22 shows the average QPS of the legacy HBase-B and IS-HBase-SA designs under different available network bandwidths. Generally, the throughputs of both

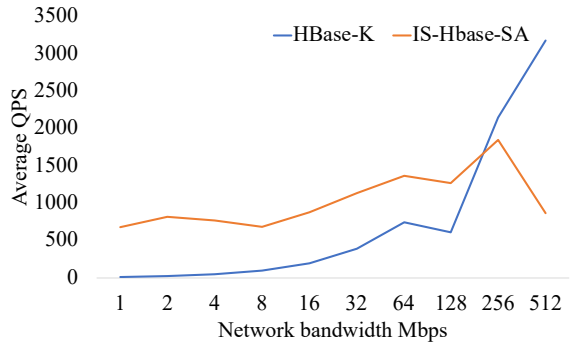


Figure 6.22: The average QPS in the independent study on network condition

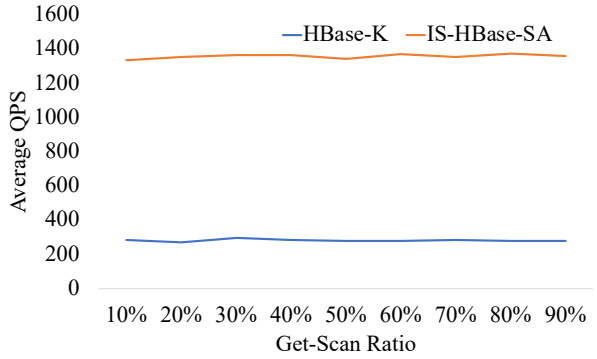


Figure 6.23: The average QPS in the independent study on Get-Scan ratio

designs increase when the available network bandwidth increases. The average QPS of HBase-B is roughly linearly proportional to the available network bandwidth, as it always transmits data blocks. On the other hand, in the IS-HBase-SA design, the system read KV-pairs to memory as partial blocks first and upgrade some of the partial blocks to complete ones as the available bandwidth increases. The QPS of IS-HBase-SA is higher than HBase-B when the network bandwidth is under 256 Mbps as shown in Figure 6.22. If the network condition is very good, the latency of transmitting a block is not high, while sending multiple KV-pairs needs several RPCs. Therefore, it is likely that transmitting a whole block is faster in this case. That is the reason the legacy design performs well with very high available network bandwidth.

The average QPS of the legacy HBase-B and IS-HBase-SA under different Get-Scan ratios are shown in Figure 6.23. Both designs keep a stable throughput performance no

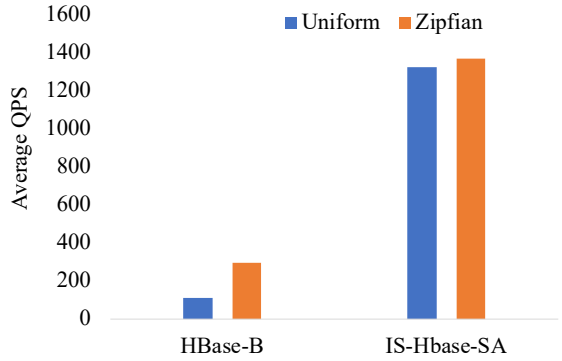


Figure 6.24: The average QPS in the independent study on Key Access Distribution

matter what the Get-Scan ratio is. The average QPS of IS-HBase-SA is much better than that of HBase-B. Always reading complete blocks ensures a stable performance of HBase-B. Whether a KV-pair or a whole block is needed, HBase-B keeps read blocks to the RegionServer. For IS-HBase-SA, it adaptively adjusts the cache space partition between the partial and complete blocks and is able to adapt to the Get-Scan ratio variations.

Figure 6.24 shows the average QPS of the legacy HBase-B and IS-HBase-SA under different KV-pair access patterns. As we can see, both designs have better performance with Zipfian distributions since there is more data locality.

Figure 6.25 shows the average QPS of the legacy HBase-B and IS-HBase-SA designs under different filter ratios. The legacy HBase-B does not consider the filter ratio and always takes the whole blocks as a unit; thus the filter ratio has no impact on its performance. On the other hand, a low filter ratio decreases the benefit of transmitting only the required KV-pairs since more KV-pairs in a block are needed. Therefore, as the filter ratio decreases, the performance of IS-HBase-SA slightly goes down.

The ablation study illustrates the effectiveness of components in IS-HBase-SA. Moreover, the independent studies show the impact of different design factors and the proposed IS-HBase-SA performs well in most cases. Specifically: 1) Smaller KV-pair size is better for the throughput when taking the advantage of ISSNs; 2) Higher available network bandwidth makes it faster to transmit the same amount of data, and the IS-HBase-SA performs well especially when the available network bandwidth is low; 3)

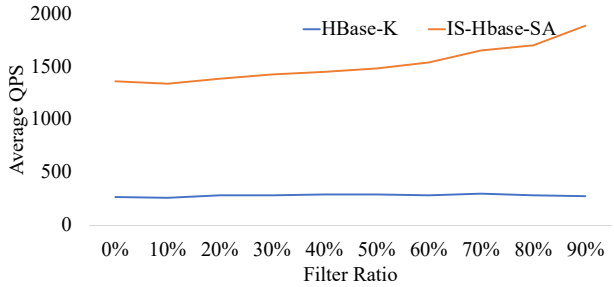


Figure 6.25: The average QPS in the independent study on Filter Ratio

IS-HBase-SA is able to maintain a stable performance under different Get-Scan ratio; 4) The throughput of HBase, whether attached with ISSNs or not, increases with more data locality; 5) The throughput improvement is more significant when the filter ratio is high. It is because only a small portion of data is really needed in this case, and the overhead of transmitting blocks instead of KV-pairs is much higher.

6.8 Conclusion and Future Work

In this research, we investigate the value, challenges, feasibility, and potential solutions of using in-storage computing to improve the performance of HBase in a compute-storage disaggregated infrastructure. The potential of using in-storage computing and the effectiveness of using ISSNs in storage nodes for HBase is demonstrated. We first conduct a set of experiments to investigate the performance of HBase in a compute-storage disaggregated infrastructure. We find that when the available network bandwidth is small, the performance of HBase is very low which is mainly caused by a high NET-AM. Based on these observations, we propose a new HBase architecture extended from the in-storage computing concept called IS-HBase. IS-HBase runs an ISSN in each storage node and an ISSN can process the queries of Get and Scan in the storage node. Only useful KV-pairs are sent through the network to RegionServers. The read performance of IS-HBase is further improved by conducting a self-adaptive block cache in RegionServers, which addresses the caching issue of in-storage computing. IS-HBase is a good example that demonstrates the potential of adopting in-storage computing for other data-intensive distributed applications. In our future work, we will further investigate

the opportunities, challenges, and effectiveness of using in-storage computing for other applications to improve performance such as machine learning services

Chapter 7

Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook

7.1 Introduction

In current IT infrastructure, persistent key-value stores (KV-stores) are widely used as storage engines to support various upper-layer applications. The high performance, flexibility, and ease of use of KV-stores have attracted more users and developers. Many existing systems and applications like file systems, object-based storage systems, SQL databases, and even AI/ML systems use KV-stores as backend storage to achieve high performance and high space efficiency [192, 193, 194, 46].

However, tuning and improving the performance of KV-stores is still challenging. First, there are very limited studies of real-world workload characterization and analysis for KV-stores, and the performance of KV-stores is highly related to the workloads generated by applications. Second, the analytic methods for characterizing KV-store workloads are different from the existing workload characterization studies for block

storage or file systems. KV-stores have simple but very different interfaces and behaviors. A set of good workload collection, analysis, and characterization tools can benefit both developers and users of KV-stores by optimizing performance and developing new functions. Third, when evaluating underlying storage systems of KV-stores, it is unknown whether the workloads generated by KV-store benchmarks are representative of real-world KV-store workloads.

To address these issues, in this research, we characterize, model, and benchmark workloads of RocksDB (a high-performance persistent KV-store) at Facebook. To our knowledge, this is the first study that characterizes persistent KV-store workloads. First, we introduce a set of tools that can be used in production to collect the KV-level query traces, replay the traces, and analyze the traces. These tools are open-sourced in RocksDB release [195] and are used within Facebook for debugging and performance tuning KV-stores. Second, to achieve a better understanding of the KV workloads and their correlations to the applications, we select three RocksDB use cases at Facebook to study: 1) UDB, 2) ZippyDB, and 3) UP2X. These three use cases are typical examples of how KV-stores are used: 1) as the storage engine of a SQL database, 2) as the storage engine of a distributed KV-store, and 3) as the persistent storage for artificial-intelligence/machine-learning (AI/ML) services.

UDB is the MySQL storage layer for social graph data at Facebook, and RocksDB is used as its backend storage engine. Social graph data is maintained in the MySQL tables, and table rows are stored as KV-pairs in RocksDB. The conversion from MySQL tables to RocksDB KV-pairs is achieved by MyRocks [196, 193]. ZippyDB is a distributed KV-store that uses RocksDB as the storage nodes to achieve data persistency and reliability. ZippyDB usually stores data like photo metadata and the metadata of objects in storage. In this research, the workloads of ZippyDB were collected from shards that store the metadata of an object storage system at Facebook (called *ObjStorage* in this research). The key usually contains the metadata of an ObjStorage file or a data block, and the value stores the corresponding object address. UP2X is a special distributed KV-store based on RocksDB. UP2X stores the profile data (e.g., counters and statistics) used for the prediction and inferencing of several AI/ML services at Facebook. Therefore, the KV-pairs in UP2X are frequently updated.

Based on a set of collected workloads, we further explore the specific characteristics

of KV-stores. From our analyses, we find that 1) read dominates the queries in UDB and ZippyDB, while read-modify-write (Merge) is the major query type in UP2X; 2) key sizes are usually small and have a narrow distribution due to the key composition design from upper-layer applications, and large value sizes only appear in some special cases; 3) most KV-pairs are cold (less accessed), and only a small portion of KV-pairs are frequently accessed; 4) Get, Put, and Iterator have strong key-space localities (e.g., frequently accessed KV-pairs are within relatively close locations in the key-space), and some key-ranges that are closely related to the request localities of upper-layer applications are extremely hot (frequently accessed); and 5) the accesses in UDB explicitly exhibit a diurnal pattern, unlike those in ZippyDB and UP2X, which do not show such a clear pattern.

Benchmarks are widely used to evaluate KV-store performance and to test underlying storage systems. With real-world traces, we investigate whether the existing KV benchmarks can synthetically generate real-world-like workloads with storage I/Os that display similar characteristics. YCSB [197] is one of the most widely used KV benchmarks and has become the gold standard of KV-store benchmarking. It provides different workload models, various query types, flexible configurations, and supports most of the widely used KV-stores. YCSB can help users simulate real-world workloads in a convenient way. However, we find that even though YCSB can generate workloads that have similar key-value (KV) query statistics as shown in ZippyDB workloads, the RocksDB storage I/Os can be quite different. This issue is mainly caused by the fact that the YCSB-generated workloads ignore key-space localities. In YCSB, hot KV-pairs are either randomly allocated across the whole key-space or clustered together. This results in an I/O mismatch between accessed data blocks in storage and the data blocks associated with KV queries. Without considering key-space localities, a benchmark will generate workloads that cause RocksDB to have a bigger read amplification and a smaller write amplification than those in real-world workloads.

To develop a benchmark that can more precisely emulate KV-store workloads, we propose a workload modeling method based on the hotness of key-ranges. The whole key-space is partitioned into small key-ranges, and we model the hotness of these small key-ranges. In the new benchmark, queries are assigned to key-ranges based on the distribution of key-range hotness, and hot keys are allocated closely in each key-range.

In our evaluation, under the same configuration, YCSB causes at least 500% more read-bytes and delivers only 17% of the cache hits in RocksDB compared with real-world workloads. The workloads generated by our proposed new benchmark have only 43% more read-bytes and achieve about 77% of the cache hits in RocksDB, and thus are much closer to real-world workloads. Moreover, we use UDB as an example to show that the synthetic workloads generated by the new benchmark have a good fit of the distributions in key/value sizes, KV-pair accesses, and Iterator scan lengths.

This chapter is organized as follows. First, we introduce RocksDB and the system background of three RocksDB use cases in Section 7.2. We describe our methodology and tools in Section 7.3. The detailed workload characteristics of the three use cases including the general query statistics, key and value sizes, and KV-pair access distributions are presented in 7.4, 7.5, and 7.6 respectively. In Section 7.7, we present the investigation results of the storage statistics of YCSB, and describe the proposed new modeling and benchmarking methods. We also compare the results of YCSB with those of the new benchmark. Related work is described in Section 7.8, and we conclude this research in Section 7.9.

7.2 Background

In this section, we first briefly introduce KV-stores and RocksDB. Then, we provide background on three RocksDB use cases at Facebook, UDB, ZippyDB, and UP2X, to promote understanding of their workloads.

7.2.1 Key-Value Stores and RocksDB

KV-store is a type of data storage that stores and accesses data based on {key, value} pairs. A key uniquely identifies the KV-pair, and the value holds the data. KV-stores are widely used by companies as distributed hash tables (e.g., Amazon Dynamo [45]), in-memory databases (e.g., Redis [198]), and persistent storage (e.g., BigTable [199] from Google and RocksDB [200, 42] from Facebook).

RocksDB is a high-performance embedded persistent KV-store that was derived from LevelDB [43] by Facebook [200, 42] and was optimized for fast storage devices such as Solid State Drives. RocksDB is also used by many large websites, like Alibaba [201],

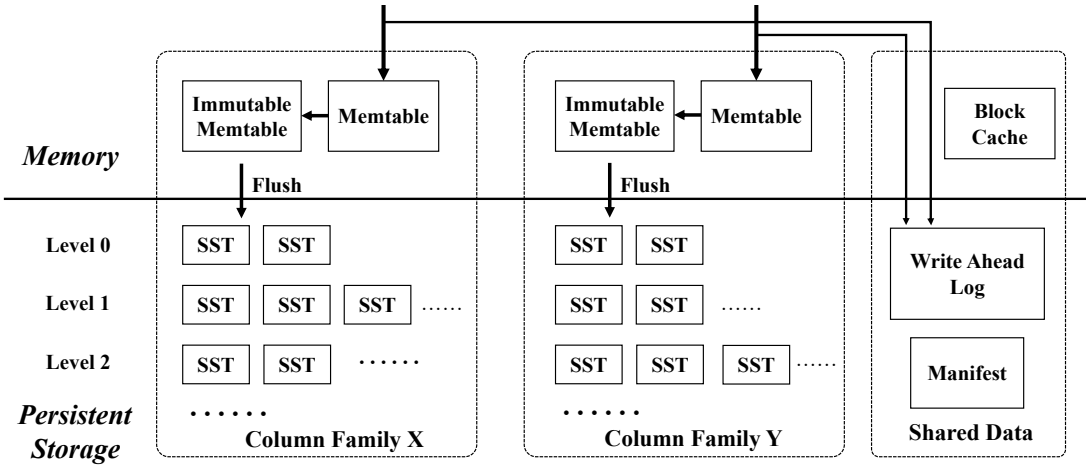


Figure 7.1: The basic architecture of RocksDB.

Yahoo [202], and LinkedIn [203]. At Facebook, RocksDB is used as the storage engine for several data storage services, such as MySQL [196, 193], Laser [204], Cassandra [46], ZippyDB [205], and AI/ML platforms.

RocksDB supports KV interfaces like Get, Put, Delete, Iterator (scan), SingleDelete, DeleteRange, and Merge. Get, Put, and Delete are used to read, write, and delete a KV-pair with certain key respectively. Iterator is used to scan a set of consecutive KV-pairs beginning with a *start-key*. The scan direction can be either forward (calling Nexts) or backward (calling Prevs). SingleDelete can only be used to delete a KV-pair that has not been overwritten [206]. DeleteRange is used to delete a range of keys between [start, end) (the end-key is excluded from the deletion). RocksDB encapsulates the semantics for read-modify-write into a simple abstract interface called Merge [207], which avoids the performance overhead of a random Get before every Put. Merge stores the delta of the write to RocksDB, and these deltas can be stacked or already combined. This incurs a high read overhead because a Get to one key requires finding and combining all the previously stored deltas with the same key inserted by a Merge. The combine function is defined by users as a RocksDB plugin.

RocksDB adopts a Log-Structured Merge-Tree [208] to maintain the KV-pairs in persistent storage (e.g., file systems). The basic architecture of RocksDB is shown in Figure 7.1. One RocksDB maintains at least one logical partition called *Column Family* (CF), which has its own in-memory write buffer (Memtable). When a Memtable is full,

it is flushed to the file system and stored as a *Sorted Sequence Table* (SST) file. SST files persistently store the KV-pairs in a sorted fashion and are organized in a sequence of levels starting from Level-0. When one level reaches its limit, one SST file is selected to be merged with the SST files in the next level that have overlapping key-ranges, which is called *compaction*. Detailed information about RocksDB is described in [200, 42]

7.2.2 Background of Three RocksDB Use Cases

We discuss three important and large-scale production use cases of RocksDB at Facebook: 1) **UDB**; 2) **ZippyDB**; and 3) **UP2X**. Sharding is used in all three use cases to achieve load balancing. Therefore, the workloads are very similar among all shards, and we randomly select three RocksDB instances from each use case to collect the traces.

UDB: Social graph data at Facebook is persistently stored in UDB, a sharded MySQL database tier [209]. The cache read misses and all writes to social graph data are processed by UDB servers. UDB relies on the MySQL instance to handle all queries, and these queries are converted to RocksDB queries via MyRocks [196, 193]. Much of the social graph data is presented as *objects* and *associations*, and is maintained in different MySQL tables following the model introduced in [209]. RocksDB uses different Column Families (CFs) to store object- and association-related data.

There are 6 major CFs in UDB: `Object`, `Assoc`, `Assoc_count`, `Object_2ry`, `Assoc_2ry`, and `Non_SG`. `Object` stores social graph object data and `Assoc` stores social graph association data, which defines connections between two objects. `Assoc_count` stores the number of associations of each object. Association counters are always updated with new values and do not have any deletions. `Object_2ry` and `Assoc_2ry` are the CFs that maintain the secondary indexes of objects and associations, respectively. They are also used for the purpose of ETL (Extract, Transform, and Load data from databases). `Non_SG` stores data from other non-social graph related services.

Because the UDB workload is an example of KV queries converted from SQL queries, some special patterns exist. We collected the traces for 14 days. Since the workload characteristics of three UDB servers are very similar, we present only one of them. The total trace file size in this server is about 1.1 TB. For some characteristics, daily data is more important. Thus, we also analyzed the workload of the last day in the 14-day period (24-hour trace) separately.

ZippyDB: A high-performance distributed KV-store called ZippyDB was developed based on RocksDB and relies on Paxos [210] to achieve data consistency and reliability. KV-pairs are divided into shards, and each shard is supported by one RocksDB instance. One of the replicas is selected as the primary shard, and the others are secondary. The primary shard processes all the writes to a certain shard. If strong consistency is required for reads, read requests (e.g., *Get* and *Scan*) are only processed by the primary shard. One ZippyDB query is converted to a set of RocksDB queries (one or more).

Compared with the UDB use case, the upper-layer queries in ZippyDB are directly mapped to the RocksDB queries, and so the workload characteristics of ZippyDB are very different. We randomly selected three primary shards of ZippyDB and collected the traces for 24 hours. Like UDB, we present only one of them. This shard stores the metadata of ObjStorage, which is an object storage system at Facebook. In this shard, a KV-pair usually contains the metadata information for an ObjStorage file or a data block with its address information.

UP2X: Facebook uses various AI/ML services to support social networks, and a huge number of dynamically changing data sets (e.g., the statistic counters of user activities) are used for AI/ML prediction and inferencing. UP2X is a distributed KV-store that was developed specifically to store this type of data as KV-pairs. As users use Facebook services, the KV-pairs in UP2X are frequently updated, such as when counters increase. If UP2X called Get before each Put to achieve a read-modify-write operation, it would have a high overhead due to the relatively slow speed of random Gets. UP2X leverages the RocksDB Merge interface to avoid Gets during the updates.

KV-pairs in UP2X are divided into shards supported by RocksDB instances. We randomly selected three RocksDB instances from UP2X and then collected and analyzed the 24-hour traces. Note that the KV-pairs inserted by Merge are cleaned during compaction via *Compaction Filter*, which uses custom logic to delete or modify KV-pairs in the background during compaction. Therefore, a large number of KV-pairs are removed from UP2X even though the delete operations (e.g., Delete, DeleteRange, and SingleDelete) are not used.

7.3 Methodology and Tool Set

To analyze and characterize RocksDB workloads from different use cases and to generate synthetic workloads, we propose and develop a set of KV-store tracing, replaying, analyzing, modeling, and benchmarking tools. These tools are already open-sourced in RocksDB release [195]. In this section, we present these tools and discuss how they are used to characterize and generate KV-store workloads.

Tracing The tracing tool collects query information at RocksDB public KV interfaces and writes to a trace file as records. It stores the following five types of information in each trace record: 1) query type, 2) CF ID, 3) key, 4) query specific data, and 5) timestamp. For *Put* and *Merge*, we store the value information in the query-specific data. For Iterator queries like *Seek* and *SeekForPrev*, the scan length (the number of *Next* or *Prev* called after *Seek* or *SeekForPrev*) is stored in the query-specific data. The timestamp is collected when RocksDB public interfaces are called with *microsecond* accuracy. In order to log the trace record of each query in a trace file, a lock is used to serialize all the queries, which will potentially incur some performance overhead. However, according to the performance monitoring statistics in production under the regular production workloads, we did not observe major throughput degradation or increased latency caused by the tracing tool.

Trace Replaying The collected trace files can be replayed through a Replayer tool implemented in *db_bench* (special plugins like MergeOperator or Comparator are required if they are used in the original RocksDB instance). The replay tool issues the queries to RocksDB based on the trace record information, and the time intervals between the queries follow the timestamps in the trace. By setting different fast forward and multithreading parameters, RocksDB can be benchmarked with workloads of different intensities. However, query order is not guaranteed with multithreading. The workloads generated by Replayer can be considered as real-world workloads.

Trace Analyzing Using collected traces for replaying has its limitations. Due to the potential performance overhead of workload tracing, it is difficult to track large-scale and long-lasting workloads. Moreover, the content of trace files is sensitive and confidential for their users/owners, so it is very hard for RocksDB users to share the traces with other RocksDB developers or developers from third-party companies (e.g., upper-layer

application developers or storage vendors) for benchmarking and performance tuning. To address these limitations, we propose a way of analyzing RocksDB workloads that profiles the workloads based on information in the traces.

The trace analyzing tool reads a trace file and provides the following characteristics: 1) a detailed statistical summary of the KV-pairs in each CF, query numbers, and query types; 2) key size and value size statistics; 3) KV-pair popularity; 4) the key-space locality, which combines the accessed keys with all existing keys from the database in a sorted order; and 5) Queries Per Second (QPS) statistics.

Modeling and Benchmarking We first calculate the Pearson correlation coefficients between any two selected variables to ensure that these variables have very low correlations. In this way, each variable can be modeled separately. Then, we fit the collected workloads to different statistical models to find out which one has the lowest fitting error, which is more accurate than always fitting different workloads to the same model (like Zipfian). The proposed benchmark can then generate KV queries based on these probability models. Details are discussed in Section 7.7.

7.4 General Statistics of Workloads

In this section, we introduce the general workload statistics of each use case including query composition in each CF, KV-pair hotness distributions, and queries per second.

7.4.1 Query Composition

By analyzing query composition, we can figure out query intensity, the ratio of query types in different use cases, and the popularity of queries. We find that: **1) Get is the most frequently used query type in UDB and ZippyDB, while Merge dominates the queries in UP2X, and 2) query composition can be very different in different CFs**.

UDB In this UDB server, over 10.2 billion queries were called during the 14-day period, and there were about 455 million queries called during the last 24 hours. There are six CFs being used in UDB as discussed in 7.2.2. Although those CFs are stored in the same RocksDB database, the workloads are very different. It is difficult to analyze and model such a mixed workload without the separation of different CFs.

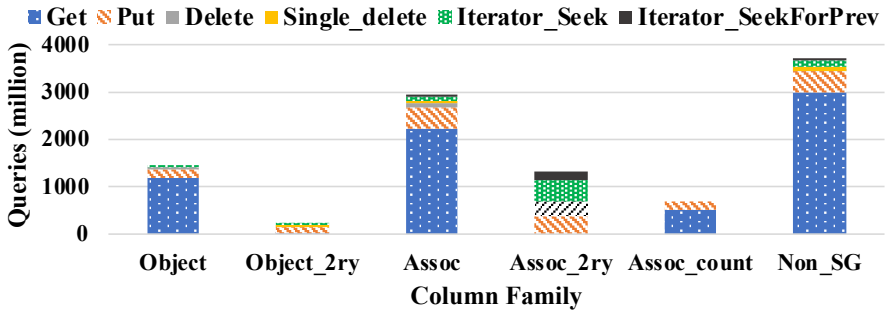
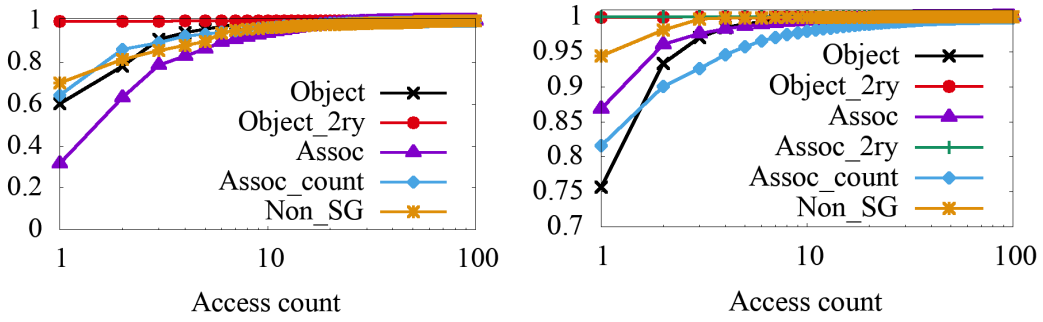


Figure 7.2: Distribution of different query types in 14 days.

The query composition in each CF is shown in Figure 7.2. Get, Put, and Iterator are three major query types in UDB, especially in `Object`, `Assoc`, and `Non_SG`. Get does not show up in the secondary indexes of objects (`Object_2ry`) and associations (`Assoc_2ry`). `Object_2ry` is built for the purpose of ETL, so Iterator is the major query type. `Assoc` mostly checks the existence of an association between two objects via Get, while the secondary index (`Assoc_2ry`) lists the objects that are associated with one target object. Since KV-pairs in `Assoc_2ry` have no repeating updates, SingleDelete is used in this CF to delete the invalid KV-pairs. In other CFs, regular Delete is called to remove the invalid KV-pairs. `Assoc_count` stores the number of associations of each object. Therefore, Get and Put are the major query types used in this CF to read and update the counters.

ZippyDB There is only one CF being used in ZippyDB. Get, Put, Delete, and Iterator_seek (forward Iterator) are the four query types that are used. Over the 24-hour period, there were about 420 million queries called in this shard. The ratios of each query type are: 78% Get, 13% Put, 6% Delete, and 3% Iterator, respectively. Get is the major query type in ZippyDB, which aligns with the read-intensive workload of ObjStorage.

UP2X Over the 24-hour period, the RocksDB instance received 111 million queries. Among them, about 92.53% of the queries are Merge, 7.46% of them are Get, and fewer than 0.01% of the queries are Put. The query composition is very different from the UDB and ZippyDB use cases, which are read dominated. About 4.71 million KV-pairs were accessed by Merge, 0.47 million by Get, and 282 by Put. Read-and-modify (Merge) is the major workload pattern in UP2X.



(a) The KV-pair access count CDF by Get

(b) The KV-pair access count CDF by Put

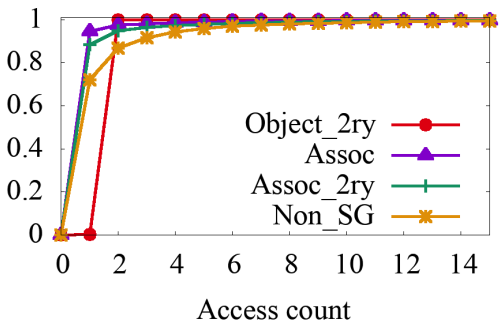
Figure 7.3: The KV-pair access count distribution queried by Get and Put in each CF during 24 hours.

7.4.2 KV-Pair Hotness Distribution

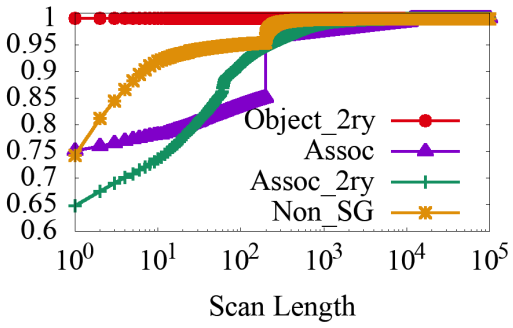
To understand the hotness of KV-pairs in each use case, we count how many times each KV-pair was accessed during the 24-hour tracing period and show them in cumulative distribution function (CDF) figures. The X-axis is the access count, and the Y-axis is the cumulative ratio between 0 and 1. We find that **in UDB and ZippyDB, most KV-pairs are cold**.

UDB We plot out the KV-pair access count CDFs for Get and Put. For Iterator, we show the start-key access count distribution and the scan length distribution. The CDFs of Get and Put are shown in Figure 7.3. Looking at Figure 7.3(a), more than 70% of the KV-pairs in `Assoc` are Get requests that occurred at least 2 times. In contrast, this ratio in other CFs is lower than 40%. It indicates that read misses of `Assoc` happen more frequently than the others. As shown in 7.3(b), in all CFs, more than 75% of the KV-pairs are Put only one time and fewer than 2% of the KV-pairs are Put more than 10 times. The majority of the KV-pairs are rarely updated.

We plot out the access count CDF of the start-keys of Iterators over the 24-hour period, as shown in Figure 7.4(a). Most of the start-keys are used only once, which shows a low access locality. Fewer than 1% of the start-keys are used multiple times by Iterators. The scan length of more than 60% of the Iterators is only 1 across all CFs, as shown in Figure 7.4(b). About 20% of the Iterators in `Assoc` scan more than 100 consecutive keys, while the ratios for `Assoc_2ry` and `Non_SG` are about 10% and 5%,



(a) The Iterator start-key access count CDF distribution



(b) The Iterator scan length CDF distribution

Figure 7.4: The Iterator scan length and start-key access count CDF of four CFs during 24 hours.

Table 7.1: The ratios of KV-pairs among all existing KV-pairs being accessed during different time periods in UDB

CF name	Object	Object_2ry	Assoc	Assoc_2ry	Assoc_count	Non_SG
24 hours	2.72%	0.62%	1.55%	1.75%	0.77%	1.38%
14 days	14.14%	6.10%	13.74%	10.37%	14.05%	11.29%

respectively. A very large scan length (higher than 10,000) is very rare, but we can still find some examples of this type in `Non_SG` and `Assoc`. The configured range query limit in MySQL creates some special scan lengths. For example, there is a jump at 200 in both `Assoc` and `Non_SG`.

We also count the number of unique keys being accessed in different time periods. As shown in Table 7.1, during the last 24 hours, fewer than 3% of the keys were accessed. During the 14-day period, the ratio is still lower than 15% for all CFs. In general, most of the keys in RocksDB are "cold" in this use case. On one hand, most read requests are responded to by the upper cache tiers [211, 41]. Only the read misses will trigger queries to RocksDB. On the other hand, social media data has a strong temporal locality. People are likely accessing the most recently posted content on Facebook.

ZippyDB The average access counts per accessed KV-pair of the four query types (Get, Put, Delete, and Iterator_seek) are: 15.2, 1.7, 1, and 10.9, respectively. Read queries (Get and Iterator_seek) show very good locality, while the majority of the KV-pairs are only Put and Deleted once in the last 24-hour period. The access count

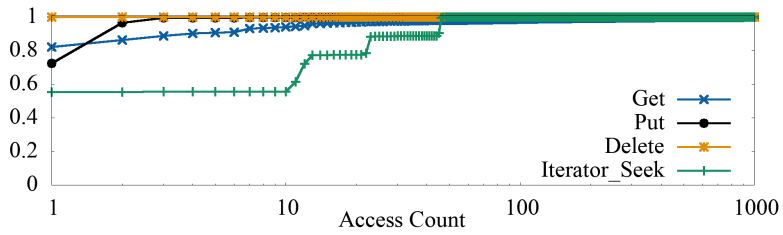


Figure 7.5: The KV-pair access count distribution of ZippyDB.

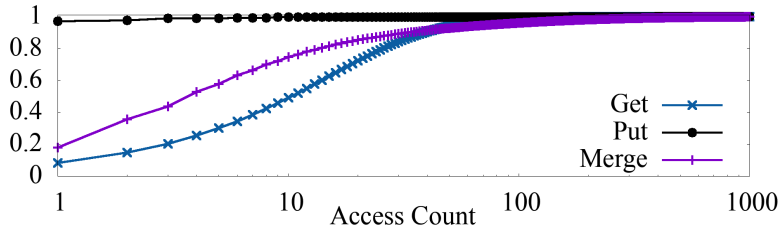
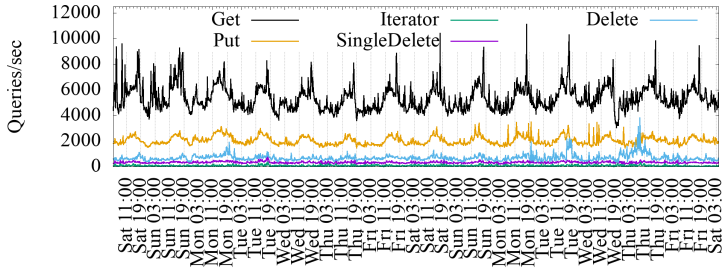


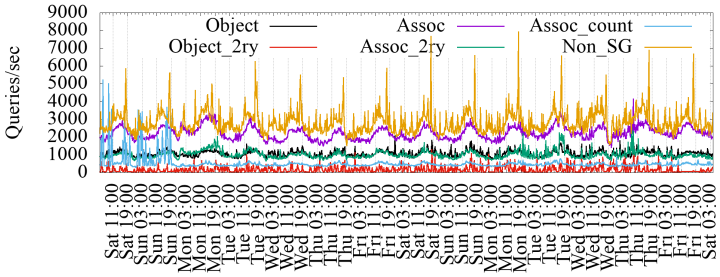
Figure 7.6: The access count distribution of UP2X.

distribution is shown in Figure 7.5. For about 80% of the KV-pairs, Get requests only occur once, and their access counts show a long tail distribution. This indicates that a very small portion of KV-pairs have very large read counts over the 24-hour period. About 1% of the KV-pairs show more than 100 Get requests, and the Gets to these KV-pairs are about 50% of the total Gets that show strong localities. In contrast, about 73% of the KV-pairs are Put only once, and fewer than 0.001% of the KV-pairs are Put more than 10 times. Put does not have as clear a locality as Get does. The CDF of Iterator_seek start-key access counts has a special distribution that we can observe very clearly through the 4 "steps" in the figure. About 55% of the KV-pairs are used as the start-key of Iterator_seek 1 time, 6% of the KV-pairs 11 times, 11% of the KV-pairs 12 times, 5% of the KV-pairs 13 times, 10% of the KV-pairs 23 times, and 10% of the KV-pairs 46 times. The special access count distribution of start-keys is caused by the metadata scanning requests in ObjStorage. For example, if one KV-pair stores the metadata of the first block of a file, it will always be used as the start-key of Iterator_seek when the whole file is requested.

UP2X The CDF distribution of KV-pair access counts is shown in Figure 7.6. Merge and Get have wide distributions of access counts. If we define a KV-pair accessed 10 times or more during the 24-hour period as a hot KV-pair, about 50% of the KV-pairs



(a) Overall QPS for each query type at different dates and times in a 14-day time span



(b) Overall QPS of each CF at different dates and times in a 14-day time span

Figure 7.7: The QPS variation at different dates and times in a 14-day time span.

accessed by Get and 25% of the KV-pairs accessed by Merge are hot. On the other hand, the ratio of very hot KV-pairs (accessed 100 times or more in the 24-hour period) for Merge is 4%, which is much higher than that of Get (fewer than 1%). Both Merge and Get have a very long tail distribution, as shown in the figure.

7.4.3 QPS (Queries Per Second)

The QPS metric shows the intensiveness of the workload variation over time. **The QPS of some CFs in UDB have strong diurnal patterns, while we can observe only slight QPS variations during day and night time in ZippyDB and UP2X. The daily QPS variations are related to social network behaviors.**

UDB The QPS of UDB is shown in Figure 7.7. Some CFs (e.g., `Assoc` and `Non_SG`) and some query types (e.g., Get and Put) have strong diurnal patterns due to the behaviors of Facebook users around the world. As shown in Figure 7.7(a), the QPS for either Get or Put usually increases from about 8:00 PST and reaches a peak at about

17:00 PST. Then, the QPS quickly drops and reaches its nadir at about 23:00 PST. The QPS of Delete, SingleDelete, and Iterator shows variations, but it is hard to observe any diurnal patterns. These queries are triggered by Facebook internal services, which have low correlation with user behaviors. The QPS of six CFs are shown in Figure 7.7(b). `Assoc` and `Non_SG` have a strong diurnal variation, but the QPS of `Non_SG` is spikier. Since ETL requests are not triggered by Facebook users, the QPS of `Object_2ry` is spiky and we cannot find any clear patterns.

ZippyDB The QPS of ZippyDB is different from that of UDB. The QPS of ZippyDB varies over the 24-hour period, but we do not find a diurnal variation pattern, especially for Put, Delete, and Iterator_Seek. Since ObjStorage is an object stored at Facebook, object read is related to social network behaviors. Therefore, the QPS of Get is relatively lower at night and higher during the day (based on Pacific Standard Time). Because range queries (Iterator_Seek) are usually not triggered by Facebook users, the QPS for this query type is stable and is between 100 and 120 most of the time.

UP2X The QPS of either Get or Put in UP2X does not have a strong diurnal variation pattern. However, the usage of Merge is closely related to the behavior of Facebook users, such as looking at posts, likes, and other actions. Therefore, the QPS of Merge is relatively lower at night (about 1000) and higher during the day (about 1500).

7.5 Key and Value Sizes

Key size and value size are important factors in understanding the workloads of KV-stores. They are closely related to performance and storage space efficiency. The average (AVG) and standard deviation (SD) of key and value sizes are shown in Table 7.2, and the CDFs of key and value sizes are shown in Figure 7.8. In general, **key sizes are usually small and have a narrow distribution, and value sizes are closely related to the types of data. The standard deviation of key sizes is relatively small, while the standard deviation of value size is large. The average value size of UDB is larger than the other two**.

UDB The average key size is between 16 and 30 bytes except for `Assoc_2ry`, which has an average key size of 64 bytes. The keys in `Assoc_2ry` consist of the 4-byte MySQL

Table 7.2: The average key size (AVG-K), the standard deviation of key size (SD-K), the average value size (AVG-V), and the standard deviation of value size (SD-V) of UDB, ZippyDB, and UP2X (in bytes)

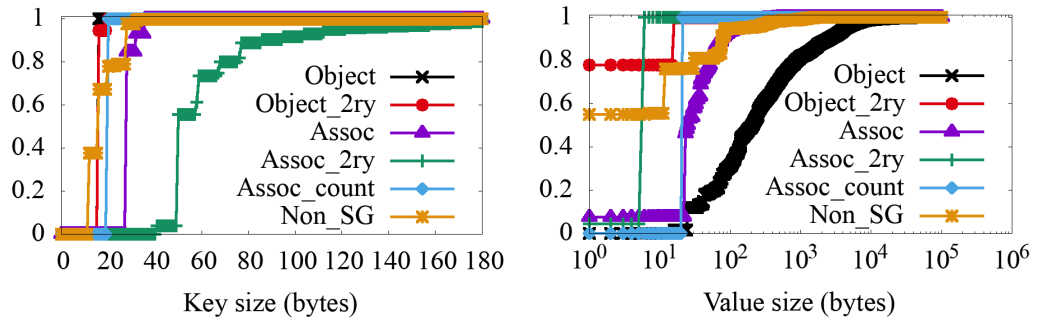
	AVG-K	SD-K	AVG-V	SD-V
UDB	27.1	2.6	126.7	22.1
ZippyDB	47.9	3.7	42.9	26.1
UP2X	10.45	1.4	46.8	11.6

table index, two object IDs, the object type, and other information. Therefore, the key size of `Assoc_2ry` is usually larger than 50 bytes and has a long tail distribution as shown in Figure 7.8(a). For other CFs, the keys are composed of the 4-byte MySQL table index as the prefix, and 10 to 30 bytes of primary or secondary keys like object IDs. Thus, the keys show a narrow distribution. Note that the key sizes of a very small number of KV-pairs are larger than 1 KB, which is not shown in the key size CDF due to the X-axis scale limit.

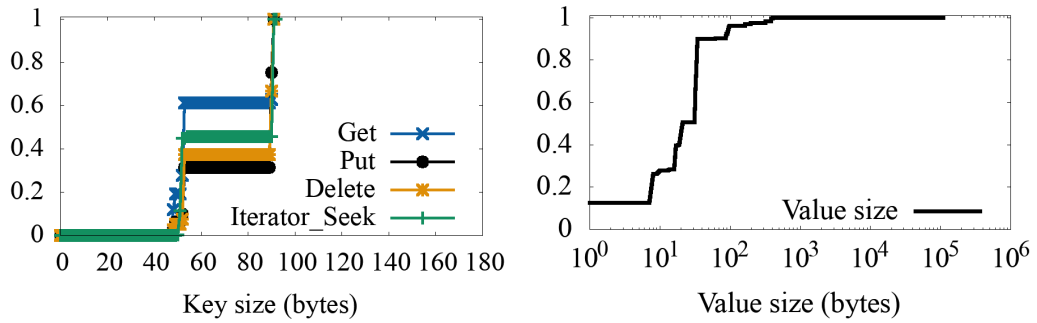
The value size distribution is shown in Figure 7.8(b). `Object` and `Assoc` have a long tail distribution. The value sizes of `Object` vary from 16 bytes to 10 KB, and more than 20% of the value sizes are larger than 1 KB. The average value size of KV-pairs in `Object` is about 1 KB and the median is about 235B, which is much larger than those in other CFs. User data, like the metadata of photos, comments, and other posted data, is stored in this CF, which leads to a large value size. In `Assoc`, the value sizes are relatively small (the average is about 50 bytes) and vary from 10 bytes to 200 bytes.

A very special case is `Assoc_count`, whose key size and value size are exactly 20 bytes. According to the design of this CF, the key is 20 bytes (*bigint* association ID) and is composed of a 10-byte counter and 10 bytes of metadata. Since all the information used in secondary index CFs (`Assoc_2ry` and `Object_2ry`) is stored in its key, the value does not contain any meaningful data. Therefore, the average value size is less than 8 bytes and there are only three possible value sizes in the distribution (1 byte, 6 bytes, or 16 bytes) as shown in Figure 7.8(b). For CFs with large value sizes like `Object`, optimizations like separating key and value [212] can effectively improve performance.

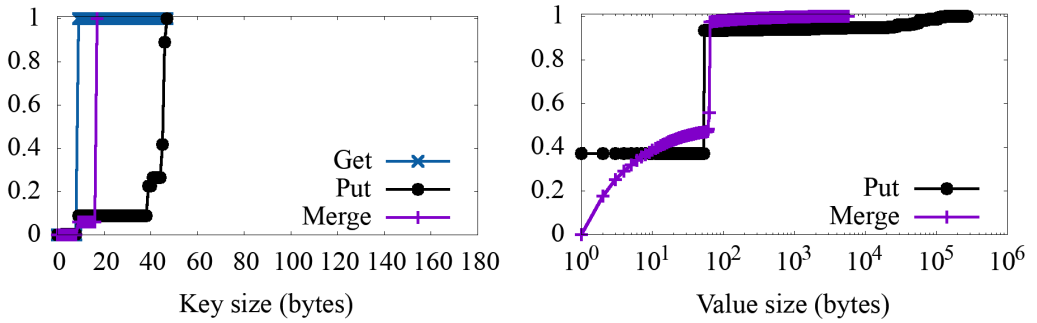
ZippyDB Since a key in ZippyDB is composed of ObjStorage metadata, the key sizes are relatively large. The CDF of the key sizes is shown in Figure 7.8(c). We can find several "steps" in the CDF. **Nearly all of the key sizes are in the two size**



(a) UDB key size CDF (b) UDB value size CDF



(c) ZippyDB key size CDF (d) ZippyDB value size CDF



(e) UP2X key size CDF (f) UP2X value size CDF

Figure 7.8: The key and value size distributions of UDB, ZippyDB, and UP2X.

ranges: [48, 53] and [90, 91]. The ratio of KV-pairs in these two key size ranges are different for different query types. For example, about 60% of the key sizes of Get are in the [48, 53] range, while the ratio for Put is about 31%.

The value sizes are collected from Put queries. As shown in Figure 7.8(d), the value size distribution has a very long tail: about 1% of the value sizes are larger than 400

bytes, and about 0.05% of the value sizes are over 1 KB. Some of the value sizes are even larger than 100 KB. However, most of the KV-pairs have a small value. More than 90% of the value sizes are smaller than 34 bytes, which is even smaller than the key sizes.

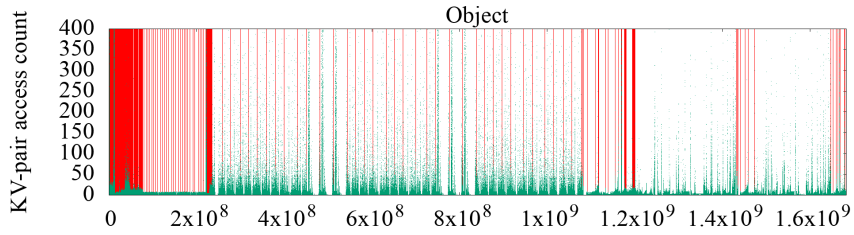
UP2X The key sizes do not have a wide distribution, as shown in Figure 7.8(e). **More than 99.99% of the KV-pairs accessed by Get have a key size of 9 bytes**. About 6% of the KV-pairs inserted by Merge have a key size of 9 bytes, and 94% are 17 bytes. The 17-byte KV-pairs are all cleaned during compaction, and they are never read by upper-layer applications through Get. Put is rarely used in UP2X. Among the 282 KV-pairs inserted by Put, about 8.9% of the key sizes are smaller than 10 bytes, and 47% of them are 46 bytes.

The value size distribution is shown in Figure 7.8(f). The value sizes of some KV-pairs inserted by Put are extremely large. The average is about 3.6 KB, and about 3% of the KV-pairs are over 100 KB. The value sizes of KV-pairs inserted by Merge have a special distribution. About 40% of the values are smaller than 10 bytes, and about 52% of the values are exactly 64 bytes. A large portion of the updates in UP2X are the counters and other structured information. Thus, the value sizes of those KV-pairs are fixed to 64 bytes.

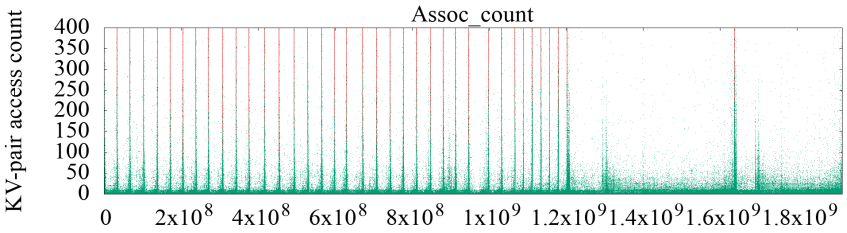
7.6 Key-Space and Temporal Patterns

KV-pairs in RocksDB are sorted and stored in SST files. In order to understand and visualize the key-space localities, we sort all the existing keys in the same order as they are stored in RocksDB and plot out the access count of each KV-pair, which is called the *heat-map* of the whole key-space. Each existing key is assigned a unique integer as its *key-ID*, based on its sorting order and starting from 0. We refer to these *key-IDs* as the *key sequence*.

The KV-pair accesses show some special temporal patterns. For example, some KV-pairs are intensively accessed during a short period of time. In order to understand the correlation between temporal patterns and key-space locality, we use a time series sequence to visualize these patterns. We sort the keys in ascending order and assign them with key-IDs as previously discussed, and this *key sequence* is used as the X-axis.



(a) The heat-map of Get in `Object` during a 24-hour period



(b) The heat-map of Get in `Assoc_count` during a 24-hour period

Figure 7.9: The heat-map of Get in `Object` and `Assoc_count` during a 24-hour period. The X-axis represents the key-ID of keys in the whole key-space, and the Y-axis represents the KV-pair access counts. The red vertical lines are the MySQL table boundaries.

The Y-axis shows the time when a query is called. To simplify the Y-axis value, we shift the timestamp of each query to be relative to the tracing start time. Each dot in the time series figure represents a request to a certain key at that time. In the UDB use case, the first 4 bytes of a key are the MySQL table index number due to the key composition of MyRocks. We separate the key-space into different key-ranges that belong to different tables by red vertical lines.

The heat-maps of the three use cases show a strong key-space locality. Hot KV-pairs are closely located in the key-space. The time series figures of Delete and SingleDelete for UDB and Merge for UP2X show strong temporal locality. For some query types, KV-pairs in some key-ranges are intensively accessed during a short period of time.

UDB We use the heat-map of Get in `Object` and `Assoc_count` over a 24-hour period as an example to show the key-space localities as shown in Figure 7.9. Hot KV-pairs (with high access counts) are usually located in a small key-range and are close to each

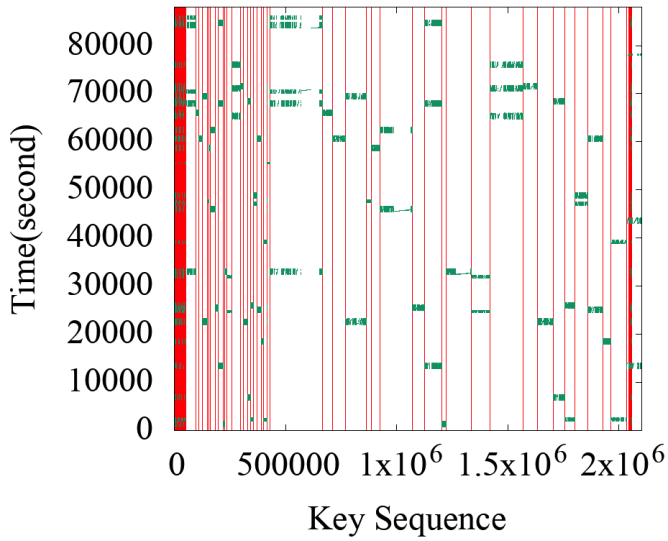
other, which shows in both Figure 7.9(a) and Figure 7.9(b). That is, they show a strong key-space locality (indicated by the dense green areas). **Some MySQL tables (the key-ranges between the red vertical lines) are extremely hot (e.g., the green dense area in `Object`), while other tables have no KV-pair accesses**. One interesting characteristic is that the KV-pairs with high access counts in `Assoc_count` are skewed toward the end of the table as shown in Figure 7.9(b). In social graphs, new objects are assigned with relatively larger IDs, and new associations are frequently added to the new objects. Therefore, new KV-pairs in `Assoc_count` are hot and are usually at the end of the MySQL table. Moreover, the heat-maps of Get and Put are similar. Usually, the keys with the most Get queries are the ones with the most Put queries.

Most KV-pairs are deleted only once, and they are unlikely to be reinserted. Therefore, there are no hot KV-pairs in Delete and SingleDelete queries. However, they show some special patterns. For example, some nearby KV-pairs are deleted together in a short period of time as shown in Figure 7.10.

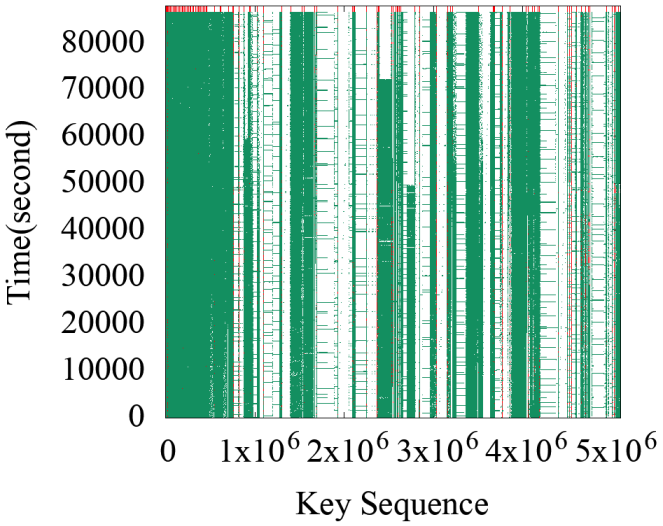
In Figure 7.10(a), the deleted KV-pairs in the same table for `Object` are removed together in a short period of time (indicated by green dots with close Y values). After that, deletions will not happen for a long period of time. Similar patterns also appear in the SingleDelete time series for `Assoc_2ry`, as shown in Figure 7.10(b). In some MySQL tables, SingleDelete is intensively called in several short time intervals to remove KV-pairs in the same table. Between any two sets of intensive deletions, SingleDelete is never called, which causes the "green blocks" in the time series figures.

In general, KV-pairs are not randomly accessed in the whole key-space. The majority of KV-pairs are not accessed or have low access counts. Only a small portion of KV-pairs are extremely hot. These patterns appear in the whole key-space and also occur in different key-ranges. KV-pairs belonging to the same MySQL table are physically stored together. Some SST files at different levels or data blocks in the same SST file are extremely hot. Thus, the compaction and cache mechanisms can be optimized accordingly.

ZippyDB The heat-map of Get in ZippyDB shows a very good key-space locality. For example, as shown in Figure 7.11, **the KV-pairs accessed by Get have high access counts and are concentrated in several key-ranges (e.g., between** 1×10^6



(a) The time series of Delete queries in `Object`



(b) The time series of SingleDelete queries in `Assoc_2ry`

Figure 7.10: The time series figure of a 24-hour trace.

and 5×10^6**)**. Hot KV-pairs are not randomly distributed: instead, these KV-pairs are concentrated in several small key-ranges. The hotness of these key-ranges is closely related to cache efficiency and generated storage I/Os. The better a key-space locality is, the higher the RocksDB block cache hit ratio will be. Data blocks that are associated

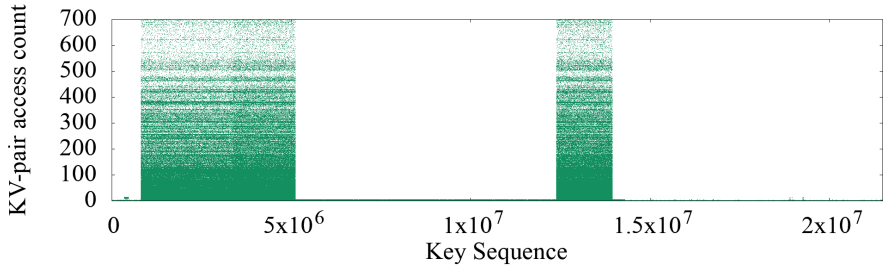


Figure 7.11: Heat-map of KV-pairs accessed by Get in ZippyDB.

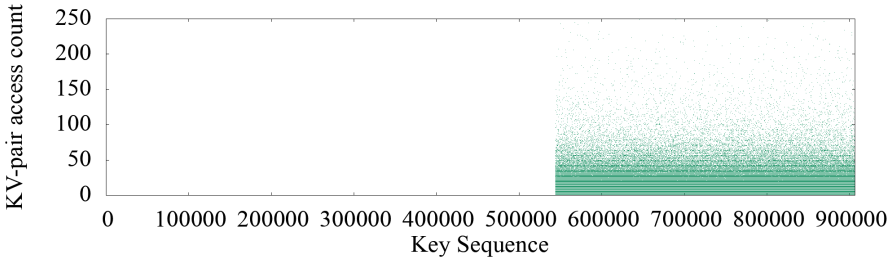


Figure 7.12: Heat-map of KV-pairs accessed by Get in UP2X.

with hot key-ranges will most likely be cached in the RocksDB block cache. These data blocks are actually cold from a storage point of view. With a good key-space locality, the number of data block reads from SST files will be much lower than a random distribution. A similar locality is also found in the Put and Iterator_seek heat-maps. Since all the KV-pairs are deleted once, we did not observe any key-space locality for Delete. In general, the ZippyDB workload is read-intensive and has very good key-space locality.

UP2X If we look at the heat-map of all KV-pairs accessed by Get as shown in Figure 7.12, **we can find a clear boundary between hot and cold KV-pairs**. Note that the whole key-space was collected after the tracing was completed. In the heat-map, the KV-pairs from 0 to about 550,000 are never accessed by Gets, but the KV-pairs from 550,000 to 900,000 are frequently accessed. A similar locality is also shown in the heat-map of Merge. While KV-pairs from 0 to about 550,000 are sometimes accessed by Merge, their average access counts are much lower than those of the KV-pairs from 550,000 to 900,000. This special locality might be caused by a unique behavior of AI/ML services and their data update patterns.

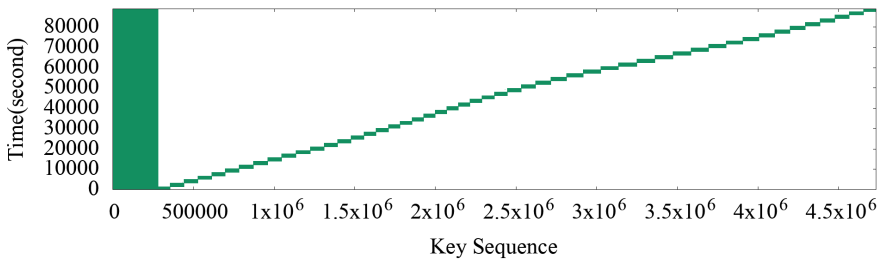


Figure 7.13: The time series of Merge in UP2X.

The UP2X use case shows a very strong key-space locality and temporal locality in Merge. However, about 90% of the KV-pairs inserted by Merge are actually cleaned during compaction. Since the key-space heat-map does not show the existence of KV-pairs cleaned by compactions, we plot out the time series sequence for Merge, which can indicate Merge accesses of all KV-pairs. As shown in Figure 7.13, KV-pairs between 0 and 250,000 are frequently accessed during the 24-hour period. These are KV-pairs between 0 and 900,000 in the whole key-space. The KV-pairs between 250,000 and 4,700,000 show very special key-space and temporal localities. The green blocks indicate that **a small range of KV-pairs are intensively called by Merge during half an hour**. After that, a new set of KV-pairs (with incrementally composed keys) are intensively accessed by Merge during the next half an hour. These KV-pairs are cleaned during compactions. Get and Put do not have similar temporal and key-space localities.

7.7 Modeling and Benchmarking

After understanding the characteristics of some real-world workloads, we further investigate whether we can use existing benchmarks to model and generate KV workloads that are close to these realistic workloads. We do not consider deletions in our current models.

7.7.1 How Good Are the Existing Benchmarks?

Several studies [213, 21, 214] use YCSB/db_bench + LevelDB/RocksDB to benchmark the storage performance of KV-stores. Researchers usually consider the workloads generated by YCSB to be close to real-world workloads. YCSB can generate queries that

have similar statistics for a given query type ratio, KV-pair hotness distribution, and value size distribution as those in realistic workloads. However, it is unclear whether their generated workloads can match the I/Os for underlying storage systems in realistic workloads.

To investigate this, we focus on storage I/O statistics such as *block reads*, *block cache hits*, *read-bytes*, and *write-bytes* collected by *perf_stat* and *io_stat* in RocksDB. To exclude other factors that may influence the storage I/Os, we replay the trace and collect the statistics in a clean server. The benchmarks are also evaluated in the same server to ensure the same setup. To ensure that the RocksDB storage I/Os generated during the replay are the same as those in production, we replay the trace on a snapshot of the same RocksDB in which we collected the traces. The snapshot was made at the time when we started tracing. YCSB is a benchmark for NoSQL applications and ZippyDB is a typical distributed KV-store. Therefore, the workloads generated by YCSB are expected to be close to the workloads of ZippyDB, and we use ZippyDB as an example to investigate. Due to special plugin requirements and the workload complexities of UDB and UP2X, we did not analyze storage statistics for those two use cases.

Before we run YCSB, we set the YCSB parameters of `workloada` and `workloadb` to fit ZippyDB workloads as much as possible. That is, we use the same cache size, ensure that the request distribution and scan length follows Zipfian, set the fieldlength as the average value size, and use the same Get/Put/Scan ratios as those shown in Section 7.4. Since we cannot configure the compression ratio in YCSB to make it the same as ZippyDB, we use the default configuration in YCSB. We normalize the results of the RocksDB storage statistics based on those from the trace replay.

The number of block reads from YCSB is at least 7.7x that of the replay results, and the amount of read-bytes is about 6.2x. The results show an extremely high read amplification. Although the collected amount of write-bytes from YCSB is about 0.74x that of the replay, the actual amount of write-bytes is much lower if we assume YCSB achieves the same compression ratio as ZippyDB (i.e., if the YCSB compression ratio is 4.5, the amount of write-bytes is about 0.41x that of the replay). Moreover, the number of block cache hits is only about 0.17x that of the replay results. This evaluation shows that, even though the overall query statistics (e.g., query number, average value size, and KV-pair access distribution) generated by YCSB are close to those of ZippyDB

workloads, the RocksDB storage I/O statistics are actually quite different. db_bench has a similar situation.

Therefore, using the benchmarking results of YCSB as guidance for production might cause some misleading results. For example, the read performance of RocksDB under a production workload will be higher than what we tested using YCSB. The workload of YCSB can easily saturate the storage bandwidth limit due to its extremely high read amplification. Also, the write amplification estimated from the YCSB benchmarking results are lower than in real production. The write performance can be overestimated and might also lead to incorrect SSD lifetime estimates.

With detailed analyses, we find that the main factor that causes this serious read amplification and fewer storage writes is the ignorance of key-space locality. RocksDB reads data blocks (e.g., 16 KB) instead of a KV-pair from storage to memory when it encounters a cache miss. In YCSB, even though the overall KV-pair hotness follows the real-world workload distribution, the hot KV-pairs are actually randomly distributed in the whole key-space. The queries to these hot KV-pairs make a large number of data blocks hot. Due to the cache space limit, a large number of hot data blocks that consist of the requested KV-pairs will not be cached, which triggers an extremely large number of block reads. In contrast, in ZippyDB, hot KV-pairs only appear in some key-ranges, so the number of hot data blocks is much smaller. Similarly, a random distribution of hot KV-pairs causes more updated KV-pairs to be garbage collected in the newer levels during compactions. Therefore, old versions of cold KV-pairs that are being updated are removed earlier in the newer levels, which leads to fewer writes when compacting older levels. In contrast, if only some key-ranges are frequently updated, old versions of cold KV-pairs are continuously compacted to the older levels until they are merged with their updates during compactions. This causes more data to be written during compactions.

7.7.2 Key-Range Based Modeling

Unlike workloads generated by YCSB, real-world workloads show strong key-space localities according to the workload characteristics presented in Sections 7.6. Hot KV-pairs are usually concentrated in several key-ranges. Therefore, to better emulate a real-world workload, we propose a key-range based model. The whole key-space is partitioned into

several smaller key-ranges. Instead of only modeling the KV-pair accesses based on the whole key-space statistics, we focus on the hotness of those key-ranges.

How to determine the key-range size (the number of KV-pairs in the key-range) is a major challenge of key-range based modeling. If the key-range is extremely large, the hot KV-pairs are still scattered across a very big range. The accesses to these KV-pairs may still trigger a large number of data block reads. If the key-range is very small (e.g., a small number of KV-pairs per range), hot KV-pairs are actually located in different key-ranges, which regresses to the same limitations as a model that does not consider key-ranges. Based on our investigation, when the key-range size is close to the average number of KV-pairs in an SST file, it can preserve the locality in both the data block level and SST level. Therefore, we use average number of KV-pairs per SST file as key-range size.

We first fit the distributions of key sizes, value sizes, and QPS to different mathematical models (e.g., Power, Exponential, Polynomial, Webull, Pareto, and Sine) and select the model that has the minimal fit standard error (FSE). This is also called the root mean squared error. For example, for a collected workload of ZippyDB, the key size is fixed at either 48 or 90 bytes, the value sizes follow a Generalized Pareto Distribution [215], and QPS can be better fit to Cosine or Sine in a 24-hour period with very small amplitude.

Then, based on the KV-pair access counts and their sequence in the whole key-space, the average accesses per KV-pair of each key-range is calculated and fit to the distribution model (e.g., power distribution). This way, when one query is generated, we can calculate the probability of each key-range responding to this query. Inside each key range, we let the KV-pair access count distribution follow the distribution of the whole key-space. This ensures that the distribution of the overall KV-pair access counts satisfies that of a real-world workload. Also, we make sure that hot KV-pairs are allocated closely together. Hot and cold key-ranges can be randomly assigned to the whole key-space, since the locations of key-ranges have low influence on the workload locality.

Based on these models, we further develop a new benchmark using db_bench. When running the benchmark, the QPS model controls the time intervals between two consecutive queries. When a query is issued, the query type is determined by the probability

of each query type calculated from the collected workload. Then, the key size and value size are determined by the probability function from the fitted models. Next, based on the access probability of each key-range, we choose one key-range to respond to this query. Finally, according to the distribution of KV-pair access counts, one KV-pair in this key range is selected, and its key is used to compose the query. In this way, the KV queries are generated by the benchmark and follow the expected statistical models. At the same time, it better preserves key-space locality.

7.7.3 Comparison of Benchmarking Results

We fit the ZippyDB workload to the proposed model (Delete is excluded) and build a new benchmark called Prefix_dist [195]. To evaluate the effectiveness of key-range-based modeling, we also implement three other benchmarks with different KV-pair allocations: 1) Prefix_random models the key-range hotness, but randomly distributes the hot and cold KV-pairs in each key-range; 2) similar to YCSB, All_random follows the distribution of KV-pair access counts, but randomly distributes the KV-pairs across the whole key-space; and 3) All_dist puts the hot keys together in the whole key-space instead of using a random distribution. All four benchmarks achieve a similar compression ratio as that of ZippyDB.

Similar to the process described in Section 7.7.1, we configure YCSB *workloada* and *workloadb* to fit the ZippyDB workload as closely as possible. We run YCSB with the following 4 different request distributions: 1) uniform (YCSB_uniform), 2) Zipfian (YCSB_zipfian), 3) hotspot (YCSB_hotspot), and 4) exponential (YCSB_exp). We use the same pre-loaded database (with 50 million randomly inserted KV-pairs that have the same average key and value sizes as those of a real-world workload) for the 8 benchmarks. The RocksDB cache size is configured with the same value as the production setup. We run each test 3 times (the following discussion uses average value) and normalize the results based on that of replay.

Figure 7.14 compares the I/O statistics of the 8 benchmarks. The total number of block reads and the amount of read-bytes by YCSB_zipfian workloads are at least 500% higher than those of the original replay results. Even worse, the number of block reads and the amount of read-bytes of the other three YCSB benchmarking results are even higher, at 1000% or more compared with the replay results. In contrast, the amount

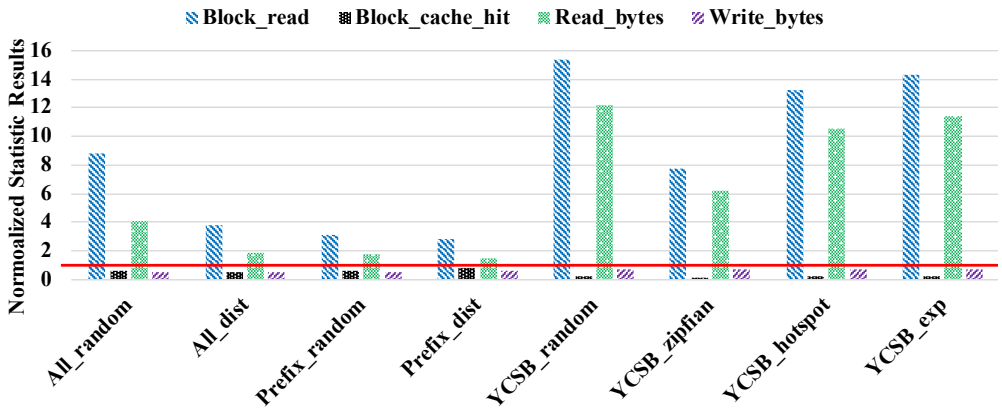
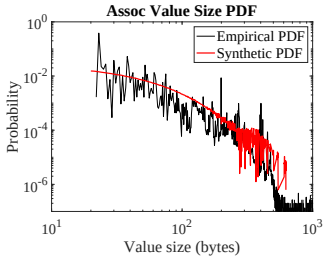


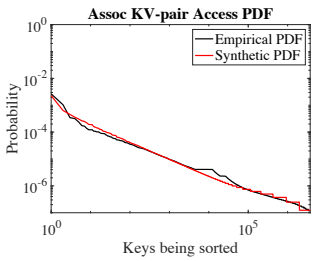
Figure 7.14: The normalized block read, block cache hit, read-bytes, and write-bytes of benchmarks based on that of the replay. We collected statistics from ZippyDB trace replay results and normalized the statistics from 8 benchmarks. The red line indicates the normalized replay results at 1. The closer the results are to the red line, the better.

of read-bytes of Prefix_dist are only 40% higher, and are the closest to the original replay results. If we compare the 4 benchmarks we implemented, we can conclude that Prefix_dist can better emulate the number of storage reads by considering key-space localities. All_dist and Prefix_random reduce the number of extra reads by gathering the hot KV-pairs in different granularities (whole key-space level vs. key-range level). Note that if YCSB achieves a similar compression ratio, the RocksDB storage I/Os can be about 35-40% lower. However, this is still much worse than the storage I/Os of All_dist, Prefix_random, and Prefix_dist.

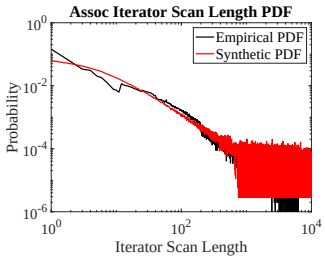
If the same compression ratio is applied, the actual amount of write-bytes by YCSB should be less than 50% of the original replay. Prefix_dist achieves about 60% write-bytes of the original replay. Actually, the mismatch between key/value sizes and KV-pair hotness causes fewer write-bytes compared with the original replay results. In general, YCSB can be further improved by: 1) adding a key-range based distribution model as an option to generate the keys, 2) providing throughput control to simulate the QPS variation, 3) providing key and value size distribution models, and 4) adding the ability to simulate different compression ratios.



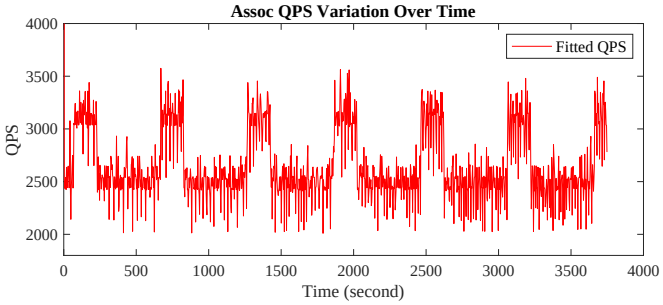
(a) The value size PDF comparison between the workload in `Assoc` and the synthetic workload



(b) The access count PDF comparison between the workload in `Assoc` and the synthetic workload



(c) The iterator scan lengths access count PDF comparison between the workload in `Assoc` and the synthetic workload



(d) The QPS comparison between the workload in `Assoc` and the synthetic workload

Figure 7.15: The synthetic workload QPS, and the PDF comparisons between the collected workload and the synthetic workload.

7.7.4 Verification of Benchmarking Statistics

We select the `Assoc` workload from UDB as another example to verify whether our benchmark can achieve KV query statistics that are very similar to those of real-world workloads. Since 90% of keys are 28 bytes and 10% of keys are 32 bytes in `Assoc`, we can use these two fixed key sizes. We find that Generalized Pareto Distribution [215] best fits the value sizes and Iterator scan length. The average KV-pair access count of key-ranges can be better fit in a two-term power model [216, 217], and the distribution of KV-pair access counts follows a power-law that can be fit to the simple power model [216, 217]. As we discussed in Section 7.4.3, because the QPS variation has a strong diurnal pattern, it can be better fit to the Sine model [218] with a 24-hour period.

To compare the workload statistics obtained from benchmarking with those of realistic workloads, we run the new benchmark with a different workload scale: 1) 10 million queries, 2) 30 million existing keys, 3) a 600-second period of QPS *sine*, and 4) a {Get, Put, Iterator} ratio of {0.806, 0.159, 0.035}, respectively (the same as in UDB `Assoc`). We collect the trace during benchmarking and analyze the trace. Figure 7.15 shows the QPS variation (shown in Figure 7.15(d)) and the probability density function (PDF) comparison of value sizes (shown in Figure 7.15(a)), KV-pair access counts (shown in Figure 7.15(b)), and Iterator scan lengths (shown in Figure 7.15(c)) between the UDB `Assoc` workload and the generated workload. Although the scale of the workload generated from our benchmark is different from that of UDB `Assoc`, the PDF figures show that they have nearly the same distribution. This verifies that the generated synthetic workload is very close to the UDB `Assoc` workload in terms of those statistics.

7.8 Related Work

During the past 20 years, the workloads of storage systems, file systems, and caching systems have been collected and analyzed in many studies. Kavalanekar *et al.* collected block traces from production Windows servers at Microsoft and provided workload characterizations that have benefitted the design of storage systems and file systems tremendously [219]. Riska *et al.* analyzed the disk-level workload generated by different applications [220]. The file system workloads were studied by industrial and academic researchers at different scales [122, 123, 114]. The workloads of the web server caches were also traced and analyzed [221, 222, 223, 224]. While the web cache can be treated as a KV-store, the query types and workloads are different from persistent KV-stores.

Although KV-stores have become popular in recent years, the studies of real-world workload characterization of KV-stores are limited. Atikoglu *et al.* analyzed the KV workloads of the large-scale Memcached KV-store at Facebook [41]. They found that reads dominate the requests, and the cache hit rate is closely related to the cache pool size. Some of their findings, such as the diurnal patterns, are consistent with what we present in Section 7.4. Major workload characteristics of RocksDB are very different from what Atikoglu *et al.* found in Memcached. Other KV-store studies, such as SILT [225], Dynamo [45], FlashStore [226], and SkimpyStash [227], evaluate designs and

implementations with some real-world workloads. However, only some simple statistics of the workloads are mentioned. The detailed workload characteristics, modeling, and synthetic workload generation are missing.

Modeling the workloads and designing benchmarks are also important for KV-store designs and their performance improvements. Several benchmarks designed for big data NoSQL systems, such as YCSB [197], LinkBench [209], and BigDataBench [228], are also widely used to evaluate KV-store performance. Compared with these benchmarks, we further provide the tracing, analyzing, and key-range based benchmarking tools for RocksDB. The users and developers of RocksDB can easily develop their own specific benchmarks based on the workloads they collect with better emulation in both the KV-query level and storage level.

7.9 Conclusion and Future Work

In this research, we present the study of persistent KV-store workloads at Facebook. We first introduce the tracing, replaying, analyzing, and benchmarking methodologies and tools that can be easily used. The findings of key/value size distribution, access patterns, key-range localities, and workload variations provide insights that can help optimize KV-store performance. By comparing the storage I/Os of RocksDB benchmarked by YCSB and those of trace replay, we find that many more reads and fewer writes are generated by benchmarking with YCSB. To address this issue, we propose a key-range based model to better preserve key-space localities. The new benchmark not only provides a good emulation of workloads at the query level, but also achieves more precise RocksDB storage I/Os than that of YCSB.

We have already open-sourced the tracing, replaying, analyzing, and the new benchmark in the latest RocksDB release (see the Wiki for more details [195]). The new benchmark is part of the benchmarking tool of *db_bench* [229]. We are not releasing the trace at this time. In the future, we will further improve YCSB workload generation with key-range distribution. Also, we will collect, analyze, and model the workloads in other dimensions, such as correlations between queries, the correlation between KV-pair hotness and KV-pair sizes, and the inclusion of additional statistics like query latency and cache status.

Chapter 8

Conclusion

In this chapter, we first conclude the contributions of this thesis. Then, we provide the visions of our future research on storage systems for big data applications.

8.1 Contributions

As more and more data is generated during everyday life all over the world, the big data infrastructure is very important to support the virtual world constructed by different types of data. The storage systems in big data infrastructure are the essential components to provide data persistency, reliability, scalability, flexibility, and high performance. This thesis studies several challenging issues in designing and developing storage systems for big data applications. Based on the data source and storing purposes, the typical storage systems in big data infrastructure are primary storage systems, warm/cold storage systems, and storage systems for big data processing. This thesis focuses on some of the research issues in the three storage system categories.

First, in primary storage systems, we propose and develop TDDFS, a tier-aware file system with data deduplication. This research addresses the tradeoffs between storage system cost and performance. To support the big data services, a huge storage space is needed in the primary storage system, which causes storage space wasting and high costs. TDDFS stores most of the hot files in the fast tier to ensure high performance. At the same time, most of the cold files are migrated to the slow tier. During the migration, files are deduplicated to further improve the slow tier space efficiency. Moreover, files

being migrated back to the fast tier maintain deduplication structure to save the storage space. This research investigates a way of making better tradeoffs between performance and space efficiency. We demonstrated that the overall performance of TDDFS is close to the pure fast tier based storage but its cost is close to the pure slow tier based storage.

Second, in cold storage systems, data deduplication is an important technique to achieve high space efficiency especially for archive and backup data. However, data deduplication causes high data fragmentation issues which lead to an extremely low read and restore performance. Two research studies improve the restore performance from different perspectives. We propose an adaptive hybrid cache to reduce the storage reads during restore. This approach uses two different cache designs for data chunks with different localities. The cache boundary of the two cache designs is dynamically changed based on the workload variations. In our evaluation, our approach can achieve a better restore throughput than using a single cache or using the hybrid cache with a fixed space boundary between the two cache schemes.

In some situations, if the data chunks are highly fragmented, caching schemes cannot further reduce the storage reads. We further propose the look-back window assisted data chunk rewrite. We found that some of the data chunk fragmentations are caused by referencing the duplicated chunks stored far away from each other. We used the look-back window to measure the data fragmentation level and predict the potential storage reads during restore. Then, some of the data chunks are rewritten based on the restore prediction. According to our evaluation, our approach can achieve fewer storage reads than the existing rewrite designs and rewrite fewer data chunks. By combining the caching and rewrite designs, we can achieve better tradeoffs between restore performance and storage space efficiency.

Third, this thesis also studied the performance issue of a distributed KV-store for big data processing platforms. Due to the compute and storage clusters disaggregation in big data infrastructure, data is always transmitted through the network, which causes the extremely heavy traffic. For some I/O intensive applications like HBase, its performance can be heavily influenced when the network is congested. In this thesis, we first investigate the reasons that cause the performance overhead. Then, we propose the in-storage computing based HBase to solve the performance issue by reducing the unnecessary network traffic. Some of the data intensive modules of HBase are offloaded to

the storage server and only the final results are sent through the network to the HBase main service running on the compute cluster. Our study demonstrates the potential of solving the performance issue in big data infrastructure with in-storage computing concepts, which might be used by other applications. Moreover, we addressed the caching issue by using a hybrid adaptive cache in the compute clusters to improve the cache hit ratio. Our approach uses much smaller network bandwidth which achieves higher performance than the legacy HBase and the new HBase design is robust to the network bandwidth variation.

Finally, the workloads in big data infrastructure are highly diversified. Understanding the characteristics of the workloads in some typical storage systems in big data infrastructure is very useful to further improve the system performance and designs. In this thesis, the key-value workload characteristics of RocksDB in primary storage, warm storage, and storage system for big data processing are explored. The findings of key/value sizes distribution, access patterns, access localities, and workload variations provide insights that can benefit KV-store performance optimization. Moreover, we propose the key-range based workload modeling which can better emulate the storage I/Os when RocksDB is evaluated with the new benchmarks developed based on the key-range based workload models.

In general, this thesis addresses some of the storage system issues and challenges in big data infrastructure. Specifically, this thesis mainly focuses on improving the performance of primary storage systems, warm/cold storage systems, and the storage systems for big data processing platforms. At the same time, we make better tradeoffs between performance and cost-effectiveness.

8.2 Future Work

The storage systems for big data applications are usually large-scale, distributed, and complicated. Different applications, different use cases, different data types cause different issues and challenges in storage systems. It creates a blue sky for us to explore the new issues and come up with novel solutions. In the future, we will focus on the following research areas of storage systems for big data applications.

First, data itself is a huge mine. Exploring the correlations and connections of data

is very useful. Some services and applications are designed specifically for exploring the value of data, for example, graph processing, social graph ML, and AI-based data graph exploring. As the data scale becomes larger and larger, a fast storage is needed as an extension of memory to support data processing. However, due to the huge performance gap between the memory and the storage, how to ensure a very fast data processing is a huge challenge to extend in-memory processing to storage. Our research will focus on proposing and designing a high performance storage system for those kinds of data processing services to reduce the memory cost.

Second, new types of storage devices based on the new storage media with much bigger capacity and a very long lifetime are experimented and developed. For example, glass and DNA might be the next generation of storage media with extremely large capacity. However, due to the special data read and write attributes of glass and DNA, the existing storage system designs cannot be directly used for them. Our future research will focus on designing new storage systems for these new storage media. Specifically, DNA-based storage tiering and high capacity glass-based storage are our future research directions.

Finally, our research will focus on the hyper-converged infrastructure for big data, especially the data storage issues will be explored. In big data, people do not care about where the data is stored and applications do not care how to read and write the data. Instead, users and applications care about how the data is created and used. Therefore, data is no longer constrained to be stored in specific storage systems only. Data can be stored in the 5G base stations, network switches, compute clusters, and storage systems. Data flows within the whole infrastructure based on the data creations, data usages, and long term destinations. How to decide the place to store the data, how to manage the data flowing, and how to help different applications to explore the data they need are the challenges to be addressed in designing a hyper-converged infrastructure for big data. Our future research will explore these interesting but challenging areas in big data infrastructure.

References

[1] Alexandros Labrinidis and Hosagrahar V Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[2] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile networks and applications*, 19(2):171–209, 2014.

[3] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)*, pages 42–47. IEEE, 2013.

[4] MarketsandMarkets. Billion by 2025 - exclusive report by marketsandmarkets. `https://www.prnewswire.com/news-releases/big-data-market-worth-229-4-billion-by-2025--exclusive-report-by-marketsandmarkets-301022839.html`, 2020.

[5] IDC. Idc forecasts revenues for big data and business analytics solutions will reach 189.1 billion this year with double-digit annual growth through 2022. `https://www.idc.com/getdoc.jsp?containerId=prUS44998419`, 2019.

[6] NKWOOD Research. Global big data market forecast 2019-2027. `https://www.inkwoodresearch.com/reports/big-data-market/`, 2020.

[7] ResearchAndMarkets.com. Global big data analytics market growth drivers and restraints during the forecast period, 2019-2027. `https://www.businesswire.com/news/home/20200305005388/en/Global-Big-Data-Analytics-Market-Growth-Drivers`, 2020.

[8] David Smith. The exponential growth of big data. https://tweakyourbiz.com/growth/productivity/big-data-growth, 2019.

[9] Ray Y Zhong, Xun Xu, Eberhard Klotz, and Stephen T Newman. Intelligent manufacturing in the context of industry 4.0: a review. *Engineering*, 3(5):616–630, 2017.

[10] Andrew Kusiak. Smart manufacturing must embrace big data. *Nature*, 544(7648):23–25, 2017.

[11] Ibrahim Abaker Targio Hashem, Victor Chang, Nor Badrul Anuar, Kayode Adewole, Ibrar Yaqoob, Abdullah Gani, Ejaz Ahmed, and Haruna Chiroma. The role of big data in smart city. *International Journal of Information Management*, 36(5):748–758, 2016.

[12] Eiman Al Nuaimi, Hind Al Neyadi, Nader Mohamed, and Jameela Al-Jaroodi. Applications of big data to smart cities. *Journal of Internet Services and Applications*, 6(1):25, 2015.

[13] Rob Kitchin. The real-time city? big data and smart urbanism. *GeoJournal*, 79(1):1–14, 2014.

[14] Amin Hosseinian-Far, Muthu Ramachandran, and Charlotte Lilly Slack. Emerging trends in cloud computing, big data, fog computing, iot and smart living. In *Technology for Smart Futures*, pages 29–40. Springer, 2018.

[15] Jianhua Li, Jiong Jin, Dong Yuan, Marimuthu Palaniswami, and Klaus Moessner. Ehopes: Data-centered fog platform for smart living. In *2015 International Telecommunication Networks and Applications Conference (ITNAC)*, pages 308–313. IEEE, 2015.

[16] Vivek Kumar. The rising trends of data storage-cloud, iot, and data center to watch in 2019. https://www.analyticsinsight.net/the-rising-trends-of-data-storage-cloud-iot-and-data-center-to-watch-in-2019/, 2019.

[17] John Gantz David Reinsel and John Rydning. The digitization of the world from edge to core. `https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf`, 2019.

[18] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158. ACM, 2009.

[19] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 08)*, volume 57, 2008.

[20] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, pages 91–103, 2011.

[21] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[22] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–11. IEEE, 2014.

[23] Eunji Lee, Hyokyung Bahn, and Sam H Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 73–80, 2013.

[24] Shiyong Liu, Zhichao Cao, Zhongwen Guo, Guohua Wang, Xupeng Wang, Zhijin Qiu, and Xukun Qin. Nvmtfs: A non-volatile memory adaptive file system for tiered storage system. In *2018 4th International Conference on Big Data Computing and Communications (BIGCOM)*, pages 201–206. IEEE, 2018.

[25] Robin Harris. Facebook's ssd findings: Failure, fatigue and the data center. `https://www.zdnet.com/article/facebooks-ssd-experience/`, 2015.

[26] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. 2016.

[27] Jon Martindale. Ssd vs. hdd. `https://www.digitaltrends.com/computing/ssd-vs-hdd/`, 2020.

[28] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[29] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[30] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[31] Varman P J Wang H. Balancing fairness and efficiency in research storage systems with bottleneck-aware allocation. pages 229–242, 2014.

[32] Samuel Burk Siewert, Nicholas Martin Nielsen, Phillip Clark, and Lars E Boehnke. Systems and methods for block-level management of tiered storage, August 5 2010. US Patent App. 12/364,271.

[33] Gong Zhang, Lawrence Chiu, and Ling Liu. Adaptive data migration in multi-tiered storage based cloud environment. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 148–155. IEEE, 2010.

[34] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. Automated lookahead data migration in ssd-enabled multi-tiered storage

systems. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[35] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[36] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, volume 8, pages 1–14, 2008.

[37] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, volume 4, page 500, 2012.

[38] Ruijin Zhou, Ming Liu, and Tao Li. Characterizing the efficiency of data deduplication for big data storage management. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 98–108. IEEE, 2013.

[39] Xue Yang, Rongxing Lu, Kim-Kwang Raymond Choo, Fan Yin, and Xiaohu Tang. Achieving efficient and privacy-preserving cross-domain big data deduplication in cloud. *IEEE transactions on big data*, 2017.

[40] Qing Liu, Yinjin Fu, Guiqiang Ni, and Rui Hou. Hadoop based scalable cluster deduplication for big data. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 98–105. IEEE, 2016.

[41] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[42] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strumm. Optimizing space amplification in RocksDB. In *CIDR*, volume 3, page 3, 2017.

[43] Sanjay Ghemawat and Jeff Dean. Leveldb. `https://github.com/google/leveldb,%20http://leveldb.org`. 2011.

[44] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[45] 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. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[46] Facebook. Cassandra on rocksdb at instagram. `https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram`. 2018.

[47] 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. IEEE, 2005.

[48] David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher, Yongdae Kim, Yingping Lu, Aravindan Raghuveer, and Sarah Sharafkandi. Experiences in building an object-based storage system based on the osd t-10 standard. In *Proceedings of MSST*, volume 6, 2006.

[49] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[50] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. Understanding write behaviors of storage backends in ceph object store. In *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology*, volume 10, 2017.

[51] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[52] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of the sixth conference on Computer systems*, pages 169–182, 2011.

[53] Yuri Demchenko, Fatih Turkmen, Cees de Laat, Christophe Blanchet, and Charles Loomis. Cloud based big data infrastructure: Architectural components and automated provisioning. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 628–636. IEEE, 2016.

[54] Ramanathan Venkatraman and Sitalakshmi Venkatraman. Big data infrastructure, data visualisation and challenges. In *Proceedings of the 3rd International Conference on Big Data and Internet of Things*, pages 13–17, 2019.

[55] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 183–198, 2013.

[56] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[57] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[58] Zhichao Cao, Hao Wen, Xiongzi Ge, Jingwei Ma, Jim Diehl, and David HC Du. Tddfs: A tier-aware data deduplication-based file system. *ACM Transactions on Storage (TOS)*, 15(1):1–26, 2019.

[59] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. Alacc: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 309–323. USENIX Association, 2018.

[60] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David HC Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 129–142, 2019.

[61] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[62] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. Joins: Meeting latency slo with integrated control for networked storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 194–200. IEEE, 2018.

[63] Fenggang Wu, Baoquan Zhang, Zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and David HC Du. Data management design for interlaced magnetic recording. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[64] Fenggang Wu, Bingzhe Li, Zhichao Cao, Baoquan Zhang, Ming-Hong Yang, Hao Wen, and David HC Du. Zonealloy: Elastic data and space management for hybrid smr drives. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[65] Fenggang Wu, Bingzhe Li, Baoquan Zhang, Zhichao Cao, Jim Diehl, Hao Wen, and David HC Du. Tracklace: Data management for interlaced magnetic recording. *IEEE Transactions on Computers*, 2020.

[66] Amandeep Khurana. Bringing big data systems to the cloud. *IEEE Cloud Computing*, 1(3):72–75, 2014.

[67] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al.

Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.

[68] William H Inmon. The data warehouse and data mining. *Communications of the ACM*, 39(11):49–51, 1996.

[69] Sandipan Ganguly, Ashish Consul, Ali Khan, Brian Bussone, Jacqueline Richards, and Alejandro Miguel. A practical approach to hard disk failure prediction in cloud platforms: Big data model for failure management in datacenters. In *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 105–116. IEEE, 2016.

[70] Victor Chang. Towards a big data system disaster recovery in a private cloud. *Ad Hoc Networks*, 35:65–82, 2015.

[71] Robert JT Morris. System for reducing storage requirements and transmission loads in a backup subsystem in client-server environment by transmitting only delta files from client to server, May 27 1997. US Patent 5,634,052.

[72] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[73] Shan Suthaharan. Big data classification: Problems and challenges in network intrusion prediction with machine learning. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):70–73, 2014.

[74] Laizhong Cui, F Richard Yu, and Qiao Yan. When big data meets software-defined networking: Sdn for big data and big data for sdn. *IEEE network*, 30(1):58–65, 2016.

[75] George Porter. Decoupling storage and computation in hadoop with superdatanodes. *ACM SIGOPS Operating Systems Review*, 44(2):41–46, 2010.

[76] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[77] Huaqun Wang. Identity-based distributed provable data possession in multicloud storage. *IEEE Transactions on Services Computing*, 8(2):328–340, 2014.

[78] Jianyong Ouyang, Chih-Wei Chu, Charles R Szmanda, Liping Ma, and Yang Yang. Programmable polymer thin film and non-volatile memory device. *Nature materials*, 3(12):918–922, 2004.

[79] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. *Proc. Dell Tech. White Paper*, pages 8–9, 2011.

[80] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating host aware smr drives. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[81] SM Hossein Tabatabaei Yazdi, Yongbo Yuan, Jian Ma, Huimin Zhao, and Olgica Milenkovic. A rewritable, random-access dna-based storage system. *Scientific reports*, 5:14138, 2015.

[82] Patrick Anderson, Richard Black, Ausra Cerkauskaite, Andromachi Chatzieleftheriou, James Clegg, Chris Dainty, Raluca Diaconu, Rokas Drevinskas, Austin Donnelly, Alexander L Gaunt, et al. Glass: A new media for a new era? In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[83] Ibm tivoli. `http://www-03.ibm.com/software/products/en/tivoli-system-automation`.

[84] Fully automated storage tiering (fast). `http://www.emc.com/corporate/glossary/fully-automated-storage-tiering.htm`.

[85] Dell storage sc8000 array controller – data center san. `http://www.dell.com/us/business/p/dell-compellent-sc8000/pd`, 2020.

[86] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, 2016.

[87] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[88] Wei Zhang, Tao Yang, Gautham Narayanasamy, and Hong Tang. Low-cost data deduplication for virtual machine backup in cloud storage. In *HotStorage*, 2013.

[89] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 331–344, 2015.

[90] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[91] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–9. IEEE, 2009.

[92] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, volume 12, pages 1–14, 2012.

[93] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: in-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, 2016.

[94] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX annual technical conference (USENIX ATC 10)*, 2010.

[95] Yoshihiro Tsuchiya and Takashi Watanabe. Dblk: Deduplication for primary block storage. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–5. IEEE, 2011.

[96] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*, 2014.

[97] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, 2016.

[98] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[99] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. Fastcdc: a fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[100] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 15. ACM, 2012.

[101] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 581–586. IEEE, 2011.

[102] Young Jin Nam, Dongchul Park, and David HC Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 201–208. IEEE, 2012.

[103] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based

backup systems via exploiting historical information. In *USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, 2014.

[104] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Sar: Ssd assisted restore optimization for deduplication-based storage systems in the cloud. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, pages 328–337. IEEE, 2012.

[105] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[106] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[107] Dan E Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, 1984.

[108] John Rydning David Reinsel, John Gantz. The digitization of the world from edge to core. `https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf`, 2018.

[109] Liu Y. Big data Chen M, Mao S. Big data: A survey. *Mobile Networks and Applications*, 19(2): 171-209, 2014.

[110] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, volume 2, 2002.

[111] J. bonwick. zfs deduplication, november 2009. `https://blogs.oracle.com/bonwick/entry/zfs_dedup`.

[112] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 10)*, pages 1–10. IEEE, 2010.

[113] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI 06)*, pages 307–320. USENIX Association, 2006.

[114] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 08)*, volume 1, pages 2–5, 2008.

[115] Kitchin R. *The data revolution: Big data, open data, data infrastructures and their consequences*. Sage, 2014.

[116] Sr. Product Marketing Manager Paul Feresten and Technical Director Rajesh Sundaram. Automated storage tiering and the netapp virtual storage tier. `https://community.netapp.com/t5/Tech-OnTap-Articles/Automated-Storage-Tiering-and-the-NetApp-Virtual-Storage-Tier/ta-p/84825`, 2011.

[117] Anant Baderdinni. Relative heat index based hot data determination for block based storage tiering, July 2 2013. US Patent 8,478,939.

[118] Sangwook Kim, Hwanju Kim, Sang-Hoon Kim, Joonwon Lee, and Jinkyu Jeong. Request-oriented durable write caching for application performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 15)*, pages 193–206, 2015.

[119] Gpfs. `https://www.ibm.com/support/knowledgecenter/en/STXKQY$_$4.1.1/com.ibm.spectrum.scale.v4r11.adv.doc/bl1adv$_$afm.htm`.

[120] Refs. `https://docs.microsoft.com/en-us/windows-server/storage/refs/refs-overview`.

[121] Oracle hierarchical storage manager. `http://www.oracle.com/us/products/servers-storage/storage/035865.pdf`, 2020.

[122] Drew Roselli and Thomas E Anderson. *Characteristics of file system workloads.* University of California, Berkeley, Computer Science Division, 1998.

[123] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 00)*, pages 41–54, 2000.

[124] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[125] Yinjin Fu, Hong Jian, Nong Xiao, Lei Tian, and Fang Liu. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 112–120. IEEE, 2011.

[126] Danny Harnik, Ety Khaitzin, and Dmitry Sotnikov. Estimating unseen deduplication from theory to practice. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 277–290, 2016.

[127] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[128] Jyoti Malhotra and Jagdish Bakal. A survey and comparative study of data deduplication techniques. In *Pervasive Computing (ICPC), 2015 International Conference on*, pages 1–5. IEEE, 2015.

[129] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[130] Guanlin Lu, Young Jin Nam, and David HC Du. Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[131] Fuse, may 2015. `http://fuse.sourceforge.net/`.

[132] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[133] Dell sc series. http://en.community.dell.com/techcenter/extras/m/white_papers/20442763/download.

[134] filebench, july 2014. https://github.com/filebench/filebench/wiki.

[135] Intel 750 series 400gb ssd, https://www.amazon.com/Intel-Single-400GB-Solid-SSDPE2MW400G4X1/dp/B011I61L70/ref=sr_1_4?s=electronics$&$ie=UTF8$&$qid=1523731878$&$sr=1-4$&$keywords=Intel+750+Series+400GB+SSD.

[136] Seagate 6t enterprise hdd (st6000nm0024), https://www.amazon.com/Seagate-BarraCuda-3-5-Inch-Internal-ST6000DM004/dp/B01LOOJBH8/ref=sr_1_fkmr0_3?s=electronics$&$ie=UTF8$&$qid=1523731981$&$sr=1-3-fkmr0$&$keywords=segeat+ST6000NM0024.

[137] Aaron Brown and Kristopher Kosmatka. Block-level inline data deduplication in ext3. *University of Wisconsin-Madison Department of Computer Sciences, http://pages.cs.wisc.edu/~kosmatka/dedupfs/paper.pdf*, 2010.

[138] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference (USENIX ATC 14)*, pages 501–512, 2014.

[139] Lessfs, january 2012. http://www.lessfs.com/wordpress/.

[140] Opendedup – sdfs, januray 2012. http://www.opendedup.org.

[141] Dongchul Park, Ziqi Fan, Young Jin Nam, and David HC Du. A lookahead read cache: Improving read performance for deduplication backup storage. *Journal of Computer Science and Technology*, 32(1):26–40, 2017.

[142] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, volume 2, pages 89–101, 2002.

[143] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 09)*, volume 9, pages 197–210, 2009.

[144] Ziqi Fan, Fenggang Wu, Dongchul Park, Jim Diehl, Doug Voigt, and David HC Du. Hibachi: A cooperative hybrid cache with nvram and dram for storage arrays. In *Mass Storage Systems and Technologies (MSST), 2017 IEEE 33th Symposium on*. IEEE, 2017.

[145] Fsl traces and snapshots public archive. `http://tracer.filesystems.org/`, 2020.

[146] Nohhyun Park and David J Lilja. Characterizing datasets for data deduplication in backup applications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[147] Roderick Bauer. Hdd vs ssd: What does the future for storage hold? `https://www.backblaze.com/blog/hdd-vs-ssd-in-data-centers/`, 2018.

[148] Michal Kaczmarczyk and Cezary Dubnicki. Reducing fragmentation impact with forward knowledge in backup systems with deduplication. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 17. ACM, 2015.

[149] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):855–868, 2016.

[150] Yujuan Tan, Baiping Wang, Jian Wen, Zhichao Yan, Hong Jiang, and Witawas Srisa-an. Improving restore performance in deduplication-based backup systems via a fine-grained defragmentation approach. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[151] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. A cost-efficient rewriting scheme to improve restore performance in deduplication systems. In *Proc. MSST*, 2017.

[152] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. Improving restore performance in deduplication systems via a cost-efficient rewriting scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[153] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *USENIX Annual Technical Conference (USENIX ATC 12)*, pages 261–272, 2012.

[154] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, et al. A long-term user-centric analysis of deduplication patterns. In *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on*, pages 1–7. IEEE, 2016.

[155] Edward Grochowski. Emerging trends in data storage on magnetic hard disk drives. *Datatech (September 1998), ICG Publishing*, pages 11–16, 1998.

[156] Feng Chen, David A Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, pages 22–32. ACM, 2011.

[157] Benny Van Houdt. Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data. *Performance Evaluation*, 70(10):692–703, 2013.

[158] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230. ACM, 2013.

[159] Devesh Tiwari, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J Desnoyers. Reducing data movement costs using energy-efficient, active computation on ssd. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, 2012.

[160] Takeshi Shibata, SungJun Choi, and Kenjiro Taura. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems.

In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 746–755. ACM, 2010.

[161] Erik Riedel and Garth Gibson. Active disks-remote execution for network-attached storage. Technical report, Carnegie-Mellon University, School of Computer Science, 1997.

[162] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[163] Erik Riedel, Christos Faloutsos, Garth A Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[164] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *IEEE 29th symposium on mass storage systems and technologies (MSST 13)*, pages 1–12. IEEE, 2013.

[165] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. Intelligent ssd: a turbo for big data mining. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1573–1576. ACM, 2013.

[166] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[167] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 4. ACM, 2016.

[168] Dongchul Park, Jianguo Wang, and Yang-Suk Kee. In-storage computing for hadoop mapreduce framework: Challenges and possibilities. *IEEE Transactions on Computers*, 2016.

[169] Shengwen Liang, Ying Wang, Cheng Liu, Huawei Li, and Xiaowei Li. Ins-dla: An in-ssd deep learning accelerator for near-data processing. In *IEEE 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 173–179. IEEE, 2019.

[170] Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for search engines. *IEEE Transactions on Computers*, 2016.

[171] Samsung. Samsung key value ssd enables high performance scaling. `https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf`, 2017.

[172] Seagate kinetic hdd product manual. `https://www.seagate.com/www-content/product-content/hdd-fam/kinetic-hdd/_shared/docs/kinetic-product-manual.pdfSamsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf`, 2014.

[173] Manas Minglani, Jim Diehl, Xiang Cao, Binghze Li, Dongchul Park, David J Lilja, and David HC Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 501–510. IEEE, 2017.

[174] Xiang Cao, Manas Minglani, and David Hung-Chang Du. Data allocation of large-scale key-value store system using kinetic drives. In *IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 60–69. IEEE, 2017.

[175] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[176] Dharmesh Desai. Apache hbase. `https://www.qubole.com/blog/advantage-decoupling/`, 2019.

[177] Amazon s3. https://aws.amazon.com/cn/s3/, 2019.

[178] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 143–157. ACM, 2011.

[179] M Zotes Resines, SS Heikkila, D Duellmann, G Adde, R Toebbicke, J Hughes, and L Wang. Evaluation of the huawei uds cloud storage system for cern specific data. In *Journal of Physics: Conference Series*, volume 513, page 042024. IOP Publishing, 2014.

[180] Apache hbase. https://hbase.apache.org/, 2019.

[181] Lars George. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.", 2011.

[182] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.

[183] 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.

[184] Apache hdfs users guide. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html, 2020.

[185] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53(1-13):2, 2008.

[186] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *IEEE 26th symposium on mass storage systems and technologies (MSST 10)*, volume 10, pages 1–10, 2010.

[187] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[188] Hui Sun, Wei Liu, Jianzhong Huang, and Weisong Shi. Co-kv: A collaborative key-value store using near-data processing to improve compaction for the lsm-tree. *arXiv preprint arXiv:1807.04151*, 2018.

[189] Hui Sun, Wei Liu, Zhi Qiao, Song Fu, and Weisong Shi. Dstore: A holistic key-value store exploring near-data processing and on-demand scheduling for compaction optimization. *IEEE Access*, 6:61233–61253, 2018.

[190] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[191] Iptables. `https://en.wikipedia.org/wiki/Iptables`, 2019.

[192] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST 15)*, pages 1–14, 2015.

[193] Yoshinori Matsunobu. Innodb to myrocks migration in main mysql database at facebook. USENIX Association, May 2017.

[194] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[195] Facebook. Rocksdb trace, replay, analyzer, and workload generation. `https://github.com/facebook/rocksdb/wiki/RocksDB-Trace%2C-Replay%2C-Analyzer%2C-and-Workload-Generationr`, 2019.

[196] Facebook. Myrocks. `http://myrocks.io/`, 2019.

[197] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[198] Redis. Redis documentation. `https://redis.io/documentation`, 2019.

[199] 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.

[200] Facebook. Rocksdb. `https://github.com/facebook/rocksdb/`, 2019.

[201] Jiayi Wang. Myrocks: best practice at alibaba. `https://www.percona.com/live/17/sessions/myrocks-best-practice-alibaba`, 2017.

[202] Satheesh Nanniyur. Sherpa scales new heights. `https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights`, 2015.

[203] Ankit Gupta. Followfeed: Linkedin's feed made faster and smarter. `https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter`, 2016.

[204] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at facebook. In *Proceedings of the International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[205] Muthu Annamalai. Zippydb: a modern, distributed key-value data store. `https://www.youtube.com/watch?v=DfiN7pG0D0k`, 2015.

[206] Facebook. Single delete. `https://github.com/facebook/rocksdb/wiki/Single-Delete`. 2019.

[207] Facebook. Merge operator. `https://github.com/facebook/rocksdb/wiki/Merge-Operator`, 2018.

[208] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[209] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[210] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[211] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[212] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[213] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. Speicher: Securing lsm-based key-value stores using shielded execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, 2019.

[214] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. Slm-db: single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[215] Jonathan RM Hosking and James R Wallis. Parameter and quantile estimation for the generalized pareto distribution. *Technometrics*, 29(3):339–349, 1987.

[216] MathWorks. Power series. `https://en.wikipedia.org/wiki/Power_series`. 2019.

[217] MathWorks. Power series models. `https://www.mathworks.com/help/curvefit/power.html`. 2019.

[218] MathWorks. Sine fitting. `https://www.mathworks.com/matlabcentral/fileexchange/66793-sine-fitting`. 2019.

[219] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *IEEE International Symposium on Workload Characterization (IISWC 08)*, pages 119–128. IEEE, 2008.

[220] Alma Riska and Erik Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference (ATC 06)*, pages 97–102, 2006.

[221] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

[222] Martin F Arlitt and Carey L Williamson. Web server workload characterization: The search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):126–137, 1996.

[223] Weisong Shi, Randy Wright, Eli Collins, and Vijay Karamcheti. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW 02)*, 2002.

[224] Adepele Williams, Martin Arlitt, Carey Williamson, and Ken Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

[225] 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 (SOSP 11)*, pages 1–13. ACM, 2011.

[226] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. In *Proceedings of the VLDB Endowment*, volume 3, pages 1414–1425. VLDB Endowment, 2010.

[227] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 25–36, 2011.

[228] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[229] Facebook. db_bench. `https://github.com/facebook/rocksdb/wiki/Benchmarking-tools`, 2019.