Design and Implementation of Data Reduction Enabled Distributed File System

March 2022

Ryan Nathanael Soenjoto Widodo

Design and Implementation of Data Reduction Enabled Distributed File System

Graduate School of
Systems and Information Engineering

University of Tsukuba

March 2022

Ryan Nathanael Soenjoto Widodo

Abstract

Distributed file systems (DFSs) play a crucial role in modern society that relies heavily on data. They leverage computer networks to organize multiple storage nodes into a storage cluster, which is expandable by adding more storage hardware or storage nodes. These systems can also work in tandem with data reduction schemes such as deduplication and compression to improve their storage efficiency at the cost of higher data access latency. To minimize these costs, some aspects of data reduction schemes in DFSs that can be improved, are the data reduction scheme, the integration of the schemes in the DFSs, and the metadata system that supports the data reduction schemes and the DFSs.

This study explores the three important aspects of data reduction schemes applications in DFS environments. In each aspect, it discusses the challenges and proposes a solution to the presented challenges. The solutions proposed in this study can improve the performance, ease of use, and compatibility of the solution with modern storage mediums.

Data reduction schemes may vary among each other in terms of the tradeoff between processing time and data reduction performance. In data deduplication, the main component that directly impacts these metrics is the chunking algorithm. Among the existing chunking algorithm categories, content-dependent chunking (CDC) algorithms are the more effective algorithms for detecting similarities among files with different versions. However, these algorithms commonly have a long processing time, which is unsuitable for some use cases. The proposed method extends the existing chunking algorithm and reduces its number of comparisons by improving its conditional structure, improving the chunking throughput by up to 49% and efficiency in reducing redundancies by up to 40% depending on the dataset.

The selection of data reduction schemes in the DFSs is crucial to minimize the redundancy in the data because the schemes may vary in performance depending on the dataset. However, the existing solutions proposed by studies and projects such as Hadoop suffer from the difficulty of integrating and enabling the schemes to the DFSs. For example, adding a new scheme, which may have a better performance than the schemes in the existing selection, can be tedious and sometimes might

even be impossible without the help of the maintainers of the DFSs. The proposed DFS design uses a dynamic library approach to allow users to compile the schemes separately from the DFS and use them in the DFS without any change in the DFS or the application code. The experimental results show that the proposed DFS design can enable data reduction schemes in the DFS with around 1% of overhead in data transfer performance through the platform-provided tools and around 5% of overhead in data processing applications.

Key-value stores (KVSs) are crucial for metadata systems to store and serve blocks and chunks of information in the DFSs and deduplication. Many KVSs utilize a data structure called log-structured merge-tree (LSM-tree) to store the data because of its scalability, performance in range query workloads, and compatibility with existing storage mediums because writes and updates are sequential in LSM-tree. However, the use of non-volatile memory (NVM) in the LSM-tree might be detrimental to the performance of the KVS because of its unique characteristic, where write throughput may degrade when too many threads submit write operations in parallel. The proposed KVS solves the performance degradation from parallel writes in NVM by using asynchronous multithreading to decouple the client threads from the NVM, increasing the consistency of write operations and the throughput of the tested KVSs by over double.

Through the discussion on the challenges in these three aspects and the presented solutions, this study provides an insight into the existing and future DFS designs in terms of data reduction schemes usage in DFS environments. Additionally, each presented solution can work together to improve the existing DFSs or independently in the existing systems that utilize deduplication, DFS, or KVS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The reliance on data in modern societies drives the requirement for storage systems to handle large amounts of data, which are useful for data-intensive applications to provide a good insight into the environment and society. For example, data mining on users' comments in blogs or social media can provide information on the direction of the society [1, 2]. Additionally, medical data such as logs from electrocardiogram machines (ECG) can help medical researchers to better understand human health [3, 4, 5]. Applications ranging from commercial to research store and use data to achieve their goals, promoting the exponential growth of data [6, 7].

Distributed file systems (DFS), storage systems that have multiple storage nodes unified through software that connect these nodes through a network, are the obvious choices to sustain the data growth because of their scalability. Most DFSs can expand their storage capacity by adding storage devices to storage nodes and increasing the number of storage nodes. However, simply storing the data in the DFS and adding more storage hardware or nodes to the DFS can be wasteful. These data might have suboptimal structures where redundancies may exist within them, reducing the efficiency of the storage system and leaving potential free space unusable.

Data reduction schemes such as lossless compression and deduplication operate through data restructuring processes that decrease the amount of redundancy within the data. For example, deduplication reduces the storage footprint by replacing the redundant parts of the data with pointers to the existing stored data. These schemes are reversible and can be transparent to the users. Through these schemes, storage systems such as DFS can transparently optimize their storage space utilization at the cost of computation time, which can be negligible depending on their complexity.

Data reduction schemes have different characteristics and trade-offs between processing time and space reduction and may vary in performance depending on the

data type; some schemes perform faster than others but reduce less space; some schemes work better at specific data types. For example, Logzip [8], a compression scheme for log files, is slower than Gzip, a general-purpose compression scheme, but can reduce more redundancy in log data. From this point of view, the users should select the most suitable schemes for their data type and their requirements. However, the selection of schemes in the existing applications or DFSs is often limited and challenging to extend.

The use of data reduction schemes in the DFS proposed by many studies [9, 10] and projects such as Hadoop [11] has several limitations in the aspect of schemes selection and ease of use. For example, extending the scheme selections can be challenging and the schemes are only available in some Hadoop applications. Additionally, the data reduction schemes may have a big impact on the performance of the DFS.

Some data reduction schemes and DFSs rely on a metadata system to manage information on the stored data. For example, MAD2 deduplication scheme [12] utilizes a metadata system to identify redundant data, and Hadoop DFS (HDFS) [11] uses a metadata system in the storage node and the name node to manage the data in the DFS. This system is crucial to the schemes and the DFSs and can directly affect their throughput and latency. However, it may suffer from performance degradation when the data grows.

We believe that a data reduction enabled DFS should have minimal overhead, provide users the ease of access for data reduction schemes, and scale well with data sizes. Users should be able to choose the schemes that have a good balance of processing time and storage space reduction and use the DFS with growing data size without any concern on performance degradation from the metadata server. To achieve this goal, an improvement in the data reduction scheme, in the design of DFS software, and in the metadata system of the DFS can improve the usability of data reduction schemes in DFS.

1.1 A balance between processing time and reduced size

Deduplication is a data reduction scheme that removes redundancies within a set of files. In this sense, it is a good solution for storage systems that manage large data like DFS because it is more effective with more files or larger data, which have a higher probability of finding duplicates in the larger data pool. However, deduplication has an issue in the trade-off between processing time and reduced

space, which stems from the chunking algorithm.

In deduplication, the chunking algorithms, which split the input data into chunks and determine their sizes, are the most important component. They affect the duplicate elimination performance of the deduplication system directly. The simplest chunking algorithm is fixed-sized chunking, which produces equal-sized chunks. It can detect byte changes among similar files, but not byte shifting, which occurs when a byte is inserted into the data. Content-dependent chunking (CDC) algorithms solve this by determining the cut-point of the chunks based on the content of the data, which produces variable-length chunks. However, CDC algorithms are compute-intensive and might significantly impact the performance of the DFS.

1.2 Data reduction schemes in the DFS

Like any other FSs, DFS is software that manages data access between the storage nodes and the application. The storage nodes commonly employ other FSs to manage the storage hardware. For example, users can use Hadoop DFS (HDFS) on top of ext4, NTFS, and ZFS FSs. Based on these facts, we can create a high-level view of data reduction scheme activation in the DFS, as shown in 1.1a. At the application layer, users can generate data by using applications, apply the data reduction schemes by adding data reduction code to the application or through another application, and store it to the DFS as shown by Zhang et al in their study on using deduplication through MapReduce [10]. At the FS layer, users can enable the data reduction through the FS and its configuration. For example, some FSs such as ZFS accept configuration to enable compression or deduplication [9]. At the DFS layer, currently, there is no implementation of data reduction schemes directly in the DFS software. These schemes only run at the application or the underlying FS.

Challenges of adding and enabling data reduction schemes in the DFS. The main benefit of using data reduction schemes in the FS layer is the transparency of the process. It requires no change in the application code to enable the schemes, minimizing the users' effort in reducing storage footprint. However, adding new schemes, which is important because data might come in different types and structures, is challenging because these FS are more complex than the application code. Finding the correct location to add the new scheme's code can be time-consuming and might break the FS.

Hadoop solves this by using programming techniques such as the dynamic library, which is loaded during the application's execution. Hadoop uses CompressionCodecs, which can be compiled separately from the application code. The user can simply enable the reduction schemes in the application through the application

configuration file or command line. However, this approach is only possible when the application supports CompressionCodecs. Additionally, users must add the support for CompressionCodecs to all applications to benefit from the reduced storage footprint. Data reduction at the DFS layer or software can theoretically offer both transparency and ease of adding new schemes. However, such DFS software does not exist yet.

Reduction in network traffic. Although the data spend most of their time in the storage drives, the data must also travel through the network to reach the storage nodes and the processing nodes. Network traffic growth is also solvable by using data reduction schemes. However, not all data reductions in the DFS have equal benefits to the network traffic. Users who opted for data reduction through the FS layer because of its transparency will have to deal with the larger network traffic because the network operation occurs in the DFS layer and the reduction occurs in the separate lower layer. On the other hand, application layer-side data reduction can produce less network traffic at the DFS layer because the reduced data applies to the lower layers. Before this work, as far as our knowledge, there are no solutions that provide the ease of adding and enabling the scheme and reduction in network traffic.

1.3 A high-performance metadata system

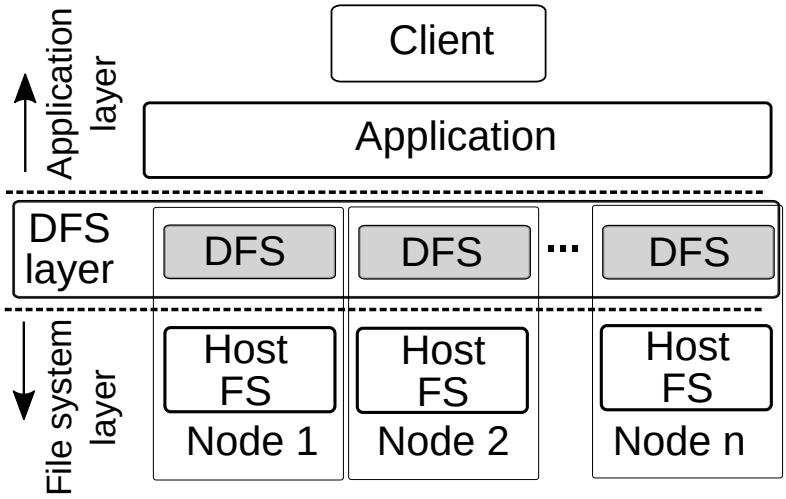
In addition to these challenges, DFSs and some of the data reduction schemes rely on a metadata system. In the DFS, this metadata system manages the location of the data in the storage nodes, such that the client can query the server and access the data in the storage nodes. In deduplication schemes, the metadata system manages the location of the unique parts of the data, such that the users can recover data from a pointer by querying the metadata system. In both use cases, this system can directly affect the performance of the application.

An engine of most metadata systems is a key-value store (KVS), which provides the value when given the key. Log-structured merge-tree (LSM-tree) is one of the solutions for metadata systems [13]. It is scalable and can offer good performance for write-intensive workloads. However, once the data grows, LSM-trees may suffer from performance degradation and high write amplification, which can shorten the lifetime of the storage device.

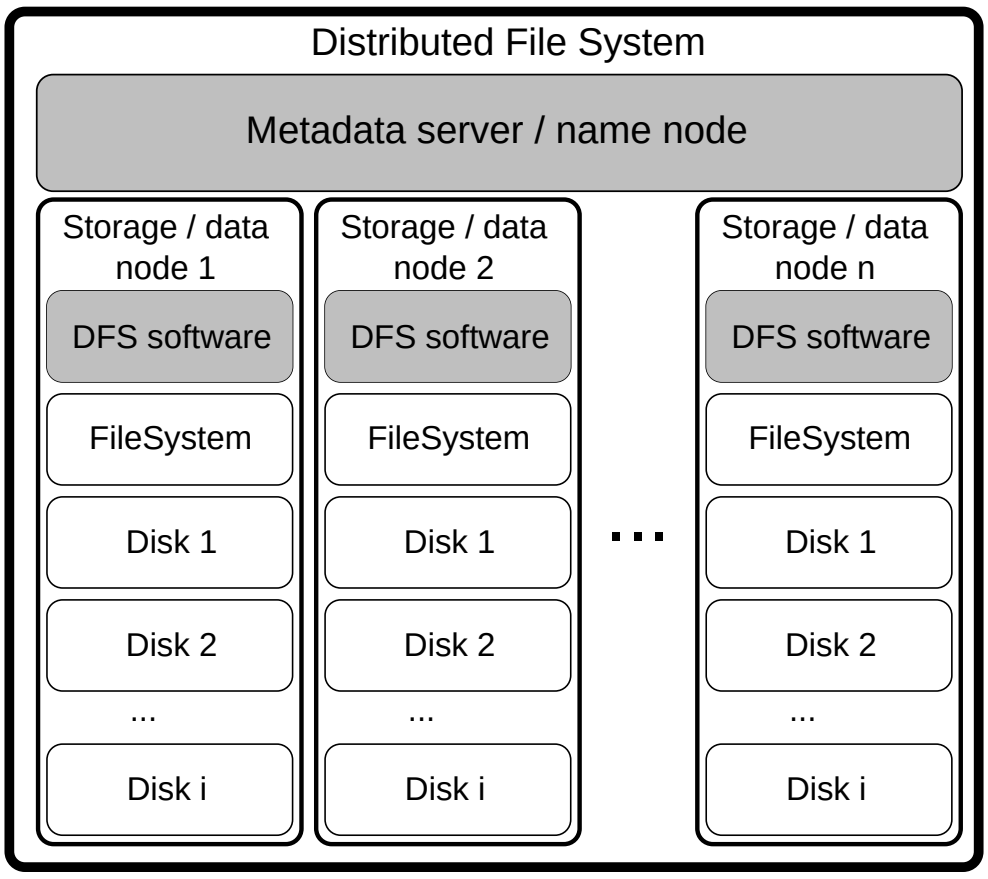
1.4 Thesis statement and approach

The central thesis of this dissertation is data reduction schemes and the DFS design that supports them. This thesis explores the key building blocks of these schemes in the DFS, which are: the reduction scheme, the DFS software, and the metadata system. We discuss and review the building blocks, answer the following questions, and reviews the answers in separate Sections.

- Is it possible to get a better trade-off between processing time and storage space reduction for deduplication when using CDC algorithms? In Section 2, we studied CDC chunking algorithms and proposed a new chunking algorithm to obtain a better trade-off between processing time and storage space reduction.
- Are ease of adding and enabling data reduction schemes achievable in the DFS environment? How big is the overhead compared to the existing approaches? How feasible is the proposed design at minimizing the network traffic? In Section 3, we discussed the challenges of using these schemes in DFSs, proposed and implemented a DFS software design that can perform these schemes, and evaluated the design by comparing it to the existing approaches.
- Can non-volatile memory (NVM) improve the performance of log-structured merge-tree (LSM-tree) based key-value store (KVS)? In the section, we detailed the challenges of LSM-tree as a write-intensive KVS and engineered a solution that uses NVM to address the challenges.



(a) A common DFS design divided into several layers.



(b) A common structure of a DFS.

Figure 1.1: 1.1a: The different layers where data reduction schemes can run in the DFS. 1.1b: An example of a DFS with n nodes and i storage disks in each node. The grayed parts are the DFS components.

Chapter 2

A high-throughput hash-less content-dependent chunking (CDC) algorithm

2.1 Introduction

Chunking algorithms are one of the most crucial components of deduplication schemes. They directly affect the scheme's performance at finding duplicates in the data and the scheme's throughput depending on their complexity [14]. These algorithms split the input data into smaller blocks of data called chunks to detect the redundant part of the data. Assuming there are similar files with some changes within them, the chunking algorithms split the files into chunks that may share some content, which is identified by using fingerprinting the chunks through a mathematical hash function such as SHA-1.

In similar files or files that differ by version, changes may occur from byte change and byte insertion. The chunking algorithm can localize the byte change by splitting the files into chunks. In this approach, the chunk size can be fix-sized, which requires minimal computation, and the size of affected data is minimized into a few affected chunks. A single instance of each chunk is stored by deduplication, thus minimizing the footprint of the data. However, fix-sized chunks may miss duplicate chunks with byte insertion because it shifts the file.

Byte insertion is more challenging because a content shift in the file may result in totally different chunks' fingerprints. For example, a string with the content of "an apple pie" is split into 4 equal-sized chunks: "an ", "app", "le ", and "pie". With a byte insertion at the beginning of the string, the chunks may produce different fingerprints, which are unidentifiable by the deduplication scheme. For example,

"xan apple pie" string produces 5 equal-sized chunks with padding: "xan", " ap", "ple", " pi", and "e ". Therefore, deduplication schemes with a naive approach of chunking such as fix-sized chunking may consume more space when processing files with byte shifting.

CDC algorithms solve byte shifting by defining the cut-point based on the content of the data. The most traditional approach is through a lightweight hashing algorithm such as Rabin-rolling hash or gear hash [15], which uses a fix-sized sliding window. It defines a cut-point by matching the hash of the window with a pre-defined pattern. This process is computation-intensive even when Rabin-rolling has reuses the value of the previous windows to calculate the value of the current window.

Hash-less CDC algorithms solve this by using byte-values and conditions to define the cut-point [14, 16]. The first to propose this method is Bjørner et al. by using two fix-sized sliding windows that clamp a byte in the between in an algorithm called local maximum chunking (LMC). The condition defines a cut-point when the byte in the middle of the windows is larger than any bytes in the windows. The main drawback of this conditional algorithm is the number of comparisons can be as high as the total number of bytes in the two windows, which in practice and experiment [14, 17], can be as slow as hash-based chunking algorithm.

Zhang et al. [14] took the idea of using byte values and conditions and improved it by changing the sliding window structure and the condition. In their work, they proposed a hash-less CDC algorithm called Asymmetric Extremum (AE), which used a single fix-sized sliding window. the condition defines a cut-point when the byte behind the window is an extreme value byte, which is either smaller or larger than the byte in the window as shown in Algorithm 1. This algorithm claims to produce up to 3-times of LMC throughput. However, there are still some possible improvements in the condition's structure to further improve the throughput.

In this work, we proposed an algorithm called Rapid Asymmetric Extremum (RAM) based on AE. RAM further optimizes the window structure and conditions of AE, improving the throughput with a negligible decrease in duplicate finding performance. Our experimental results indicate that RAM can produce 5-times the throughput of LMC or 1.4 times of AE at the cost of 2% to 18% lower found duplicate data depending on the dataset type.

2.2 Background and motivation

This section discusses the background and motivation of CDC algorithms, related work, their limitations, and motivation of our work.

Algorithm 1: The pseudo-code for AE for maximum byte value

```
Data: input string Str, length L
Result: cut-point i
Predefined values Window size w;
i ← 1;
max.value ← 0;
max.position ← 0;
while i <L do
    // Outer condition
    if Str[i] <= max.value then
        // Inner condition
        if i=(max.position + w) then
            return i;
        end
    else
        max.value = Str[i].value;
        max.position = i;
    end
    i = i + 1;
end
```

2.2.1 Background

Chunking is used in many data compression applications. For example, it is used in data deduplication and remote differential compression. Data deduplication works by eliminating duplicate data within the files and between files. In data deduplication, a chunking algorithm is one of the vital parts to achieve high duplicate elimination. By choosing the correct chunking method, we can save time and space [18]. Data deduplication can be applied on cloud storage [19], virtual disk images [20], memory [21, 22], and network traffic [23].

Chunking algorithms can be categorized into two categories: (i) whole file chunking and (ii) block chunking. Whole file chunking means the whole file is treated as one chunk, while block chunking means the file is split into multiple chunks. When chunking a file into blocks or chunks, the chunk size can be fixed-sized or variable-sized. Fixed-sized chunking is fast and not resistant to byte insertion or shifting. When the file is shifted by a byte insertion or deletion, the chunks will become completely different chunks and undetectable by the chunk duplicate search. Content Defined Chunking (CDC) solves this problem by chunking the file into variable-sized chunks. CDC algorithms find the cut-point by using internal features of the file. Therefore, when the file is shifted, only several chunks are affected. CDC

has a higher probability of eliminating duplicates within the files and between files compared to fixed-sized chunking.

One of the oldest CDC algorithms is Rabin [24] based CDC algorithm. It finds the cut-point by using Rabin rolling hash. Rabin rolling hash uses a sliding window and every time the window is moving, a hash value is calculated. When the hash value matches a predefined value, it uses the window position for the hash value as a cut-point. Since the checksum is calculated based on polynomials over a finite field, the old checksum can be used to calculate the new checksum when the window slides.

Another CDC algorithm proposed by Bjørner et al. [16] is Local Maximum Chunking method (LMC). LMC finds the maximum valued byte or local maximum byte by using a sliding window. The sliding window has three components: (i) the local maximum byte, (ii) the left window, and (iii) the right window. The local maximum byte is located in between the two fixed-sized windows as illustrated in Figure 1(a). LMC defines a byte as the cut-point when the local maximum byte is larger than all of the bytes in the fixed windows. When this condition is fulfilled, a cut-point is found. The left fixed window will be included in the chunk and the minimum chunk size is the size of the window.

To address the slow chunking performance of LMC, Zhang et al. [14] proposed Asymmetric Extremum (AE) algorithm. The algorithm is similar to LMC in terms of treating a byte as digits. As illustrated in Figure 1(b), AE uses two windows, a variable-sized window on the left and a fix-sized window on the right. The extreme-valued byte is located in the middle of the two windows. A cut-point is found when the extreme-valued byte is bigger or smaller compared to all the values in the two windows. The minimum chunk size is the size of the window. Zhang et al. [14] explain the performance difference between maximum and minimum for the extreme-valued byte is negligible. Therefore, we used only maximum for the extreme-valued byte in our analysis and performance evaluation.

2.2.2 Challenges and motivation

CDC algorithms offer more benefits than fix-sized chunking. However, their processes are slightly more time-consuming which limits their use in latency-critical applications and on devices with limited processing capability such as mobile devices and Internet of Things (IoT) devices. In our previous work [18], we used a Rabin-based chunking algorithm for the deduplication system to eliminate duplicate data. We found out that the main drawback of using CDC algorithms in mobile applications is its large processing time.

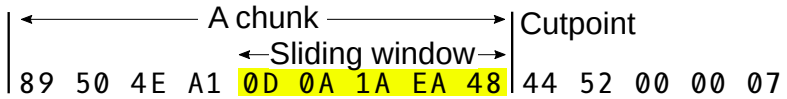
In this work, we conform to the following criteria that Zhang et al. in [14] use to compare CDC algorithms:

- **Content dependent.** The algorithm should define the cut-point based on the internal features of the file, which makes it resistant against byte shifting and allows the algorithm to find duplicate chunks between two or more files.
- **Low chunk sizes variance.** The chunks produced by the algorithm should have low chunk variance because it might affect the deduplication efficiency [25]. To limit the chunk variance, we can add a limit on the maximum or minimum size of the chunks. However, this will affect the content-dependent properties of the algorithm and make the algorithm vulnerable to byte shifting.
- **Ability to eliminate low entropy strings.** Low entropy strings are strings that consist of repetitive bytes or patterns. When it encounters strings with low entropy or low variance, the algorithm should be able to eliminate the redundancy within the string.
- **High throughput and duplicate detection.** The algorithm should have a good balance between deduplication performance and computational overhead.

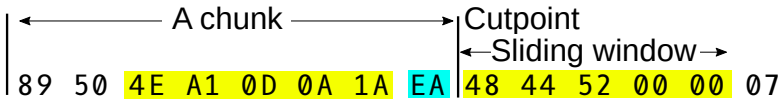
Rabin-based CDC algorithms use polynomial over a finite field and a sliding window to calculate the hash [24]. Calculating the hash is fast because it only needs to consider the new byte and the most left byte in the window. Rabin-based CDC algorithms have a few disadvantages due to the use of the hash. It is time-consuming because of the hash calculation, and changing a byte in the chunk has a high probability of changing the cut-point as it might create a different hash value. It also has a large chunk variance because of the higher probability of having a long chunk [14,16]. In order to limit the chunk variances, we can use a limit on the chunk size. However, this will reduce the resistance of the algorithm against byte shifting.

Local Maximum Chunking (LMC) [16] is a CDC algorithm that compares bytes with bytes as a number to find the cut-point. LMC has resistance against byte changing and byte shifting. When there is a change in the chunk and the change has a value less than the maximum, it will only affect that chunk. The main drawback of this method is the requirement of rechecking all the bytes within the window when the window slides. This drawback makes Rabin-based CDC algorithms faster than LMC method because when the sliding window of Rabin slides, it only needs to subtract the most left byte and add the new byte into the hash. However, LMC needs all of the bytes in the window every time it slides the window.

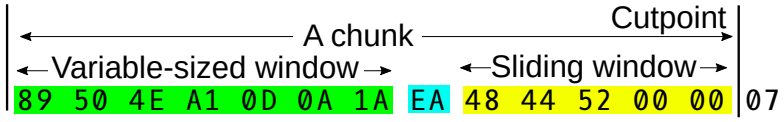
AE is similar to the local maximum method because it treats a byte as a number. Treating the chunk as the windows allows AE to have a lower computational



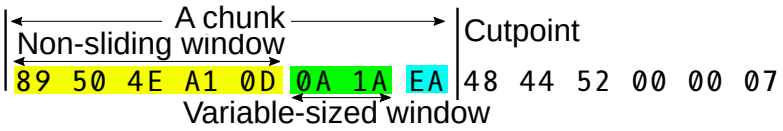
(a) Rabin based CDC algorithms



(b) LMC



(c) AE



(d) RAM

Figure 2.1: The window structure in 2.1a Rabin based CDC algorithms, 2.1b LMC, 2.1c AE, and 2.1d RAM.

overhead than the LMC method. However, unlike the LMC method, AE puts the extreme-valued byte in the middle of the chunk. This makes AE less resistant to byte shifting. When there is a byte inserted at the fixed window, it will affect the chunk and the next chunk and might affect subsequent chunks. If we put the extreme-valued byte at the boundary of the chunk, inserting a byte will not affect the next chunk. Thus, it minimizes the number of affected bytes. AE is capable of eliminating low entropy strings because AE has maximum chunk size. AE reaches its maximum chunk size when it processes a long increasing sequence. The maximum chunk size is the length of the fixed window. The pseudo-code for AE chunking is illustrated in Figure 2.1c.

In this work, we proposed a new algorithm called Rapid Asymmetric Maximum (RAM) which improves the chunking throughput of AE by putting the extreme value at the boundary of the chunk. It has a low computational overhead which makes the algorithm faster than existing CDC algorithms. The low computation overhead of RAM reduces the cost of the chunking process which makes chunking more attractive over AE for low-performance devices such as mobile devices and IoT. Based on the ideal CDC algorithm criteria, we compared the existing CDC algorithms and our proposed algorithm in Table 1.

Algorithm 2: The pseudo-code for RAM

```
Data: input string Str, length L
Result: cut-point i
Predefined values Window size w;
i ← 1;
max.value ← 0;
max.position ← 0;
while i >L do
    // Outer condition
    if Str[i] >= max.value then
        //Inner condition
        if i >=w then
            return i;
        end
        max.value = Str[i].value;
        max.position = i;
    end
    i = i + 1;
end
```

2.3 Rapid Asymmetric Maximum (RAM)

RAM is an algorithm based on AE. It uses the same single sliding window structure with a slight tweak in the conditions. However, unlike AE, which supports maximum and minimum byte value configurations, RAM only works with maximum byte value configuration like LMC. As a CDC algorithm, RAM improves over AE in the number of average comparisons at the cost of higher probability of long chunks for low entropy strings, which are strings with a long repetition of 0 or 1, increasing the throughput at the cost of lower duplicate data found for some datasets.

The algorithm works by searching a byte with the maximum value in the fixed-sized window. If the byte next to the fixed-sized window has a larger value than the one in the fixed-sized window, the byte is used as the maximum-valued byte and the cut-point is found. Otherwise, the algorithm moves to the next byte until it finds the larger byte as illustrated in the pseudo-code of RAM in Algorithm 2. Thus the algorithm's minimum chunk size is $w + 1$, where w is the size of the fixed-sized window.

In the first condition for AE, AE compares the current byte with the maximum value and proceeds with the next comparison when the current byte is lower than the maximum value as shown in Algorithm 1. Assuming that all bytes have equal chances of having a value from 0 to 255, the probability that the currently observed

byte is smaller than the current maximum value is high. In such cases, the probability of performing comparison for the inner condition is high and the number of comparisons would be closer to $3L + 1$.

As shown in Algorithm 2, RAM reduces the number of comparisons by entering the outer condition and performing the inner condition only if the current byte is larger than the maximum value. Assuming that each byte value has an equal probability and the current maximum is large, the probability of performing the inner condition would be low and closer to $2L + 1$. Additionally, RAM only assigns variables when entering the outer condition, which can reduce the computation time.

As a drawback, this change in the condition affects the characteristics of RAM in producing long chunks. In RAM, the new condition results in an inability of limiting the maximum chunk size because once the maximum value is set to a high value such as 255, the currently observed byte value must be 255 for a cut-point, which is less likely to occur assuming equal probability all byte values (0 255) to appear. This issue is even worse for low-entropy strings which have long repetition bits such as a repetition of 0 after a 255-valued byte. We solve this by adding an extra condition in the second if for the chunk's maximum length, increasing the number of comparisons to $4L + 1$ for the worst-case scenario. However, because RAM rarely performs comparison in the branched conditions and compiler optimizations, the average number of comparisons is similar to $3L + 1$ in AE but with fewer variable assignments.

2.3.1 The properties of RAM

In this section, we address the properties of RAM subject to the criteria of an ideal CDC algorithm described in Section 2.2.2.

Content dependent. RAM chunks file based on the internal feature of the file, specifically based on the extreme values within the file. When it finds a byte larger than all of the bytes in the windows, it uses the byte position as a cut-point. Figure 2.2 shows an example of RAM's byte shift-resistant. First, we assume the change is smaller than the maximum value D. If there is a byte insertion or change at the fixed window [A1, B1], Chunk 1will be realigned and Chunk 2 will not be affected by the change as long as the maximum in [A1, B1] is not out of this window after the insertion. It also will not change the cut-point if a byte is inserted in the variable-sized window [B1, C1] because Chunk 1 will realign, and Chunk 2 will not be affected. Second, we assume that the change is larger than the maximum value. If the change occurred in [A1, B1], the position of C1 will be changed until it finds a byte larger than the change. Thus, this may affect the following chunks. If the

Table 2.1: Probability of having no cut-point in long intervals.

	AE	RAM	LMC	Rabin
M	$\frac{1}{\lfloor(e-1)M\rfloor!}$	$\frac{1}{2M!}$	$\frac{2^{2M}}{(2M)!}$	e^{-M}
2	0.166667	0.041667	0.666667	0.135335
3	0.008333	0.001389	0.088889	0.049787
4	0.001389	2.48E-05	0.006349	0.018316
5	2.48E-05	2.76E-07	0.000282	0.006738
6	2.76E-07	2.09E-09	8.55E-06	0.002479
7	2.51E-08	1.15E-11	1.88E-07	0.000912
8	1.61E-10	4.78E-14	3.13E-09	0.000335

change occurred in [B1, C1], then D1 will be realigned to the changed location and thus may affect Chunk 2 and Chunk 3. The realignment will stop if the maximum byte in the fixed window region of the next chunk is bigger and still in the fixed window region. With this example, we can see the algorithm can realign the cut position while minimizing the number of chunks affected by the change.

Low chunk sizes variance and the ability to eliminate low entropy strings. Chunk variance can affect the duplicate finding performance as shown by H. Khuern in their study [25] and one of the variables to the chunk variance is the probability of a long chunk. A higher the probability of long chunks can cause higher chunk variance. When a long chunk is modified, the amount of new data to be stored will be increased more than when a small chunk is modified. Therefore, we want to minimize the amount of chunk's size variance and the probability of long chunk. As analyzed in the article [17], the probabilility of long chunk of RAM is $\frac{1}{2M!}$, where M is the multiplier for the average chunk.

Low entropy string is a problem for RAM. When the low entropy string starts at the beginning of the fixed-sized window, RAM can eliminate the low entropy string because the condition for a cut-point is that the maximum-valued byte must be equal to or larger than the maximum in the fixed window. On the contrary, when there is a byte larger than any value in the low entropy string is in the fixed-sized window, the chunk size can become infinite because it cannot find a byte with larger value. To solve this, we can add a limitation on the maximum chunk size.

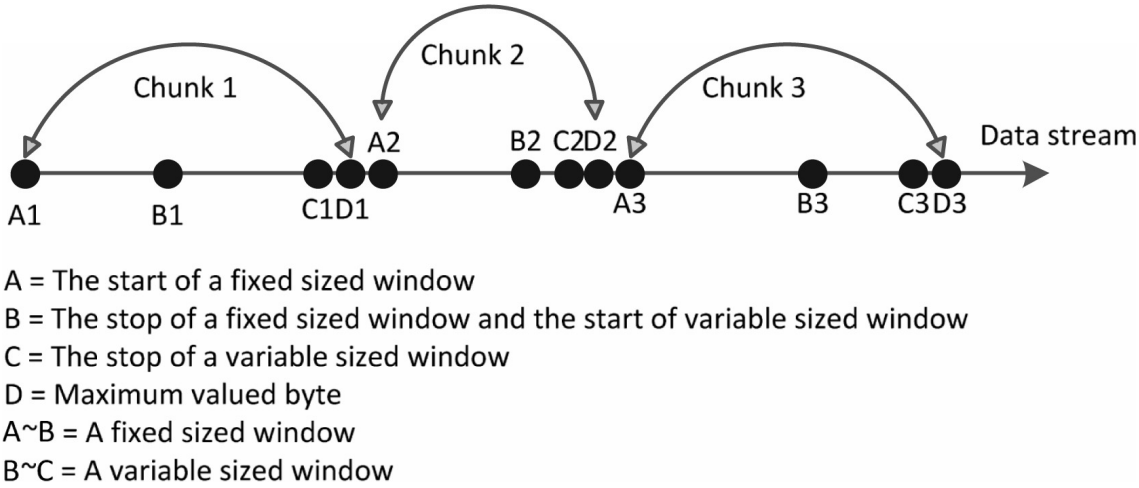


Figure 2.2: Byte shifting and byte change example.

High throughput. To prove that RAM has a low performance overhead, we use the worst case of RAM, based on the number of comparisons. RAM uses while loop which takes $L + 1$ comparisons and two additional conditional branches which add $2L$ comparisons, where L is the length of the input data stream in bytes. Thus in total, it uses only comparisons in the worst case scenario. Since the probability of finding a byte larger than the max is smaller than finding a smaller byte, on the average case it uses $2L + 1$ comparisons.

2.4 Performance evaluation

To evaluate the proposed algorithm, we implemented all algorithms reviewed in this work in C++ and compared them in a series of chunking tests with different dataset types. We also implemented RAM with the chunk size limit and called it RAML in the evaluation. We compared the throughput and the byte saved per second (BSPS) metric, which is also used in the study of AE [14]. BSPS is a metric that represents both the number of duplicates found and the throughput of the algorithms. The throughput is calculated as:

$$Throughput = \frac{datasetsize(bytes)}{timeforchunking(s)}$$

and BSPS is calculated as:

$$BSPS = \frac{duplicatefoundsize(bytes)}{originaldatasetsize(bytes)}$$

Table 2.2: The datasets used in the tests.

Dataset	Content	Size
Dataset 1: Operating system installation image	Arch Linux 2016.08.01 dual [27] Chromium OS (2016.08.09) [28] Debian 8.5.0 [29] ElementaryOS 0.3.2 [30] Linux Mint 18 Cinnamon 64 bit [31] Lubuntu 14.04.5 Desktop [32] Solus 1.2 [33] Ubuntu16.04.1 [34]	7.6 GB
Dataset 2: Media files	10 × 23 min H.264 videos	5.7 GB
Dataset 3: Network traffic	Data Capture from National Security Agency (NSA) CDX 2009 [26]	9.7 GB

Table 2.3: The window configurations for the tested algorithms used in the tests.

Dataset	AE	RAM	RAML	LMC	Rabin
Dataset 1	770	764	764	198	(through pattern length)
Dataset 2	770	770	770	203	(through pattern length)
Dataset 3	770	770	770	197	(through pattern length)

We ran all algorithms on the same test system with Intel i7 6700, 16-GB of DDR4 memory at 2133 MHz, and 120-GB SSD.

The performance comparison consists of three datasets. The datasets used in the tests are chosen to represent the use cases of the chunking algorithm. The first dataset is the compilation of multiple Linux distributions which have a lot of duplicate data in different locations in each file. The second dataset consists of 10 H.264 encoded videos of length 23 min each to simulate deduplication of media files in cloud storage. Lastly, the third dataset contains TCP dump files from NSA [26] to represent deduplication network traffic. The dumps contain 15 GB of data. However, we only used 9 GB of the data because of the limitation of our test system. The chunks' metadata consumes a lot of memory and causes the program to stop working when the total number of chunks went over 10 million chunks. We did not optimize the chunks management because our focus in this performance evaluation is chunking performance. Table 2.2 summarizes the content of the three datasets.

To make a fair comparison between the chunking algorithms, we configure each chunking algorithm with the same average chunk size. The average chunk size has a direct relation to the chunking performance. For example for network traffic, we need to make the average chunk size similar to the size of the average TCP packet

because if the chunk size is too large, the algorithm might miss duplicate data. On the other hand, if the chunk size is too small, the metadata of the chunks will be bigger than the size of the duplicate data.

We used SHA1, a bloom filter, and a hash table for hashing and comparing chunks. A SHA1 hash value will be calculated for each chunk. The bloom filter and the hash table are used for the duplicate finding process. Each hash value will be checked by using the bloom filter. Because of the nature of the bloom filter, false positives are a possibility. Therefore, when the bloom filter says that the hash existed, the hash is also checked in the hash table. If the hash is not in the hash table, it means the chunk corresponding to the hash is not a duplicate chunk and the hash is added to the hash table. If the bloom filter or hash table says the hash is new or not available in memory, then the hash will be added to the bloom filter and the hash table. By using the bloom filter, we can speed up the lookup process by not checking the hash table when the bloom filter says the hash is new because the bloom filter's lookup time is shorter than a hash table lookup. In this paper, we focused on the chunking algorithm performance. Thus, this paper does not include the chunk management and storage processes. The data deduplication process in the performance evaluation is only done until the duplicate finding process.

For AE, RAM, and LMC, we configured the windows' size to get the same average chunk size. Since it is harder for Rabin-based chunking to adjust the average chunk size, we ran Rabin chunking algorithm first to get the average chunk size. Then, we adjust the other three algorithms to match the average chunk size. From our experiment, changing the datasets does affect the average chunk size. This is due to the fact that some datasets may contain low entropy strings. Table 2.3 shows our configuration for the algorithms in our test. Additionally, we set the maximum chunk size limitation for RAML to 4 times the window size. As for Rabin, there is no limitation to the minimum and maximum chunk sizes.

2.4.1 Chunks properties

In this section, we compare the chunking algorithms based on the chunk properties. The chunks produced by each chunking algorithm are analyzed based on the chunk size distribution and the chunk variance. The chunk size distribution will be displayed in a histogram while the chunk variance is calculated using $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})$. As discussed in Section 2.2.2, we want the chunk variance to be as low as possible.

Dataset 1: Linux distribution

Dataset 1 contains Linux distribution operating system installation images which have a lot of similar contents and low entropy strings. This dataset is used to show the performance of a chunking algorithm in finding duplicates between files that have similar contents. The results for this dataset are compiled in Table 2.4 Figure 2.3.

There are three interesting things that we can observe: chunk variance, chunk size distribution, and the duplicates found by the algorithms. The chunk variance data shows that AE has the lowest chunk variance compared to others because of the ability of AE to eliminate low entropy strings. As for RAM, the chunk variance is worse than other algorithms. This is caused by the lack of the maximum chunk size of RAM which can affect the low entropy string elimination performance. The limit on RAML reduces RAM chunk variance and improves its low entropy string elimination. This can be seen in the 1.8% improvement in duplicates found for RAML over RAM. LMC and Rabin have high chunk variance because of high counts of long chunks.

The chunk size distribution of RAM is similar to RAML and AE. The chunks are mostly narrowly distributed near the mean as illustrated in Figure 2.3. We can also see that the chunk sizes for RAM and RAML are more concentrated at below the mean because of the higher probability of finding the cut points, while the chunk size distributions for LMC and Rabin are wider than others.

The duplicates found by the algorithms for dataset 1 are not much different. We observed a difference of only 2.1% between the maximum and minimum. All of the tested algorithms were able to find 17%–18% duplicate between files. The low entropy string elimination capability of AE and RAML does help to increase the duplicate detection.

Table 2.4: Chunks properties for each algorithm for dataset 1.

	# of chunks in range				
Range	**AE**	**RAM**	**RAML**	**LMC**	**Rabin**
[0, 1000]	4,278,622	4,643,773	4,595,328	4,865,576	4,606,661
[1000, 2000]	3,101,223	2,660,591	2,718,236	1,645,625	1,733,733
[2000, 3000]	13,790	85,032	89,629	550,893	655,020
[3000, 4000]	121	7203	16,046	198,763	246,085
[4000, 5000]	19	2008	0	77,425	93,209
[5000, 6000]	2	829	0	33,040	35,183
[6000, 7000]	1	396	0	16,094	13,376
[7000, 8000]	0	235	0	8188	5141
[8000, 9000]	0	130	0	5000	2170
[9000, 10000]	0	115	0	2979	823
[10000, 11000]	0	0	0	4	1
Total number of chunks	7,393,778	7,400,312	7,419,239	7,403,587	7,391,402
Mean (bytes)	1030	1030	1020	1025	1030
Variance	48,700	15,400,000	88,300	12,200,000	10,800,000
Duplicates found (MB)	1,024	1,006	1,025	1,004	1,011

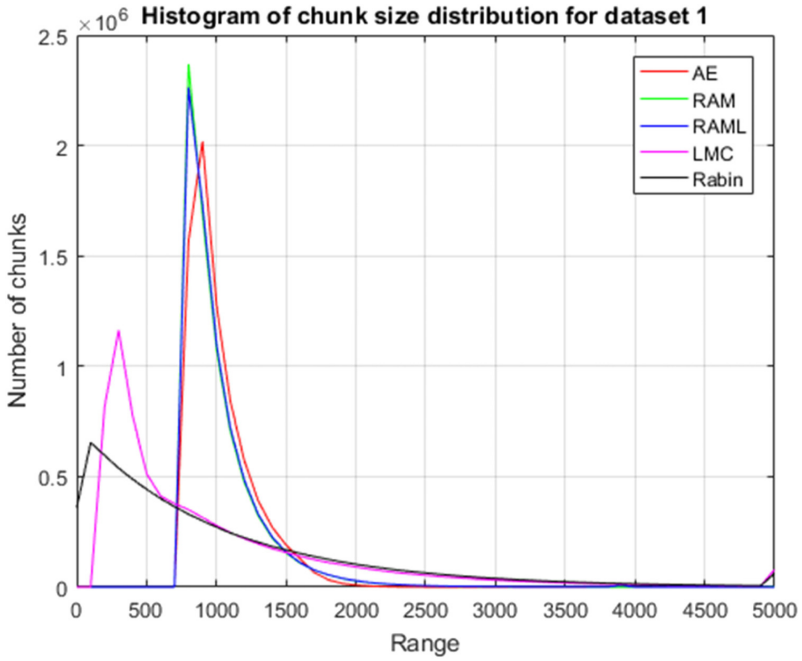


Figure 2.3: Histogram of chunk size distribution properties for dataset 1.

Dataset 2: media files

With the compressed H.264 video files in dataset 2, we saw different results. Dataset 2 contents are compressed video files encoded in H.264. Therefore, the number of low entropy strings is lower in this dataset compared to the other two datasets. The results are compiled in Table 2.5 and chunk size distribution is illustrated in a histogram in Figure 2.4.

In this dataset, RAM has a lower variance compared to dataset 1 because the dataset contains fewer low entropy strings. Another interesting point in this dataset is RAML's result. The limit on RAML affects the content-defined property of RAM and reduces the number of duplicates found in this dataset. Thus, adding the limit to RAM is only encouraged for a dataset with a high amount of low entropy strings.

Similar to the previous dataset, the value of chunk size variances for LMC and Rabin are larger than AE, and RAML. On the contrary, we find that LMC which has the high chunk variance detected more duplicates compared to the other algorithms despite the larger chunk size variance. As we can see in Figure 2.4, the chunk size distribution of LMC peaked at 300 bytes, which is not close to the minimum chunk size of LMC which is 203 bytes. Although the chunk distribution of LMC is not better than AE, RAM, and RAML, LMC eliminated more duplicates than the other algorithms and netted 12.8% more duplicates found compared to Rabin.

Table 2.5: Chunks properties for each algorithm for dataset 2.

	# of chunks in range				
Range	**AE**	**RAM**	**RAML**	**LMC**	**Rabin**
[0, 1000]	3,117,878	3,403,698	3,403,746	3,495,518	3,448,464
[1000, 2000]	2,392,131	2,138,311	2,138,316	1,314,110	1,301,558
[2000, 3000]	4142	40,001	40,021	460,609	489,890
[3000, 4000]	1	951	1042	164,826	184,840
[4000, 5000]	0	73	0	59,608	69,392
[5000, 6000]	0	5	0	21,271	26,165
[6000, 7000]	0	0	0	7815	9970
[7000, 8000]	0	0	0	2759	3610
[8000, 9000]	0	11	0	1032	1416
[9000, 10000]	0	1	0	372	559
[10000, 11000]	0	0	0	0	0
Total number of chunks	5,514,152	5,583,051	5,583,125	5,527,920	5,535,864
Mean (bytes)	1029	1016	1016	1026	1025
Variance	43,466	62,075	61,930	921,860	1,049,000
Duplicates found (kB)	2,128	2,030	2,021	2,259	2,002

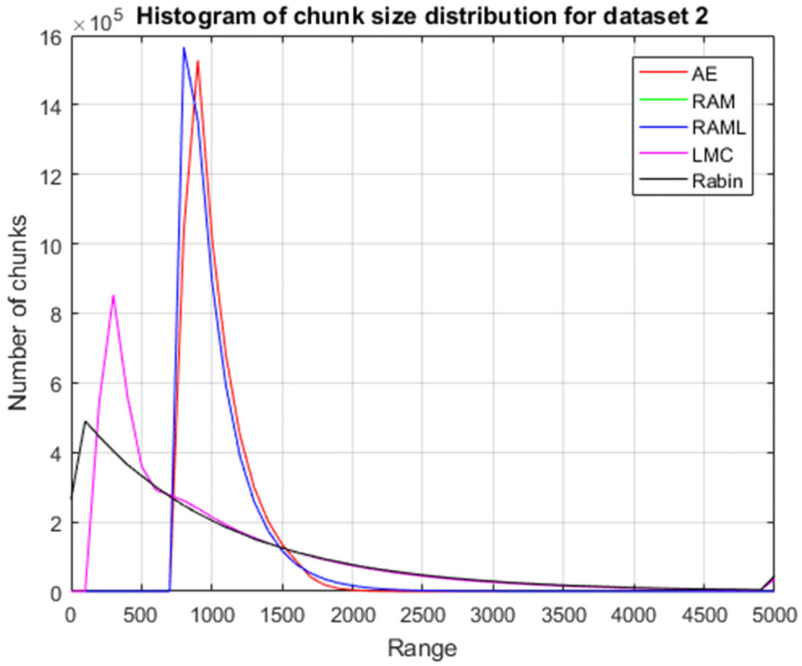


Figure 2.4: Histogram of chunk size distribution properties for dataset 2.

Dataset 3: network dump files

The test on dataset 3 represents the performance of a chunking algorithm when processing network traffic data. The content of dataset 3 is not compressed and it is possible to find low entropy strings in this dataset. The size of the packet is not defined. However, based on our knowledge, the default MTU size for most machines with traditional Ethernet is 1500 bytes. Similar to the other two datasets, we compiled the results in Table 2.6 for chunk size information, and Figure 2.5 for the histogram of the chunk size distribution.

The chunk size variance for dataset 3 shows a result similar to that using dataset 2. RAM is still behind AE and ahead of LMC and Rabin in terms of chunk size variance. Although not significant, adding a limit to RAML improves the number of duplicates found by RAM. The main cause of the improvement due to the limit is the existence of a low entropy string in the dataset.

In terms of duplicates found, the Rabin chunking algorithm found more duplicate data compared to other algorithms in dataset 3, while RAM is behind AE in terms of duplicates found. We suspect that no limit on the Rabin chunking algorithm helps it to detect duplicates on the network traffic because of the packet size randomness. The chunk size distribution in Figure 2.5 shows all algorithms produced similar chunk size distributions to the previous datasets.

Overall, RAM performance is between AE, LMC, and Rabin in terms of chunk qualities. Additionally, we found that, depending on the dataset, some chunking algorithms are better than others. For example, LMC is better for compressed files and Rabin is more suitable for network traffic files. This fact is similar to what has been reported by Meister et al. [35].

Table 2.6: Chunks properties for each algorithm for dataset 3.

	# of chunks in range				
Range	**AE**	**RAM**	**RAML**	**LMC**	**Rabin**
[0, 1000]	5,435,437	5,768,110	5,783,201	6,280,039	5,752,016
[1000, 2000]	3,961,755	3,438,323	3,467,617	2,145,609	2,170,580
[2000, 3000]	25,264	120,281	123,282	652,904	836,368
[3000, 4000]	177	15,750	35,747	238,053	327,584
[4000, 5000]	13	5479	0	99,870	127,361
[5000, 6000]	1	3155	0	42,968	50,628
[6000, 7000]	0	1795	0	23,496	20,390
[7000, 8000]	0	971	0	13,456	8399
[8000, 9000]	0	793	0	7391	3931
[9000, 10000]	0	896	0	5058	1973
[10000, 11000]	0	2	0	8	1
Total number of chunks	9,422,647	9,355,555	9,409,847	9,508,852	9,299,231
Mean (bytes)	1,034	1041	1036	1023	1048
Variance	52,992	306,650	104,550	5,386,300	1,185,100
Duplicates found (kB)	329	277	277	257	340

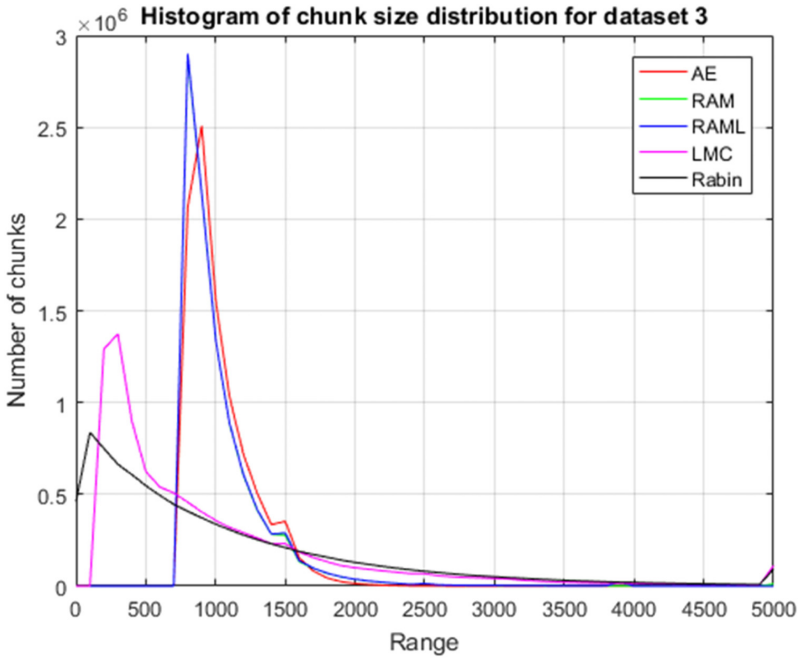


Figure 2.5: Histogram of chunk size distribution properties for dataset 3.

2.4.2 Throughput

This section compares CDC algorithms based on the chunking throughput and Bytes Saved per Second (BSPS). The purpose of this section is to find the algorithm that has the best balance between chunking throughput and the number of duplicates found. The chunking throughput is calculated by dividing the amount of data processed and the amount of time consumed for chunking the files. BSPS was first used by Yinjin et al. [36] as a performance metric for deduplication systems. The BSPS is calculated by dividing the number of duplicates found by the number of files processed and multiplying the result with the throughput.

$$Throughput = \frac{datasetsize(bytes)}{timeforchunking(s)}$$

$$BSPS = \frac{duplicatefoundsize(bytes)}{originaldatasetsize(bytes)}$$

The results for dataset 1, dataset 2, and dataset 3 for the tested chunking algorithms considered in this paper are compiled in Table 2.7, the chunking throughput for the tested algorithms is illustrated in Figure 2.6, and the byte saved per second is shown in Figure 2.7. The chunking time excludes the read time from the drive because our main focus is the chunking performance.

The results shown in Table 2.7 indicates that the throughput of the chunking algorithms in our test system is as follows: RAM up to 561 MBps, RAML up to 448 MBps, AE up to 386 MBps, LMC up to 89 MBps, and Rabin up to 105 MBps. This throughput result indicates that the new condition structure in RAM can improve the throughput of AE by around 45%. As for the duplicate found, RAM detects 2% to 18% fewer duplicates than AE depending on the dataset. In Dataset 3, RAM performed 18% less than AE because the dataset contains many low-entropy strings. However, in terms of BSPS, RAM is around 40% to 30% better than AE.

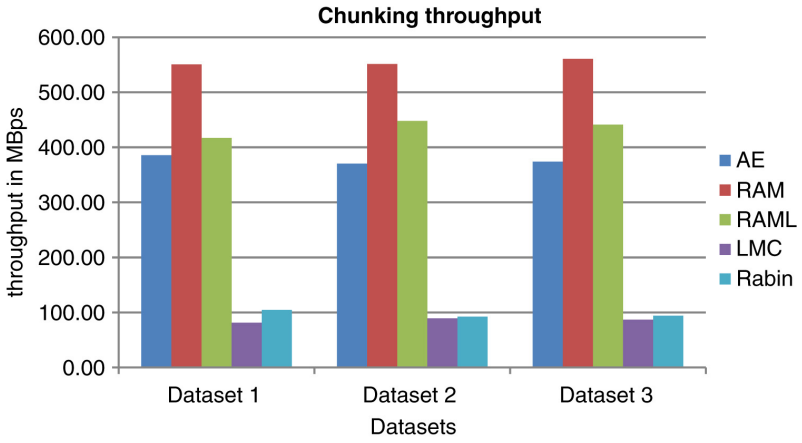


Figure 2.6: Chunking throughput for the tested algorithms.

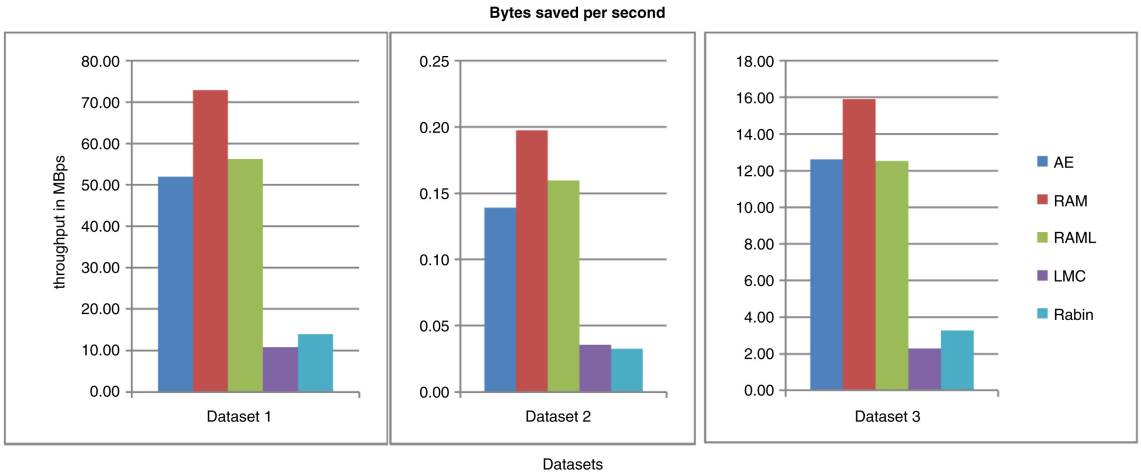


Figure 2.7: Byte saved per second for the tested algorithms.

Table 2.7: Duplicate found, chunking throughput, and byte saved per second results for the tested algorithms.

	AE	**RAM**	**RAML**	**LMC**	**Rabin**
Dataset 1 duplicates found (MB)	1024	1006	1025	1004	1011
Dataset 2 duplicates found (MB)	2128	2031	2021	2259	2003
Dataset 3 duplicates found (MB)	329	277	277	257	340
Throughput dataset 1 (MBps)	385.8	550.8	417.2	81.2	104.6
Throughput dataset 2 (MBps)	370.4	551.3	448.1	89.2	92.2
Throughput dataset 3 (MBps)	374.2	560.9	441.4	86.9	94.2
BSPS dataset 1 (MBps)	52	72.9	56.2	10.7	13.9
BSPS dataset 2 (MBps)	0.14	0.2	0.16	0.04	0.03
BSPS dataset 3 (MBps)	12.6	15.9	12.5	2.3	3.3

2.4.3 Discussion

As a chunking algorithm, RAM performs well compared to other chunking algorithms in terms of produced chunk characteristics. The chunk size distribution produced by RAM is narrower than the other tested algorithms. Although RAM has a higher chunk size variance because of the lack of maximum chunk size, the chunks have similar chunk distributions to the chunks produced by AE. The loss on higher chunk size variance makes RAM perform better in terms of chunking throughputs. Generally, RAM improved the performance of AE on bytes saved per second.

In Section 2.4.1, we found that lower chunk size variance does not mean that the algorithms will perform worse than the algorithm with lower chunk size variance, which is shown in Section 2.4.1 where RAML with worse chunk size variance compared to AE detected more duplicates, in Section 2.4.1 where LMC outperform AE in terms of duplicate detected, and in Section 2.4.1 where AE once again outperformed by other algorithms with worse chunk variance. This means the content-dependent capability of a CDC algorithm is more important than chunk size variance. This is proven by fixed-sized chunking. Fixed-sized chunking has zero chunk size variance but it is not content dependent which makes fixed-sized chunking performs worse in terms of duplicate detection compared to CDC algorithms. Another proof that chunk variance does not significantly affect the duplicate detection is our test in Section 4 for RAML, which is another version of RAM with a limit on the maximum chunk size. For dataset 2, adding limits can reduce the chunk variance of RAM. However, it also decreased the number of duplicates found because the limit makes it less content-defined. Therefore, chunk variance is an appropriate metric to compare CDC algorithms when the algorithms have similar content-defined characteristics.

The results in Section 4.3 indicate that RAM is preferable for low performance devices such as mobile devices and IoT, or applications where computing time and storage is important. Additionally, RAM is also useful for client side data deduplication or remote differential compression because with lower computational overhead, RAM can reduce the stress on client hardware.

2.5 Conclusion of this chapter

In this work, we discussed the importance of content-defined chunking for multiple applications and why it is better than fix-sized chunking. We proposed a new chunking algorithm, called Rapid Asymmetric Maximum (RAM) based on an asymmetric chunking algorithm. We analyzed and compared RAM with other chunking algorithms. Our results show that RAM offers a lower computational overhead compared

to other CDC algorithms.

The main advantage of RAM is its low computation overhead which allows high chunking throughput. The high chunking throughput comes at the cost of higher chunk variance. The higher chunk variance produced by RAM is negligible compared to the performance gain over other chunking schemes based on local maximum chunking. In some cases, RAM offers 26%–40% higher byte saved per second compared to the other chunking algorithms.

In the future, we will address the high chunk variance of RAM to improve the duplicate finding performance. We will also study how chunk size variance affects the duplicate finding performance.

Chapter 3

Data reduction schemes enabled DFS design

3.1 Introduction

DFS are storage systems that utilize multiple storage nodes in a cluster to be highly scalable. It can expand its storage capacity by adding more nodes to the cluster or increasing the storage capacity of each node through extra hardware. A DFS commonly has two main components, the metadata server and the DFS software that runs on each storage node. The metadata server manages all storage nodes that run the DFS software, which has access to the disks in each storage node. The DFS software usually manages the storage disks through a filesystem. Figure 1.1b illustrates a common DFS design. To further improve space utilization, users can use data reduction schemes such as deduplication and compression in their DFS applications.

Data reduction schemes can improve the storage space efficiency of any storage solutions including DFS at the cost of computation as shown in existing studies [9, 10,37]. Based on the structure of the DFS software in an application, data reduction schemes may run in three different layers as shown in 1.1a. The existing solutions have been using the application [10, 37] or the underlying FS [9] for activating the schemes. These approaches have their characteristics in terms of the ease of adding or enabling the data reduction schemes.

Using the appropriate data reduction schemes is crucial for achieving the best storage space efficiency because data may have different structures. For example, Logzip can reduce more storage space than Gzip for log data type [8]. Additionally, these schemes may have a different trade-off between processing time and space reduction. Users might want to use the best scheme for their use case. Adding new

schemes to improve the existing selections is available through different solutions. For the application layer, Hadoop, a big data platform, provides CompressionCodecs for enabling schemes in the Hadoop applications, which can store the data in HDFS. Through this approach, users can easily add new schemes by extending the CompressionCodecs class. For the FS layer, currently, no such approach exists for the user to use. Users must modify the existing FS code if they want to extend the selections, which can be challenging depending on the code's complexity.

Enabling the schemes is a different issue from adding the scheme. For example, successfully adding new CompressionCodecs does not always translate to success in using them in Hadoop applications. In Hadoop, when the application does not support CompressionCodecs, the schemes won't run on the data and HDFS will store the data without any changes from the application. Users can add support for the CompressionCodecs to the existing application. Adding this support to the application code requires the availability of the application code, which is not always accessible, and the understanding of the application code. At the FS layer, enabling the schemes for all applications is significantly easier assuming that the FS already supports the schemes. Users can enable them by making some changes in the FS configuration without any changes in the application because the process in the FS is transparent to the application.

The DFS layer, where the DFS software runs on the storage nodes, can enable the data reduction schemes like the FS because its process is transparent from the applications. However, adding new schemes can be challenging because of the complexity of the DFS software. As far as the author's knowledge, currently there are no DFS that can directly perform these schemes directly in the DFS software that runs in the storage nodes.

In this work, we proposed a new DFS design that can enable data reduction schemes in the DFS software. The proposed design uses the same approach as CompressionCodecs in the application layer for the ease of adding new schemes and has the same process transparency as the FS. In our experiments, the implementation of the proposed design called Hadoop Data Reduction Framework (HDRF) [38], which operates inside HDFS, can enable data reduction schemes in applications that cannot run CompressionCodecs without any extra effort from the application side. The results also indicate that the overhead is negligible.

3.2 Background and related work

This section discusses the challenges of existing DFS designs for data reduction schemes and our motivation to solve the challenges.

3.2.1 Data reduction schemes

Data reduction schemes like compaction, compression, and deduplication can solve the data growth, which is a challenge for most storage systems, especially those that handle large data like cloud storage systems [39, 40, 41, 42], virtual machine images storage systems [12, 41, 43, 44, 45], and big data platforms [37, 46, 47, 48, 49]. These schemes work by reorganizing the data more efficiently to minimize potential redundancies at the cost of computation, and thus minimizing the storage footprint [49]. In several cases where the disk is significantly slower than the processing unit, the reduction in I/O operations can improve the system performance from storing or reading fewer data. In this work, we focus on lossless compression and deduplication.

Lossless compression is a reversible process that removes redundancy within a file or data block. This type of scheme can be found in many applications, including memory compression [50, 51, 52] and file compression (e.g. Snappy [53], Lz4, Bzip2, and Gzip). For file compression, most lossless compression algorithms are based on a dictionary coder algorithm.

A dictionary coder operates by finding redundancies within a file. Let file f be a set of n blocks b, $f = \{b_1, b_2, ..., b_n\}$, and each b is composed of m words w, $b = \{w_1, w_2, ..., w_m\}$. The dictionary coder uses a data structure called a dictionary d, which is a subset of b, that maintains i unique w, $d \subseteq b$. When it encounters a redundant w, the dictionary coder replaces it with a pointer p to the corresponding unique w in the dictionary. In summary, the dictionary coder produces a compressed file f' with a total size of $f' = \sum b'$, where $b' = d + \sum_{x=1}^{m-i} p_x$ and $b' \ll b$ for best-case scenarios. However, when $m - i$ is close to 0, b' can be larger than b because of the overhead of d.

Deduplication is also a lossless process that maps files into smaller files called chunks with chunking algorithms [14, 54] and stores unique chunks in containers. Deduplication finds duplicate chunks by comparing the chunks' fingerprints, which are obtainable through mathematical hash functions like SHA1. Storage-saving is achieved by replacing duplicate chunks with pointers to the existing chunks. To reconstruct the original file back, deduplication systems use the file's recipe, which lists the chunks that correspond to the file. In comparison to compression, deduplication can remove redundancies on a larger scale such as those among files and storage devices. However, deduplication is usually more memory-intensive because it needs to compare a larger number of chunks. Deduplication works well for datasets that share similar parts like virtual machine images [55] and large storage systems [56].

Deduplication is similar to a dictionary coder, but it works with a set of files, $\{f_1, f_2, ..., f_m\}$ and $f = \{b_1, b_2, ..., b_n\}$, and at a more coarse granularity. It exploits the possibility of duplicate blocks b, which are also called chunks in deduplication,

within f and among files, $f_a \cap f_b \cap f_c = duplicate\ b$. Because comparing blocks with bit-by-bit comparison is slow when considering the size of b, deduplication uses a mathematical hash function to produce a hash h_b or the fingerprint of each chunk and uses it in the comparison process. If multiple b have a matching hash value, then only a single instance of b and its hash h_b is stored in a data structure or database DB, $DB = \sum(b + h_b).\ p$. Finally, it replaces the file with a recipe r that contains a list of hashes h_b of the chunks in the original file, $f = \{b_1, b_2, ..., b_n\} \rightarrow r = \{h_{b_1}, h_{b_2}, ..., h_{b_n}\}$. In the best-case scenario, the total size, which is $sumr + DB$, is significantly smaller than $sumf$. However, in a number of cases where the number of redundant b is too small, the space overhead of the DB and r might be larger than the total size of redundant b.

Several studies proposed the use of data deduplication in a DFS through the application layer [10, 57, 58]. These approaches are more beneficial for data size reduction because data deduplication works best with a large dataset. However, data read processing through the DFS is a challenge because the deduplicated data is not the same as the original data. Directly processing the deduplicated data with a platform-provided data access method like Hadoop MapReduce and Apache Spark may result in an inaccurate output. In such cases, the application may not be able to leverage the distributed computing capability of the DFS and it requires extra processing to revert the deduplication process to produce an accurate result.

3.2.2 Distributed File System (DFS)

A DFS is a cluster of storage nodes that is scalable both vertically and horizontally. Each storage node can be expanded vertically by attaching more storage devices. Once the limit of vertical scaling is reached, the DFS can still grow through horizontal scaling or by adding more storage nodes to the cluster. This approach enables the DFS to keep up with data growth and increase storage I/O performance through parallelization.

Although DFSs are scalable in every direction, data growth is still a problem for all storage devices. To resolve this problem, DFSs can be combined with data reduction techniques like compression and deduplication. Because the DFS is software or middleware that connects the application to another FS, it can be split into three layers on the basis of the location of the data reduction, which are the application, DFS, and FS layers, as shown in Figure 1.1a. At the application layer [37, 59, 60, 61], the application must support data reduction schemes to benefit from more efficient space utilization. At the FS layer [9, 62], this is no longer an issue because the data processing of the FS operates separately from the applications in their layer.

3.2.3 Hadoop and Hadoop Distributed File System

Hadoop [63] is a commonly used big data platform that uses the Hadoop DFS (HDFS) [11] to store datasets in blocks. HDFS has two nodes: the name node, which handles the files' metadata and blocks' location, and the data node, which stores fixed-sized blocks. HDFS saves these blocks through another underlying FS like EXT4, NTFS, or ZFS. To ensure the blocks' reliability and availability, it uses a block replication scheme, which puts several copies of the same block in different data nodes to prevent losing blocks during data node failures. This replication process occurs when a node receives a packet from a client or another node.

HDFS, unlike other DFSs like Lustre [64], supports the MapReduce programming paradigm to parallelize data processing in the cluster, thus enabling data nodes to perform data processing or compute. The MapReduce code can also be combined with Hadoop CompressionCodecs to optimize the storage usage in HDFS. The supported codecs are Gzip, Bzip2, Lz4, Snappy, and Zlib. Adding new reduction schemes as a codec is also possible. However, these codes must be included in the application code to enable them. A simpler approach would be to use an FS like ZFS to enable compression or deduplication in HDFS or other DFSs, which do not need any modification in the application code.

Another concern of using the codecs in combination with MapReduce is splitability, which is the possibility of independently decompressing each block. It is crucial because MapReduce may operate parallelly on each block. However, several codecs produce non-splittable blocks, which may require other blocks to decompress. In such cases, the MapReduce application cannot run in parallel and the data node may need to retrieve blocks that it does not have from other nodes. The FS-based approach has no such issue because each HDFS block is compressed independently from the DFS layer.

3.2.4 Related work on data reduction schemes in DFS and HDFS

At the application layer, the users can add a scheme into the application code or enable it through a platform-specific API. For example, client-side an achieve a compression ratio of 1.5 in Lustre with Lz4 [65] and users of Hadoop can use Hadoop CompressionCodecs to enable schemes in their MapReduce [66] applications or directly add the scheme code to the application when using Hadoop [10, 37, 57, 58, 59, 60, 61] or Lustre [64]. Applying the scheme at this layer also provides benefits for a reduction in network traffic at other layers, thus speeding up the data transfer in the lower layers through the network. Additionally, the receiving node does not

need to recompress the data independently when replication is set to above 1-time for HDFS as shown by Widodo et al. [38]. However, application code modification, which may not be possible if the code is not available, is mandatory to enable the scheme.

An important aspect of the data reduction scheme in the application layer is the split-ability of the data reduction schemes. HDFS splits files that are larger than the HDFS block into multiple block files. These split block files may not be splittable and processed independently when compressed with non-splittable algorithms such as Snappy, limiting the number of mappers for MapReduce applications. A solution is to use splittable algorithms such as Bzip2. Another solution is to use sharding in the applications that generate the data such that the compression output is tied to HDFS's block size.

Several studies [10, 57, 58] have proposed data deduplication applications at the application layer that store the data in HDFS. Ranjitha et al. [57] and Sun et al. [58] studied deduplication applications that manage the output data in HDFS. Zhang et al. [10] explored the possibility of deduplication through the MapReduce programming model that passes the data to HDFS. The main drawback of these studies is that the deduplication cannot be independently reversed by each data node. These applications must revert the deduplicated data to the original data and store it back in HDFS or other storage solutions before processing it to ensure consistent results, increasing the cost of data processing for Hadoop clusters.

Another common approach to enable data reduction schemes in a DFS is through the FS layer [9, 62]. For example, Zhou et al. [9] used ZFS as the FS for HDFS and enabled deduplication and compression through ZFS configurations. This setup provides a transparent data reduction to the application and DFS layers. However, there is no reduction in network traffic because once the data leave the ZFS, the data is back to its original structure and size.

Data reduction schemes at the DFS layer can solve these challenges when the DFS is designed to run with the schemes. However, enabling them at the DFS layer has yet to be done because of several possible reasons. First, it adds complexity to the DFS, which increases the cost of development and the chance of breaking other components in the DFS. Second, based on our experience, directly modifying the DFS source code to enable the schemes can be complex and time-consuming. Because of these reasons, it is difficult to compare reduction schemes at the DFS layer to the other approaches based on the benefits and development costs.

In this paper, we proposed and tested a new DFS design that eases the data reduction schemes implementation and usage in a DFS. Users can use and enable the schemes through programming techniques like dynamic library linking or depen-

dency injection. The DFS design is also transparent to all applications, enabling data reduction schemes to run on all applications' data. With this DFS design, users can add and enable these schemes at the DFS layer without modifying the DFS code.

3.3 Design and implementation

This section provides the details of the proposed DFS design and the implementation of the design in HDFS.

3.3.1 The DFS design

Implementing reduction schemes in the existing DFS design requires careful handling in several aspects. The first is the ease of use from the user's perspective. If the use of the scheme is challenging, the users might not want to use it. Second, the placement of these schemes in the DFS may affect DFS functions like the metadata system and data integrity checker. Third, the DFS may not benefit from the network traffic reduction because of its smaller data size.

To overcome these challenges, we proposed a DFS design that supports data reduction schemes and extends existing DFS designs as shown in Figure 3.1. The key features of the design are: (1) it uses existing programming techniques to link the schemes' libraries to the DFS; (2) it runs the scheme close to the host FS to minimize the changes to other DFS components; (3) it can provide reduced and original data streams to the DFS and schemes to avoid reprocessing within the DFS.

Ease of use of data reduction schemes. Adding and enabling data reduction schemes in the application and FS layers have their limitations. At the application layer, they depend on the capability of each application and its availability of the source code. For example, in Hadoop, the users can add new schemes through CompressionCodecs and enable them in the code or configurations when the application supports it. However, these can be difficult depending on the application's source code's availability. At the FS layer, enabling the data reduction schemes is not a problem because it is transparent to the applications. However, the adding part is like the application layer because it depends on the source code's availability. Even when available, modifying their code is difficult because its complexity is commonly higher than the applications' code.

The design of our DFS solves these problems through common programming techniques and by exploiting the nature of the DFS layer. The design uses the same

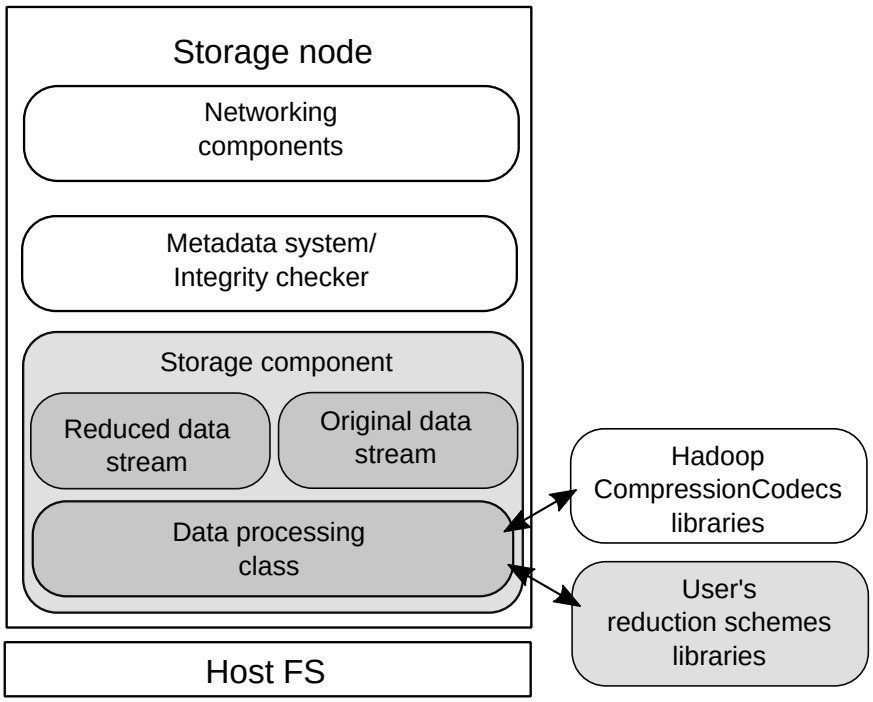


Figure 3.1: Proposed DFS design. The data reduction scheme is placed close to the host FS to minimize the changes in existing DFS designs. The grayed parts are the new additions to the existing DFS components.

approach as Hadoop CompressionCodecs, which connects to the application through independent Java jar files. Users can add new data reduction schemes to the DFS by extending a template data reduction class or by adding it through CompressionCodecs and extending the template class. The application can load these schemes through jar libraries. As for enabling these schemes, the design benefits from the nature of the DFS layer, which works transparently to the application layer like the FS layer.

Schemes placement in the DFS. The implementation of the schemes in the DFS may harm several DFS functionalities depending on the placement. For example, running the schemes before the metadata system can generate errors because of the metadata mismatch between the client's and DFS's generated metadata. The design prevents such issues by activating the schemes before storing the data in the underlying FS, similar to FS-based reduction schemes like ZFS. This design also minimizes the changes over the existing DFS design because once the data leaves the storage part, it is back to its original form.

Network traffic reduction. One of the benefits of using data reduction schemes

at the DFS layer is that it can also reduce network traffic, speeding up DFS operations that use the network. However, when these schemes work at the storage part of the system, there is no reduction in the network traffic because the networking part of the DFS usually operates above the storage components. Additionally, the DFS may also suffer from extra reprocessing during data replication. This reprocessing cost can be expensive for older nodes with low processing capability. To solve this design constrain, the proposed DFS design provides a selection of data streams, which contains the reduced and original data streams. Other DFS components can access both versions of the data outside of the storage portions of the DFS through these streams.

3.3.2 Implementation

As a proof of concept, we implemented the proposed design in HDFS, which is the default DFS for Hadoop, and named it HDRF. The implementation uses Hadoop 3.1.0 as the base and requires around 1000 lines of additional code and modification. Additionally, it works well with HDFS and MapReduce applications without any configuration or modification to the application code.

HDRF. As the name implies, HDRF is a framework that enables data reduction schemes in HDFS. HDRF operates within the DFS layer and works with all applications without any modification to their code. The data reduction schemes can be connected to HDRF through Java's dynamic library linking. This approach is similar to Hadoop CompressionCodecs, which works at the application layer, however, it works transparently with all applications. It records separate metadata for each block in Redis [67, 68] to ensure block integrity is not affected by the schemes. As shown in Figure 3.2, it requires some changes in HDFS's source code and has several features to minimize its processing overhead.

HDRF supports two types of streams: direct block array and processed data. The block array stream buffers the whole block data in the memory and can provide faster data access to the application by prefetching the block data in the memory. The processed data stream is similar to HDFS's file streams, which reads the data by chunks. Unless the user's data processing application requests the block array stream, HDRF uses the processed data stream because it is closer to the default code in the vanilla HDFS and minimizes the memory usage.

Local metadata system. HDRF maintains a local metadata system in each node. This system functions as the translator for the addresses of each HDFS block in HDRF. When HDFS checks the length of the HDFS block file, HDRF queries this system and returns the stored length. HDRF also uses this metadata system to

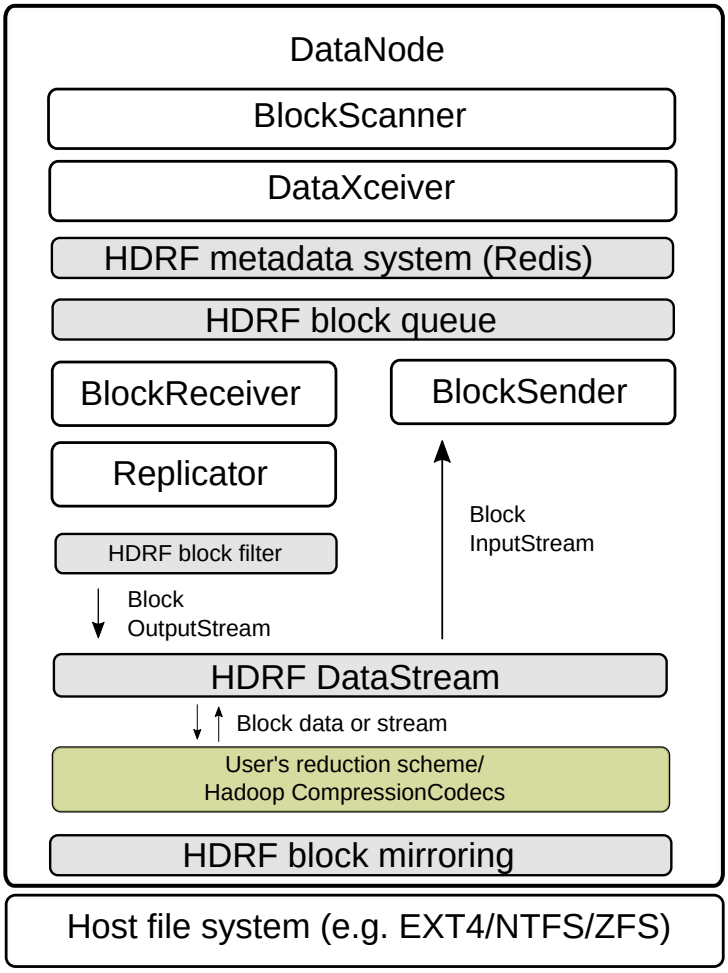


Figure 3.2: Proposed design on HDRF. The grayed parts are HDRF's components.

get the directory of the processed block from HDFS's blocks. The key for the HDFS block is its ID, and the value is the its length concatenated with the location of the processed block.

Data integrity changes in HDFS. HDRF works within the storage part of HDFS by replacing HDFS's storage stream with its own stream, resulting in empty block files. Without any changes to the data integrity checker of the data node in HDFS, this will cause errors because of the non-matching length between the block file, which is zero bytes, and the metadata. HDRF prevents these errors by replacing the length check by retrieving length information from its metadata system and passing it to the data node length checker. There is no change to the data integrity

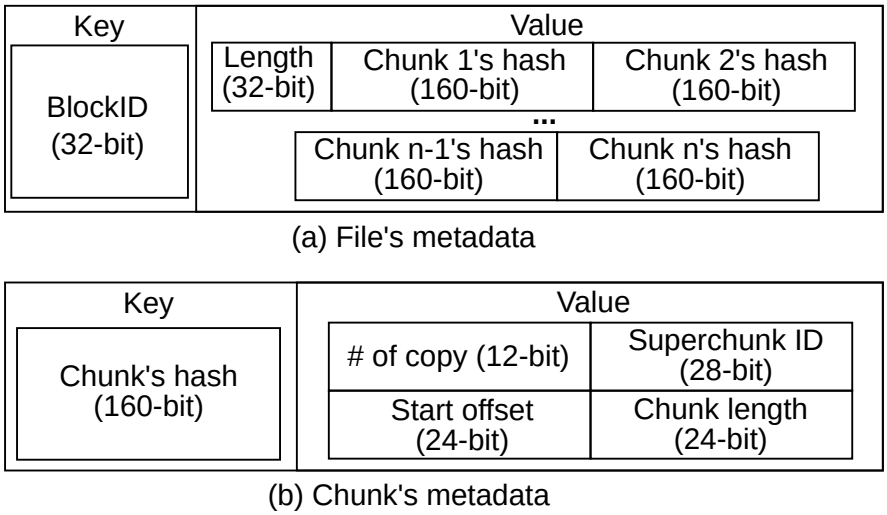


Figure 3.3: Key and value for: (a) file's metadata and (b) chunk's metadata.

check in the data node because HDFS performs it after HDRF produced the original blocks' data.

Deduplication through HDRF. To prove that users can implement their reduction scheme through HDRF, we created a simple deduplication scheme that splits the input data into small chunks, fingerprints the chunks with a hash function, matches the chunks with the hashes to find duplicates, and stores the chunks in a drive. The chunks metadata is stored in Redis. The scheme uses external libraries for chunking [54], hashing [69], and communicating with Redis [68]; the metadata database. The chunking algorithm is a content-dependent chunking (CDC) algorithm, which can automatically align the cut-point on the basis of the input data. However, it is still vulnerable to byte change, which may shift the cut-point or chunk boundary. In such cases, the affected chunk will be treated as a new chunk because it does not match the old chunk's hash. The deduplication scheme is connected to HDRF through the data reduction abstract class of HDRF. This scheme groups the chunks and store them in a large chunk called Superchunk to avoid random writes, which can be detrimental to most storage devices.

Figure 3.3 illustrates the structure of the key and value for the file's and chunk's metadata. This structure provides a maximum of 4 GB for block size, 256 MB for Superchunk size, and 16 MB for chunk length. "# of copy" is the number of duplicates for a chunk, and the maximum is 4096. With these tables, the deduplication scheme can rebuild the file with a small amount of overhead. In our experiment, these tables amount to less than 5% of the original data size.

3.3.3 Features of HDRF

HDRF has several features to make it easier for the user to implement their preferred data reduction schemes in HDFS and minimize its performance overhead.

Ease of adding new data reduction schemes. HDRF has two approaches for adding new data reduction schemes. First, through user configuration and dynamic linking, which is similar to Hadoop CompressionCodecs in the application layer. Second, by extending the data reduction abstract class of the HDRF. The former is similar to how Hadoop CompressionCodecs works in the application layer. The latter is more challenging because it needs to be compiled together with HDFS and HDRF. However, this solution is still easier than modifying the FS source code because the user can simply follow the abstract class to implement their new data reduction schemes.

Transparent data reduction. Application layer-side data reductions have several benefits over other approaches. For example, data size reduction is more effective at the application layer with schemes like deduplication, which works better with a large dataset. Additionally, adding new schemes is easier through a few extra lines in the code or through dynamic library linking, which is used by Hadoop CompressionCodecs. Although the data reduction library is only compiled once, the user must change the code for each application to enable data reduction. In this case, DFS- and FS-side data reduction is easier for the user because it works for all applications with the tradeoff of less effective data reduction. However, adding a new data reduction scheme to the FS and DFS can be challenging depending on the availability of the source code and its complexity. In this aspect, HDRF combines the benefits of the application layer- and the FS layer-side data reductions. Adding new data reduction to HDRF is possible through dynamic library linking and changes in the configuration like the application layer-side data reduction, and it works transparently for all applications like the FS layer.

Block mirroring. When the user applies data reduction schemes at the application layer, the lower layers like the DFS and FS layers can benefit from the reduced data size, thus minimizing data transfer time during DFS operations that use the network. For example, we can reduce the data transfer time in HDFS by applying compression when replication is in use as shown in Figure 3.11. However, such benefits may not exist when applying them at the lower layers because the scheme must reconstruct the data back to its original form to make the process transparent at the upper layers. This limitation increases the data transfer time when compared with application layer-based schemes. A solution to this is to send the reduced data instead of the original data, which is possible for the DFS layer.

To minimize the network traffic, HDRF has a block mirroring feature, which

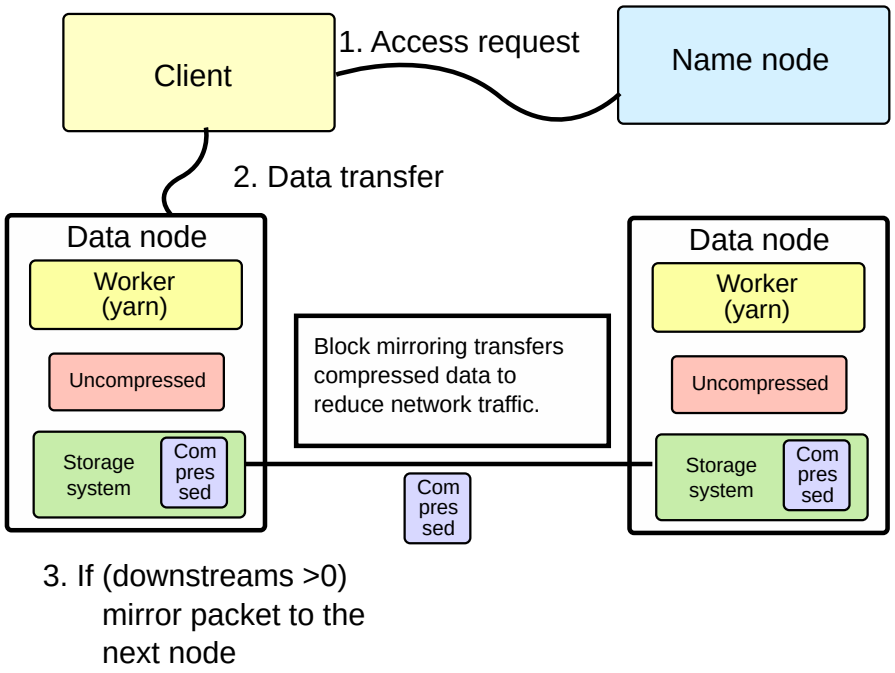


Figure 3.4: Block mirroring in HDRF. It checks the number of nodes downstream and sends the reduced data to the target nodes.

disables the block replication part of HDFS and sends the reduced block data to other nodes. It uses scheme stacking to read the data from the data reduction scheme and send it to another node through TCP communication at a latency like that of the vanilla HDFS and CompressionCodecs. Figure 3.4 illustrates how the block mirroring works in HDFS. During a block write, HDRF performs a check on the downstream number. If the number is more than 0, it will start block mirroring and contact the target nodes. Once the process is completed, the block is registered and accessible in the other nodes.

Seamless data access. A number of data reduction schemes can only process complete data, while others can process data block by block. To support these various methods of processing, HDRF has two block access modes: direct block array and direct stream access. In the block array mode, HDRF buffers the block data into an array and passes it into the scheme. HDRF can also revert the block array into a stream and pass it to HDFS for read-operations. However, block array mode requires higher memory consumption to buffer the block data. In the direct stream access mode, the HDRF directly passes HDFS's stream data to the reduction scheme to minimize memory consumption and possibly latency from buffering the data. Additionally, HDRF can provide either the original or processed block data

to the user's scheme for scheme stacking.

Scheme stacking. Additional processing of the blocks is often crucial to prepare them before or after they have been processed by the data reduction scheme. Although preprocessing can be done in the application layer through MapReduce code, this approach is inefficient because it must be enabled multiple times when applied to different applications. HDRF supports such processing through scheme stacking, which can connect a preprocessing scheme to a data reduction scheme or vice versa. This feature is possible because HDRF enables each scheme to request the original data or the scheme's processed data.

Block filter. Processing MapReduce job files can be wasteful for the node's resources because MapReduce job files are often short-lived and discarded after job completion. Additionally, several data types may not benefit from data reduction schemes. For example, data reduction schemes may have a low reduction ratio when processing encrypted data.

HDRF has a block filter that searches for user-defined keywords in the content of the first received packet for each block. These blocks are then stored without any additional processing, and thus minimizing the node's resource usage. With the block filter, users can exclude blocks through HDRF configuration files and potentially save some performance.

Block queue. A vanilla HDFS can parallelly process multiple block requests, which speeds up the block-read process by not waiting for other requests' completion. However, not all data reduction schemes can work in parallel. For example, a number of deduplication schemes are limited to one thread to maximize their deduplication ratio [70]. In such cases, users can use the block queue system to limit parallel block accesses.

Failures handling. System crashes and hardware failures are common and expected in production systems. HDFS handles this through block reporting. When a block is corrupted or has a mismatched length, the data node will report the block to the name node. Additionally, the name node will attempt to replace the block with one from other available nodes. HDRF takes the same approach as HDFS with a few changes. For the block size, a number of HDFS blocks in HDRF have 0-byte lengths. In this case, we made changes to the HDFS block length management system to check the length from the HDRF's metadata.

Another possible failure point of HDRF is the metadata server, which is isolated in each data node. When the metadata server fails, HDRF will report that the current node is broken to the name node. The name node will attempt to recover the data through the block replication of HDFS, which is similar to how HDFS handles node failures. HDRF handles other types of data corruption, node failures,

and other unexpected events similarly to HDFS.

3.4 Experimental results

This work answers the research questions presented in the Introduction by proposing and comparing HDRF to two existing setups, which are data reduction in the application and FS layers, respectively. For data reduction in the application layer, we used a vanilla HDFS setup where HDFS uses an ext4 FS and enables data reduction schemes through Hadoop CompressionCodecs in the applications. For data reduction in the FS layer, we used a ZFS setup where HDFS runs on top of ZFS and enables the schemes through ZFS.

We evaluated the proposed design by extending HDFS's code, which may affect its functionality when running various workloads. To show that the implementation can operate without any issue, we ran various workloads and observed HDFS's log in each data node. We ensured no errors such as mismatch checksum, missing blocks, or other related Java errors occurred in all setups. Additionally, for data processing results, we checked and compared the output.

Supporting new data reduction schemes is important to maximize the efficiency of the storage solutions. HDRF solves this through a data processing module that can load these schemes. HDRF can load any schemes that extend HDRF's data reduction scheme class or Hadoop CompressionCodecs class, which is similar to how Hadoop CompressionCodecs operates in the application layer. To demonstrate the operation of the solution, we implemented a local data deduplication scheme described in Section 3.3.2, which extends HDRF's data reduction scheme. We also tested it in a series of tests to compare it with ZFS's deduplication scheme.

The difficulty of enabling the data reduction schemes may vary among setups. Enabling these schemes for the vanilla HDFS setup is the most challenging because a number of applications do not support them. As for the ZFS setup, enabling the schemes is as easy as changing configurations of ZFS because the data processing in ZFS is transparent to the applications. To confirm the difficulty of enabling these schemes for HDRF, this work used different applications on the three different setups, discussed the difficulty of enabling these schemes on these setups based on our experience, and observed the data processing and storage overhead.

The cost of operation is also important because if it is too big, it might deter users from using the setups. To show this cost, we ran several applications and data access tests to measure and compare the overhead of the setups. The goal for HDRF is to have a negligible difference from the best approaches.

Applications. The Hadoop applications that we used for the data access tests

Table 3.1: Specification for the test nodes.

	Cluster A	Cluster B	Name node
CPU	Xeon E-2224 (4c4t)	E3-1220L V2 (2c4t)	i3-7100 (2c4t)
Memory	32-GB DDR4	8-GB DDR3	24-GB DDR4
Storage	Samsung 983 DCT 960-GB (800 MBps write, 2.5 GBps read)	Micron CT250MX500SSD1 250-GB (400 MBps write, 500 MBps read)	Sandisk SDSSDH32 2-TB (400 MBps write, 500 MBps read)
Network	10-Gbps	1-Gbps	10-Gbps
Number of nodes	14	6	1

Table 3.2: The datasets used in the storage overhead tests.

Datasets	Content	Size
Wikipedia	A compilation of Wikipedia dumps from the first five dumps in 2021 (20210101, 20210120, 20210201, 20210220, 20210301) [71]	380-GB
PhysioNet	A compilation of medical datasets provided by PhysioNet [4, 5, 72, 73]	96.9-GB
Cocoimages	A compilation of 2017 unlabeled and train JPEG-formatted images from cocodataset [74].	36.8-GB

are Hadoop DistCP and Hadoop Streaming, which are both MapReduce-based and cannot and can use Hadoop's CompressionCodecs, respectively. With Hadoop DistCP, a user can define the number of mappers and reducers in the configuration when running. Hadoop Streaming can read and process the input data by using an input format before storing it in HDFS. The default input format is TextInput-Format, which reads a file line by line. However, it is not as useful for our testing because it adds a line number at the beginning of each line, increasing the storage space and making it difficult for us to compare it with Hadoop DistCP. To solve this, we made a custom input format based on TextInputFormat but without the extra line numbering. The number of mappers and reducers for Hadoop Streaming is defined by the split size. In the tests, we maintain the number of splits to match the number of mappers for Hadoop DistCP to ensure the fairness of the tests.

Additionally, we included Intel's big data benchmark called Intel HiBench, which can test the performance of the cluster when running MapReduce and Spark workloads. The workload that we used is Wordcount, which loads and reads the datasets

directly from HDFS and then stores the results back in HDFS. HiBench can generate the dataset by using a MapReduce application with configurable size.

Cluster specifications Table 3.1 lists the specification of the clusters. We used two clusters, A and B. Cluster A has more nodes, higher compute power, more I/O performance, and more importantly, a higher network speed. Cluster B consists of nodes with older hardware and a slower network speed, which is only 1-Gbps maximum. We ran all tests on Cluster A except for the network scaling test, which was run on both clusters to show the impact of replication scaling on older hardware. All tests were run three times and the results provided here are the average of the three runs.

Datasets The evaluation used three different types of datasets to measure the storage overhead of the tested setups. The first dataset is the Wikipedia dump dataset with a size of almost 400 GB and represents a human-readable dataset. The second dataset is the medical dataset from PhysioNet [5] with a size of almost 100 GB and contains log data from machines like electrocardiograms (ECGs) and patient activity records. The third dataset is an image dataset from Cocoimages with a size of around 35 GB and contains JPEG-compressed training and unlabeled images for machine learning applications. Table 3.2 shows the details of the datasets. We chose these datasets to show the storage overhead of HDFS when storing various dataset types. This test uses 3-times replication, which is the default for HDFS.

The structure of the tests This work answers the research questions presented at the beginning of this Section through several tests with the described applications, clusters, and datasets. Because directly answering the first (1) and second (2) questions is difficult, we structured the tests to observe the overhead of the tested setups and answered the questions through the data presented in the results. The test structure is as follows. First, this work measures the throughput of the clusters for data access within the storage nodes in Subsection 3.4.1. Second, it demonstrates the overhead of HDRF when uploading and downloading the dataset from the cluster in Subsection 3.4.2. Third, it evaluates the storage consumptions in Subsection 3.4.3. Next, it shows the performance of the tested setups in MapReduce and Spark workloads. Finally, it discusses the network overhead when replication is enabled in Subsection 3.4.5.

This work shows the ease of adding new data reduction schemes in Subsections 3.4.2 and 3.4.3 through the implementation of the data deduplication scheme in HDRF. The ease of enabling the schemes can be observed in Subsection 3.4.2 where we attempted to enable data reduction schemes in Hadoop DistCP and Hadoop Streaming and in 3.4.4 where we attempted to use Lz4 on the tested setups for MapReduce and Spark workloads. The overhead is shown throughout the test results

by comparing the results from the different setups.

3.4.1 Data access overhead test

Big data platforms such as Hadoop can use the storage nodes to generate data and store it directly in the DFS. Additionally, the storage nodes can also process the data from its storage device directly. To show the overhead of HDRF in such workloads, we generated and stored a 100 GB dataset in the cluster by using the HiBench prepare tool, which uses RandomTextWriter to generate the dataset. To evaluate the read overhead, we used Hadoop Streaming to read the data and set the output format as NullOutputFormat to simply evaluate the read performance.

The results for the data access overhead test are depicted in Figure 3.5. For write, the results show minimal overhead when running the same data reduction schemes. For deduplication, HDRF performed worse than ZFS because the deduplication scheme in HDRF is more computationally intensive with the CDC algorithm and smaller chunk size than the deduplication scheme in ZFS. Vanilla HDFS performed worse when we enabled Lz4 CompressionCodec through the configuration. For read without any reduction schemes, the results indicate around 12% performance overhead when compared to vanilla HDFS and 17% compared to ZFS. With the Lz4 enabled, HDRF performed around 18% to 23% slower for reading because of the extra processing in the storage node to alter the data access path. For the read performance with deduplication enabled, HDRF performed around 45% slower than ZFS's deduplication because of the more complex processes. The overhead of HDRF might look significant in this test because these tests simply read and write the data in the storage nodes. With other applications such as data transfer with Hadoop Streaming or WordCount, the overhead is significantly smaller because of the involvement of other operations that masks the overhead. Additionally, the overhead might decrease with better integration in the DFS software.

3.4.2 Data transfer overhead test

To confirm the overhead of HDRF in data transfer, which is a basic and crucial task for DFSs, we compared HDRF with the vanilla HDFS and ZFS in a data transfer test. This test uses 50 GB of the Wikipedia dump dataset in the form of split 1-GB files to measure the data transfer speed for the tested setups. For the test configuration, we chose 1-time replication to show the compute overhead without being affected by the network during the replication. We simulated the 1-Gbps network by using Wonder Shaper 1.4.1 [75]. The applications used in these tests are Hadoop Streaming and DistCP.

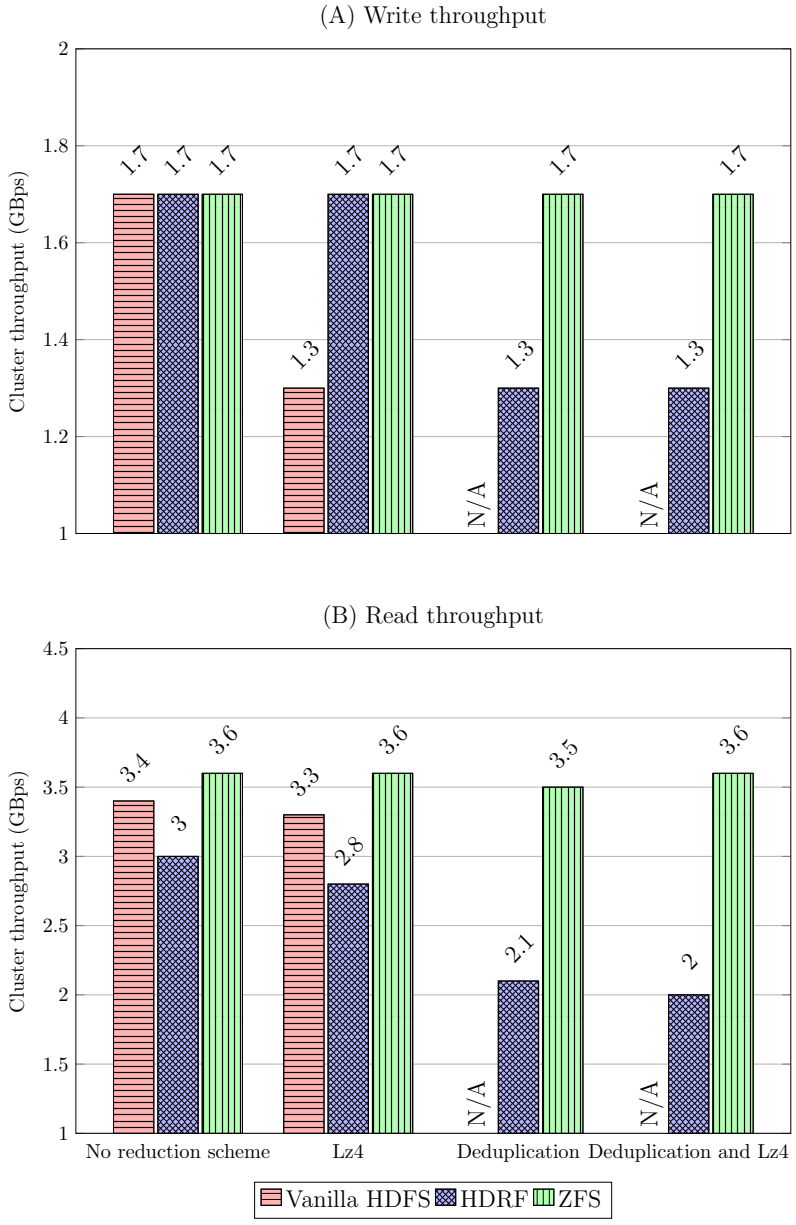


Figure 3.5: Cluster A's throughput for write with RandomTextWriter and read with Hadoop streaming. Vanilla HDFS does not have an official support for deduplication.

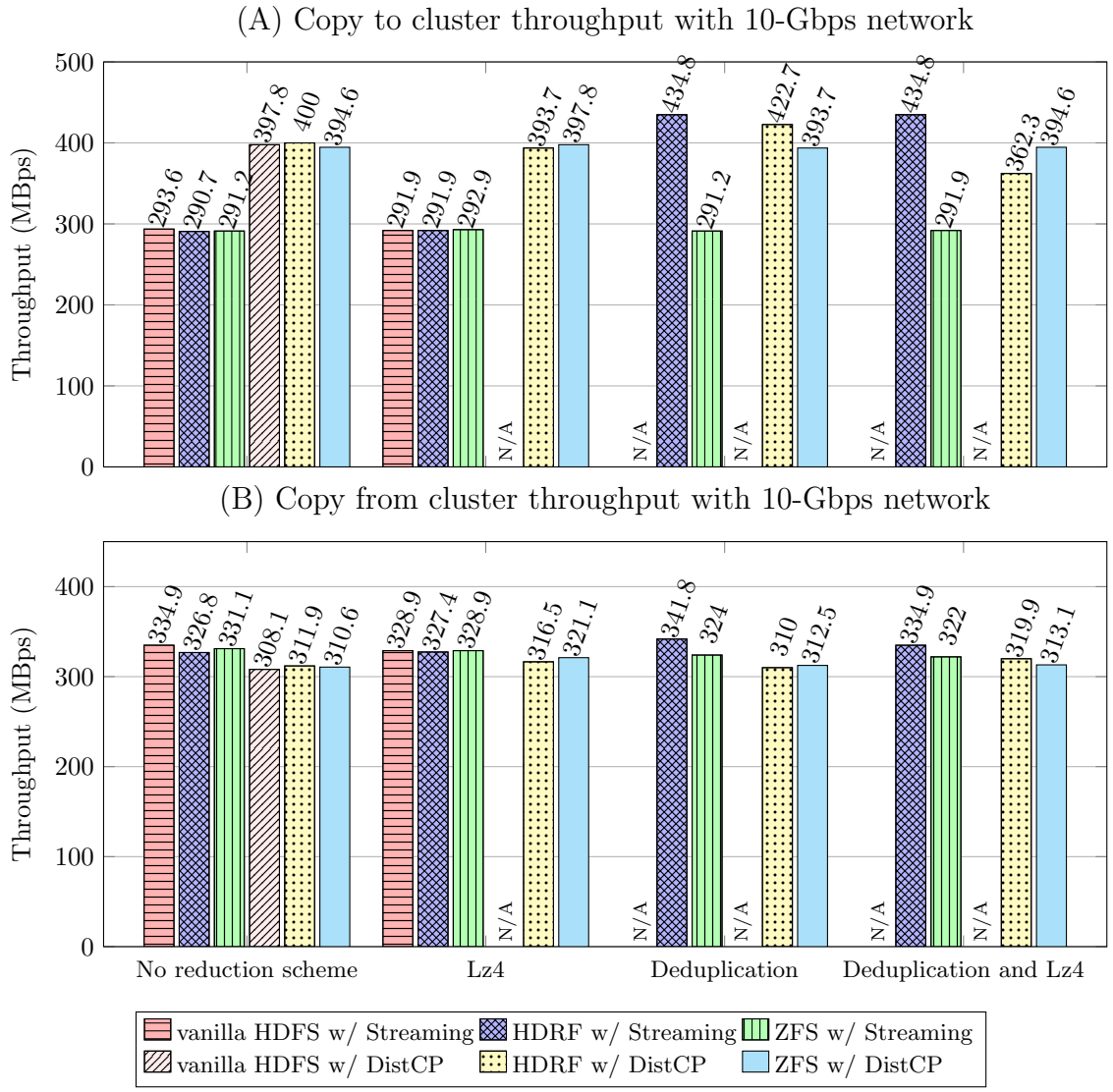


Figure 3.6: Throughput for dataset transfer between the client and Cluster A. DistCP on the vanilla HDFS cannot use any CompressionCodecs, and the vanilla HDFS does not officially support deduplication. Higher values are better.

Figure 3.6 presents the throughput recorded when transferring the dataset between the cluster and the client. In this test, we used the deduplication scheme described in Subsection 3.3.2. Adding the scheme to HDRF is fairly easy through the provided abstract class. Throughout this test, we noticed no errors when the HDRF and ZFS setups ran the schemes in both Hadoop DistCP and Streaming. In contrast, for the application layer setup, the vanilla HDFS, we were unable to enable Hadoop CompressionCodecs on Hadoop DistCP through the command-line interface. DistCP ignored the CompressionCodec configuration in the command line and stored the data as if no CompressionCodec was specified.

For the write throughput, the differences between the vanilla HDFS, HDRF, and ZFS are less than 1% with the same reduction scheme, which shows that the overhead of HDRF is insignificant. Additionally, HDRF with deduplication enabled had close to a 50% higher throughput compared with that of the vanilla HDFS with Hadoop Streaming. The reason is that the deduplication scheme in HDRF uses the block array mode, which buffers the data in the memory before storing it in the disk, and thus maximizing the client's disk read bandwidth and network throughput. For the read overhead, the results depicted in Figure 3.6 show that the overhead is almost non-existent because the results for the vanilla HDFS, HDRF, and ZFS are negligible. In the best-case scenario, HDRF can eliminate the storage footprint by more than 50% while transferring the data at a higher throughput when compared with the vanilla HDFS without any reduction scheme.

Another observation we made from Figure 3.6 is that DistCP is up to 25% faster than Hadoop Streaming for write operation with most schemes because DistCP does not have extra processing. Additionally, DistCP reads and writes the data in chunks unlike Hadoop Streaming, which reads and writes in lines. However, with deduplication, the difference is less significant because HDRF absorbs small writes in memory for block array stream mode, used by the deduplication scheme. These results suggest that HDRF can provide many options to the users to minimize the data transfer time when applying data reduction schemes.

3.4.3 Storage overhead test

In this test, we uploaded three different datasets into Cluster A with Hadoop Streaming and measured the storage space usage of the tested setups. We chose Hadoop Streaming because it supports Hadoop CompressionCodecs Lz4, which is important when comparing the HDRF and ZFS setups. We then evaluated the storage overhead by comparing the space usage for the three different setups. We did not test the vanilla HDFS with deduplication because of the strict specification of Hadoop Com-

pressionCodecs that requires the scheme to have stream support, which is missing from the described deduplication scheme in Subsection 3.3.2.

Figure 3.7 shows the results for the storage used by each setup with no processing, Lz4 compression, deduplication, and the combination of the two. From this figure, we learned four facts. First, the overhead of HDRF is small and less than 1% of that of the vanilla HDFS across the three datasets with Lz4. Second, even though Hadoop CompressionCodecs and ZFS use the same Lz4 algorithm, they use different configurations. The Lz4 CompressionCodec in the vanilla HDFS and HDRF has a better compression ratio than that in ZFS. Third, HDRF's deduplication can reduce up to 15% of the dataset storage consumption, which is better than the deduplication scheme used in ZFS because HDRF's scheme uses a CDC algorithm. Fourth, neither Lz4 nor deduplication performs well across any of the datasets. Lz4 performs best with the human-readable text in the Wikipedia datasets with over 50% data reduction and reduces around 14% of the storage footprint for the log data in the PhysioNet datasets. However, in the image dataset, neither Lz4 nor deduplication could reduce the dataset further. Even worse, deduplication increases the storage footprint because of the deduplication metadata. This last information also shows the importance of the ease of adding new reduction schemes, to enable users to use a new reduction scheme to match their dataset's type.

3.4.4 Data processing test

This test utilizes a commonly used big data benchmark tool from Intel called HiBench. It can measure the performance of the cluster when running MapReduce and Spark workloads. For this test, we generate a dataset with HiBench and run a test for data processing. We chose WordCount as our data processing task and ran it for all tested setups with 3-times replication, which is the default for HDFS. The test has two phases: prepare, which generates 32 GB of a "huge" dataset, and process, which uses MapReduce to perform the word count. The number of executors, mappers, and reducers is set to 28. The prepare phase uses MapReduce to generate random words and store them in HDFS.

With this test, we observed that HiBench's Wordcount MapReduce workload was unable to use Hadoop CompressionCodecs. We tried the configuration "hibench.compress .profile enable" and "hibench.compress .codec.profile 'lz4' " but it still consumes the same storage space. However, adding compression options in the MapReduce command line reduces the dataset from 96 to 93 GB, which is still far larger than the other setups. In comparison, enabling Lz4 in HDRF can reduce the dataset to 42 GB, which is over 50% reduction in storage footprint. This observa-

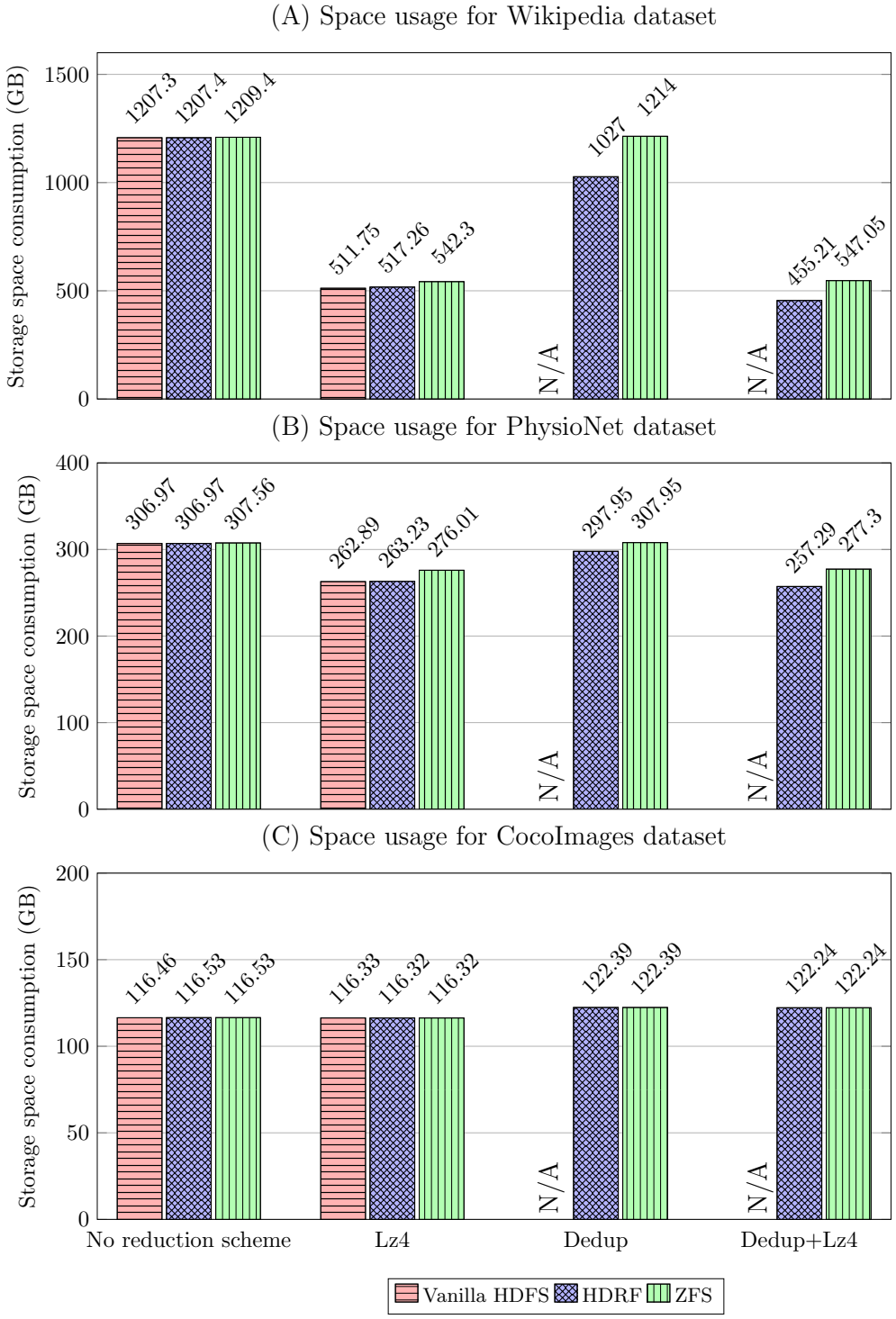


Figure 3.7: The storage space required to store the datasets with Cluster A. No processing means the dataset is directly stored as is. Dedup means deduplication is applied to the dataset. The vanilla HDFS does not officially support deduplication. Lower values are better.

tion indicates that HDRF is transparent to all applications without the hassle of configuring each application.

A similar situation also occurred when running the Spark workload counterpart. When we prepared the dataset with Hadoop CompressionCodecs, the process phase crashed stating that the Lz4 native library could not be loaded. We debugged the application for hours attempting to fix the issue and noticed that IOCommon failed to load the compressed dataset. Additionally, we confirmed with the "hadoop checknative -a" command that Lz4 is indeed installed properly in our cluster and is available natively. HDRF and ZFS can run the MapReduce and Spark workloads with Lz4 enabled without any issue. This experience on enabling Hadoop CompressionCodecs on HiBench shows that enabling the data reduction schemes can often be challenging and frustrating.

As for the overhead, the results shown in Figure 3.8A indicates that the overhead for data processing applications like MapReduce is around 3% for HDRF, and it can perform as well as the vanilla HDFS with and without compression. With the Spark workload, HDRF is around 5% slower than the vanilla HDFS because of the extra processing by the framework when loading the data from HDFS. The overhead with Spark is larger because Spark is much faster than MapReduce at processing the dataset, increasing the significance of the dataset load operation from HDFS. With a more compute-intensive workload, this overhead may become smaller.

3.4.5 Network overhead test

One of the issues for lower-layer data reduction like HDRF and ZFS is the lack of network traffic reduction and the need to reprocess the data at each node during block replications, thus increasing the data transfer time. For the application layer data reduction, these issues are not a concern because block replication occurs in the DFS layer and the data there is already reduced, eliminating the need to reprocess the data at the replication's destination nodes. To confirm this hypothesis, we scaled the network speed and replication factor of HDFS with Clusters A and B. To show the overhead of replication, each node has the dataset stored locally in the solid-state drive (SSD) to eliminate the overhead of the network transfer from the client to the data nodes. The dataset is the same as that used in the data transfer test, which is the first 50 GB of the Wikipedia dump.

Figure 3.9 and 3.10 depicts the results for the network scaling test on Cluster A with 6 nodes. According to the results, the impact of replication is more significant with a slower network and fewer nodes. With slower networks, setups that reduce the data at lower layers like HDRF and ZFS significantly consumed more time to

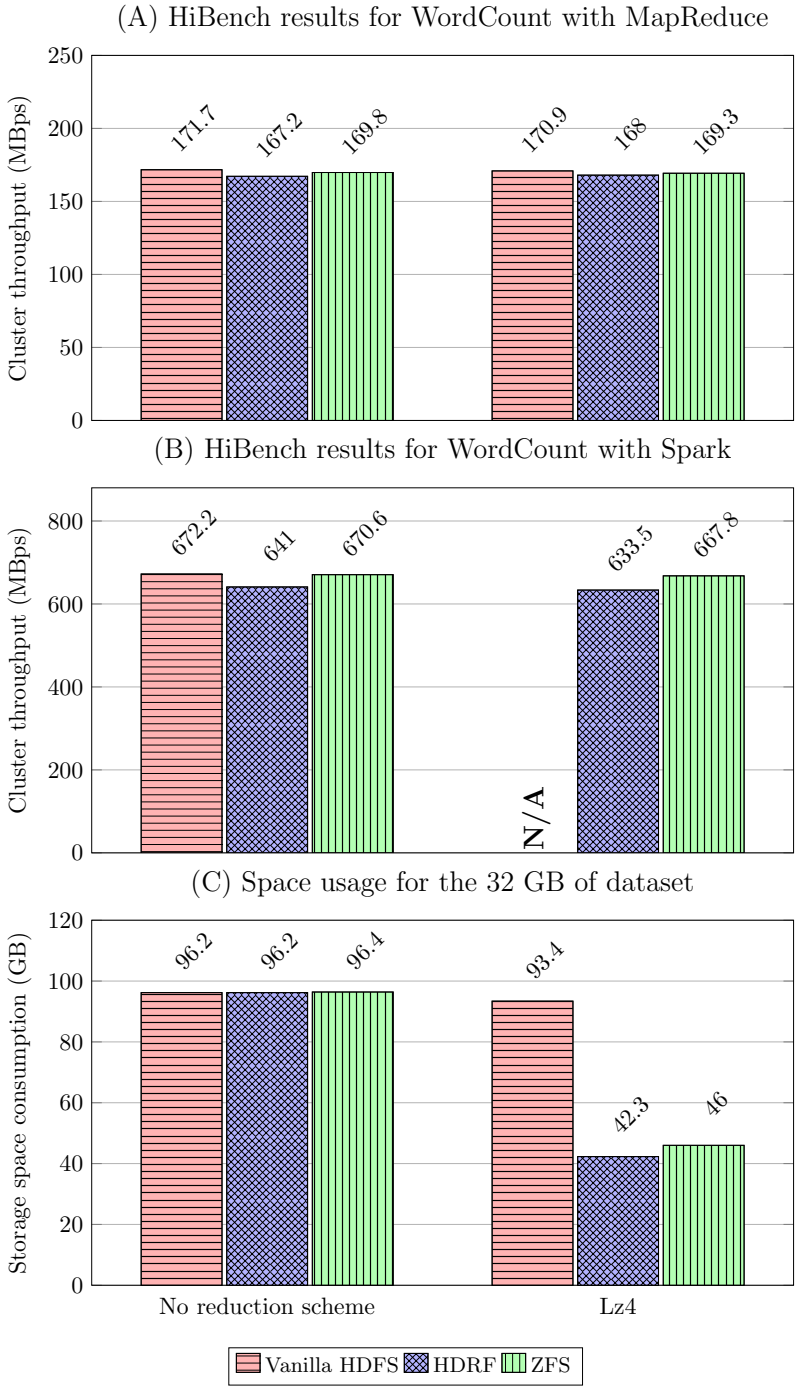


Figure 3.8: Throughput for the HiBench WordCount workload with Cluster A. Higher values are better.

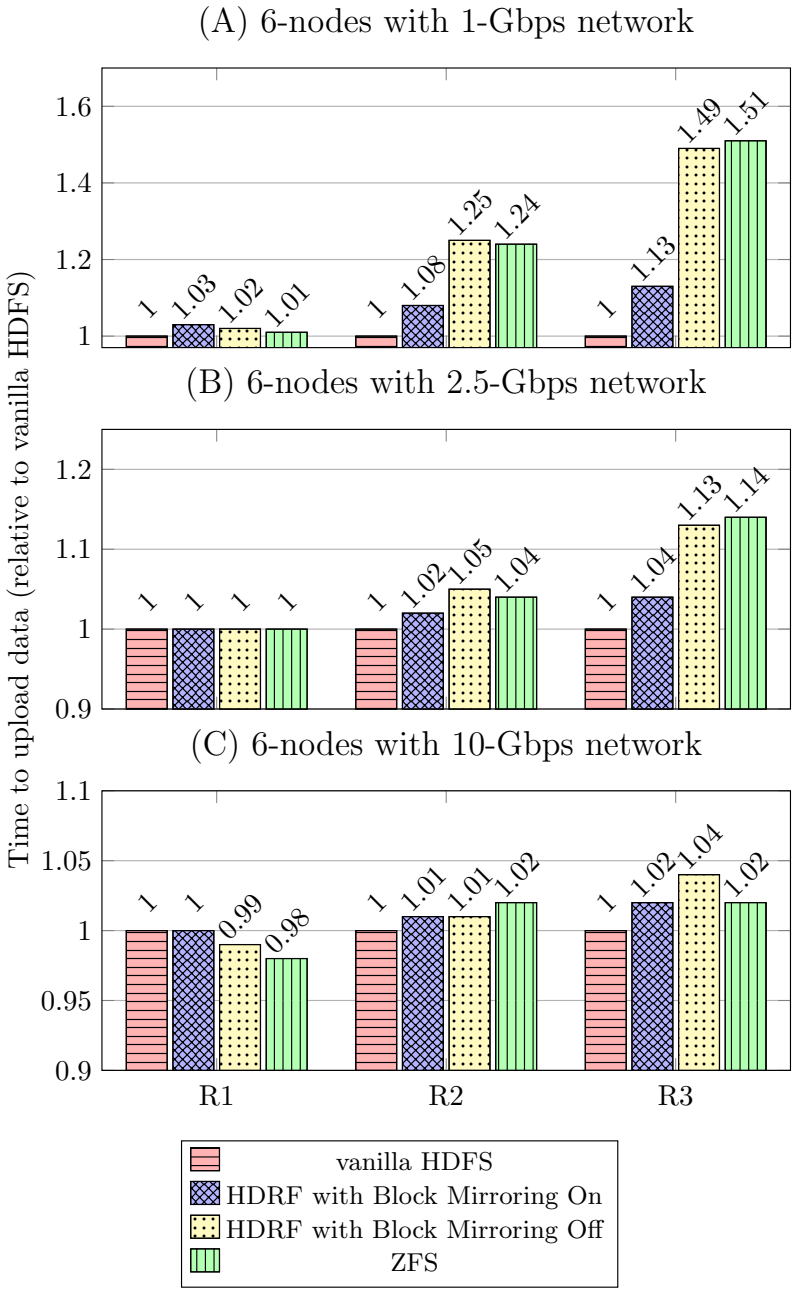


Figure 3.9: Data upload time from the client to the cluster with replication scaling on 6 nodes of Cluster A with 1-Gbps, 2.5-Gbps, and 10-Gbps network and Lz4 compression. The number behind R indicates the number of replication factors for each block. For example, R2 means 2-times replication. The result is relative to that of the vanilla HDFS. Lower values are better.

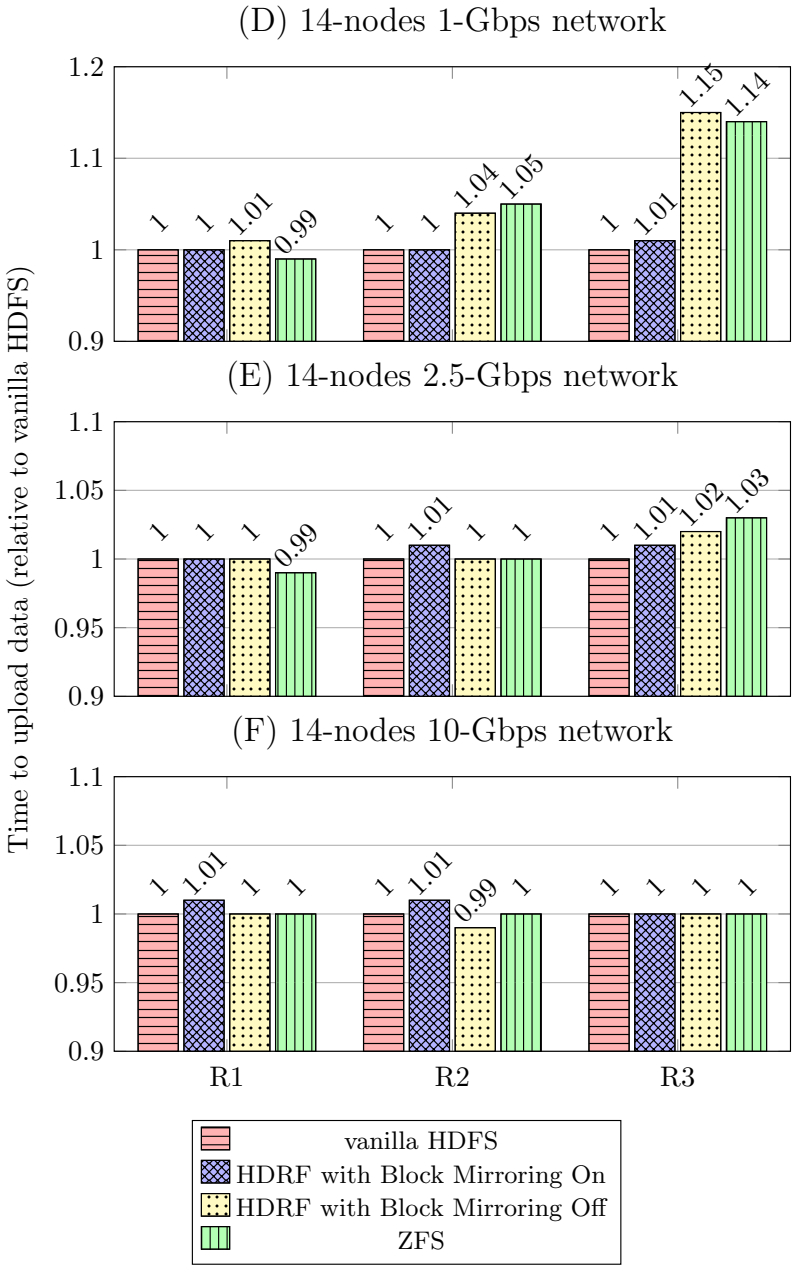


Figure 3.10: Data upload time from the client to the cluster with replication scaling on 14 nodes of Cluster A with 1-Gbps, 2.5-Gbps, and 10-Gbps network and Lz4 compression. The number behind R indicates the number of replication factors for each block. For example, R2 means 2-times replication. The result is relative to that of the vanilla HDFS. Lower values are better.

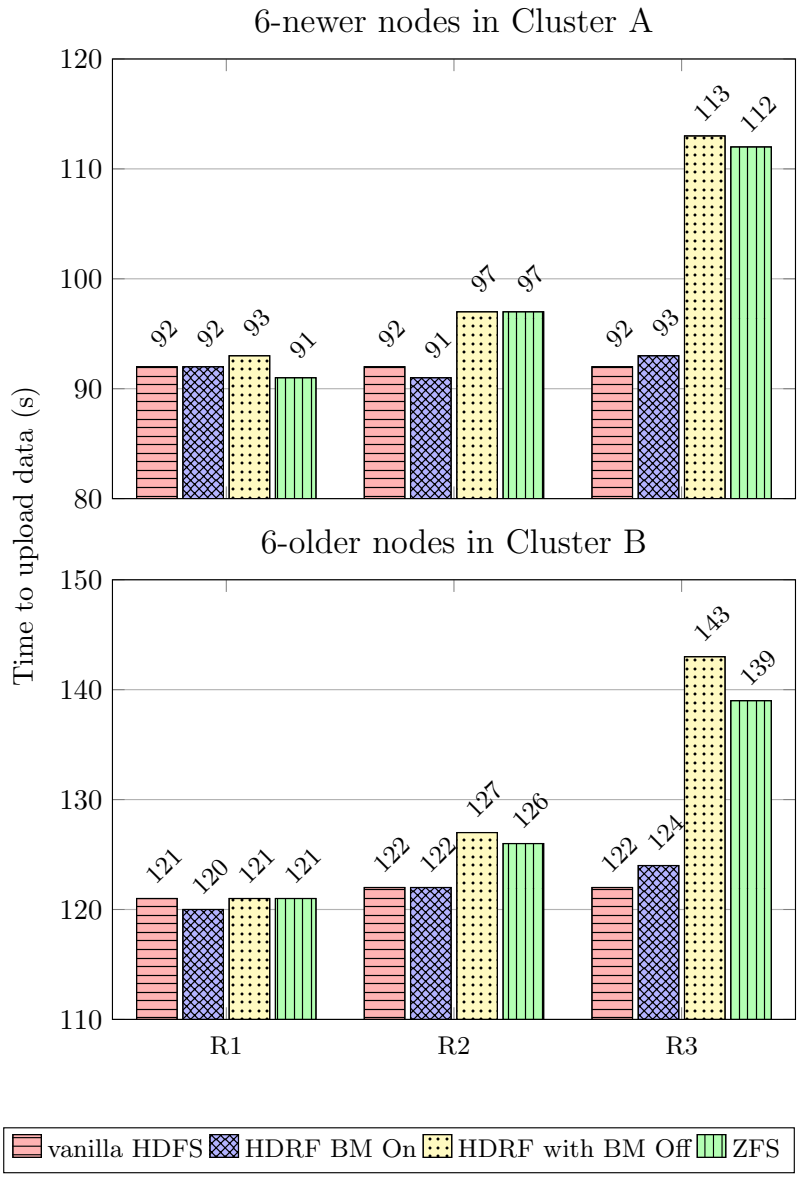


Figure 3.11: Time to upload with replication scaling for six nodes in Clusters A and B on a 1-Gbps network. BM stands for Block Mirroring. Lower values are better. The results are not scaled to the baseline to show the difference between Cluster A and Cluster B.

finish because they transfer the data without any data reduction. Additionally, they need to recompress the data at each replication's destination nodes, which is inefficient compared with the vanilla HDFS that reduces the data at the application layer. With fewer nodes, the effect is more significant because the amount of data is the same, meaning each node now must process more data. With block mirroring, HDRF can almost match the data transfer time of the vanilla HDFS with faster networks because it transfers the reduced data instead of the original data, thus minimizing the data transfer time and eliminating the need to reprocessing at the destination nodes. With the slow 1-Gbps network, there is around 13% of data transfer overhead for HDRF with block mirroring enabled over the vanilla HDFS, which is significantly smaller than that without block mirroring.

As data processing also depends on the node capability of data processing, we also compared the newer nodes in Cluster A with the older nodes in Cluster B. In this test, we used the first 20 GB of the Wikipedia dump. As shown in Figure 3.11, the data transfer overhead is more significant in the newer nodes at around 23% when compared with the older nodes at around 17% overhead for ZFS and HDRF without block mirroring. The faster disks in the newer nodes decrease the amount of time to read and write the data to HDFS, increasing the importance of the data processing capability for a task like data compression. With the block mirroring, the overhead over the vanilla HDFS is smaller because the destination nodes do not need to reprocess the data.

3.4.6 Discussion

With these tests, we answered the research questions presented in the Introduction, which are:

1. Is it possible to enable the data reduction schemes in the DFS software without affecting most of its functionality?
2. How easy is it to add and enable the schemes in the DFS software?
3. How big is the performance overhead for the schemes in the DFS software when compared with the existing solutions?

We answered the first question by adding the support for data reduction schemes to an existing DFS called HDFS with around 1000 lines of codes. We also tested it in various workloads and confirmed that there are no issues with its operation. We answered the second question by adding a deduplication scheme to HDFS through HDRF and enabling schemes such as Lz4 and deduplication in various applications. During the implementation and experiments, we observed that adding new data reduction schemes is fairly easy through the provided abstract class. Additionally, HDRF supports Hadoop CompressionCodecs, making scheme implementation as easy as those in the application layer. Enabling the schemes is similar to the FS layer approach. HDRF's data processing is transparent like that of ZFS in the FS layer. To answer the third question, we compared HDRF with the existing approaches in various workloads and observed the overheads. The data transfer and storage space overhead is fairly low at around 1%. With the Spark and MapReduce workloads, the overhead is around 5%. These results indicate that the proposed DFS design can combine the benefit of existing approaches at a fairly low overhead cost.

With the test in 3.4.3, this work also demonstrates that not all data reduction schemes are equal and some schemes work better than others for certain data types. For example, HDRF's deduplication schemes can reduce more redundancy than those of ZFS. Additionally, deduplication and compression do not perform well on datasets with log or image files. This result shows the importance of having easy access to the DFS to add and enable the schemes to the DFS.

3.5 Conclusion of this chapter

This work explores the application of data reduction schemes in big data systems with DFSs such as Hadoop. Existing approaches have challenges in terms of the

ease of adding or enabling schemes in the system. These challenges might discourage users from using these schemes to maximize their storage space efficiency. This work proposes a new DFS design that can combine the benefits of existing approaches by applying the data reduction schemes directly in the DFS layer. The advantages of this approach are threefold. First, adding a new scheme can be as easy as doing so in the application layer. Second, it can apply the data reduction scheme transparently regardless of the application. Third, the overhead is negligible compared with that of the other approaches.

To show the effectiveness of the proposed design, we implemented it in an open-source DFS called HDFS and compared it with a vanilla HDFS and the data reduction scheme in the FS layer through ZFS. Our results and experience dealing with the tested setups show that adding a new scheme through HDRF is as simple as through the vanilla HDFS or even easier; enabling the schemes is as easy as the FS layer approach because it does not require any changes on the application side. Our results also indicate that HDRF has a small overhead of around 23% for data access, around 1% for data transfer and storage space overhead, and less than 5% for compute workload.

Chapter 4

High-performance KVS based on LSM-tree

4.1 Introduction

Log-structured merge-tree (LSM-tree) is a write-optimized sorted data structure that has multiple levels with increasing size from top to bottom based on the configured ratio. One of the key features of LSM-tree is its ability to saturate the throughput of most storage devices because it only writes sequentially, which is suitable for use in the metadata system in a distributed storage solutions [13]. When a level reached its limit, it flushes and merges the data with the next level. The main drawback of LSM-trees is that the merge process can be computation heavy, which may increase the tail latency for writes occurring at the same time as the merge operation. It also rewrites KV pairs multiple times during merge operations, which increases the write amplification.

WiscKey [76] solves this issue by reducing both the tail latency and the write amplification is to write KV pairs unsorted in a log file and index them in an LSM-tree. To further improve the performance, it uses an in-memory write buffer to absorb small writes. However, write-intensive workloads may overload the buffer return to normal sequential write throughput of the storage device. Additionally, it increases the risk of data loss during system crashes. WiscKey uses a garbage collector that reads the log file from the head and checks the validity of the KV pairs in the LSM-tree. When a KV pair is still the latest version, the garbage collector appends the KV pair to the tail of the log. As a trade-off, it may suffer from high write amplification because the garbage collector may rewrite the KV-pairs back when all KV-pairs in the log are still valid and the log file is reaching its size limit.

Persistent memory or non-volatile memory (NVM) is a storage class for persistent storage that is directly attached to the CPU's IMC. It has many advantages such as low data access latency, byte-addressable, and high endurance when compared to other persistent storage solutions. In 2019 Intel announced an NVM product called PMem that uses 3D XPoint technology. PMem comes with a higher capacity than DRAM at a lower price point. It also has a higher endurance when compared to NAND-based flash storage solutions. However, it has 3 to 5-times the latency of DRAM. Because it is directly attached to the IMC, users can access PMem in bytes, which can be useful for persistent data structures. One of its unique characteristics is that PMem does not scale well with a high number of write threads because of its limited write buffer [77]. It may decline in performance if the number of write threads is too high.

One of the solutions to improve the tail latency and write amplification of LSM-tree is to use PMem, which has low data access latency and high endurance to cope with the high write amplification of log-based LSM-tree. However, directly using the log writers from the existing approach may degrade the performance of the PMem because too many threads can overload the write buffers of PMem. This work explores a solution that detaches the client's write threads from the PMem by using asynchronous multithreading.

4.2 Basic concept

In this section, we discuss the technology behind LSM-tree and NVM in detail.

4.2.1 LSM-tree

LSM-tree is a sorted data structure for KV storage that uses multiple levels that have increasing sizes based on the fan-out ratio. For example, if the fan-out ratio is k and the base size is x-bytes, then the size for level-n would be $k \cdot x^n$. Each level in the LSM-tree may use different data structures to maximize the performance for different storage mediums. The first level of the LSM-tree is called the memory table. The most common data structure for the memory table is skiplist, a sorted data structure that provides good performance for read and write-operations. As for the on-disk data structure, most LSM-trees store the data in sorted string tables (SSTables). Depending on the KVS, the SSTables may have different structures of key, value, and metadata placement. Partitioning the SSTables, like in RocksDB, may also help to reduce the write amplification during compaction because the LSM-tree only needs to update a few SSTables instead of the whole level.

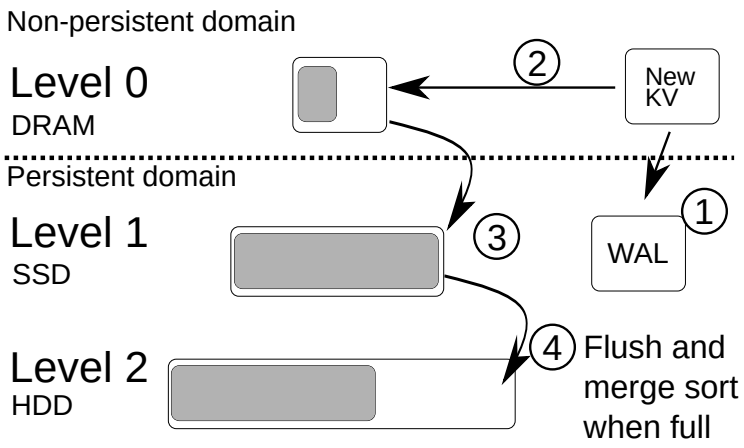


Figure 4.1: The write-process of an LSM tree. (1) It inserts the KV-pairs to the WAL. (2) L0 in the DRAM absorbs the KV-pairs. (3) Once the L0 is full, the LSM-tree flushes the data to the next level. (4) The same process also applies to the lower levels.

LSM-tree is a write-optimized data structure because it has to write buffers in the DRAM, which absorbs small writes in the memory table, stored in the DRAM. Additionally, the writes to the disk are sequential, which works well with both SSD and HDD. In some LSM-tree-based KVS like RocksDB, there are two ways to perform writes to the LSM-tree: (1) through the individual insertion of the KV-pairs and (2) through batching, which groups individual writes into a single batch. The former has lower latency, but it stresses the LSM-tree because the LSM-tree might perform duplicate operations for different KV-pairs. The latter has higher latency because the first insertion will have the same completion time as the last insertion in the batch. However, the LSM-tree can amortize the write-operations by performing fewer operations for the batch when compared to the individual insertions.

Write-operations in the LSM-tree start from the highest level, which is the level with the lower level number or usually also called the level-0 (L0). First, it writes new KV-pairs into a write-ahead log (WAL) to ensure atomicity. During unwanted events such as system crashes, the tree can replay the WAL to recover lost data. Second, it inserts the pairs into a sorted data structure such as skiplist as a memory table in the L0. Third, it flushes each level when it reached the capacity threshold. This flush process will merge the current level with the next level. This whole process is shown in Figure 4.1.

Compaction strategy. LSM-trees have multiple compaction strategies, which

influence the structure of the trees. The original LSM-tree [20] features a leveled design where each level only has a single run, which is a sorted table. This design is optimal for read-operations because we only need to travel one run for each level to get a KV-pair. However, the compaction process will require the database to rewrite the whole run in the next level to merge sort the new data, which can be expensive in terms of write amplification. Tiered design, which is used by Cassandra [78] and is available as an option in RocksDB [79], has multiple runs in each level instead of a single run in the leveled design. It only merges the runs into a single run when the level is full and flushes it to the next level without merging it with the runs in the next level. The tiered design is more efficient in terms of write amplification. However, a query will consume multiple IO because it checks multiple runs in each level.

LSM-tree based KVS. Because of the flexibility of LSM-trees, many KVSs like LevelDB [80], RocksDB [79], and Cassandra [78] use it as the main data structure to maintain the data. Among these KVSs, RocksDB, which is a database developed by Facebook based on LevelDB, has many types of optimizations to maximize its throughput for different workloads on SSDs. For example, it is configurable to use leveled, tiered, or FIFO, which is a single level LSM-tree, compaction scheme. By default, it uses a level design, which is inherited from LevelDB. However, their leveled strategy maintains multiple memory tables in the memory to boost up the write throughput through parallelization. Because of this fact, it is a hybrid LSM-tree that uses tiered compaction in the L0 and leveled compaction for other levels. In addition to these compaction strategies, RocksDB also has point and block caches to maximize the throughput for workloads.

RocksDB is one of the most updated KVS out there and many studies have used RocksDB to experiment with new ideas. For example, Speicher [81] adds shielded execution to RocksDB to protect against rollback or forking attacks, SpanDB [82] implements SPDK in RocksDB to maximize its throughput when used with Optane drives, and TRIAD [83] experiments with caching and batched I/O operations. In some cases, RocksDB adopts some of these ideas to further improve its performance.

LSM-tree with KV separation. WiscKey [76] is a key-value store that writes the keys and values to log files and indexes the keys in an LSM-tree to improve write latency and reduce the write amplification of the LSM trees. WiscKey is based on LevelDB and uses leveled design. Although, the concept applies to any LSM-tree compaction strategies, the effect on the tiered compaction strategy is unknown because WiscKey was not tested on compaction strategies other than leveled strategy. It works by writing the KV-pairs in log files and inserting the keys and pointers to the values in the log files to the LSM-tree. In WiscKey, the LSM-tree is significantly

smaller than traditional LSM-trees. In some cases, the whole tree can reside in the DRAM as demonstrated by Im et al. [84], which improves the read performance. It performs garbage collection by checking and reinserting the old KV-pairs from the log files into the LSM-tree. When the pair is valid then WiscKey reinserts them into the log and updates the LSM-tree. Because most of the writes occur in the SSD, WiscKey's main bottleneck is the SSD. With a faster storage device, KVS with KV separation like WiscKey can achieve higher throughput without heavily modifying the LSM-tree.

WiscKey has several drawbacks related to its design and side-effects of the KV separation. First, WiscKey may suffer performance degradation from a high number of client threads depending on the storage devices. WiscKey solves this by buffering the writes in the memory before flushing to the SSD. Although this solution can minimize the load at the SSD because DRAM can handle the parallel load better, this solution depends on the capacity of the buffer. Once the client write is faster than the speed of flush to the SSD, the problem becomes prominent again. Second, this also means that write is not atomic without a write-ahead log (WAL) because data might be lost if there is an unexpected system crash. Third, writes to the log may occur during compaction with large datasets because the tree's size may surpass the limit on the DRAM, increasing the write latency. Additionally, write amplification is still an issue because the garbage collector still rewrites some part of the data back to the SSD.

4.2.2 Persistent memory

Persistent memory, non-volatile memory (NVM), or also called storage class memory (SCM) is a type of storage device that is attached to the IMC of the processor. Like DRAM, NVM is byte-addressable and has low latency data access, which is suitable for high-performance applications. The first commercial product to features persistent memory is Intel's Optane DCPM memory [85], which uses phase-change memory (PCM) technology. NVM can be configured interleaved or non-interleaved. Interleaved NVMs have a higher throughput compared to non-interleaved because multiple DIMMs can share the workload [86, 87].

The writes to the persistent memory are not always guaranteed because the data might have not left the CPU's cache and might be gone if unexpected shutdowns occur. To ensure writes are persisted in the memory, applications can use cache line flush operation by using the processor instructions such as CLWB, CLFLUSH, and CLFLUSHOPT [88]. In most cases, CLWB is recommended over the other instructions because it only persists the data in the NVM without evicting the data

Table 4.1: The Fio sequential write test's latency results. The lower is better.

	1 thread average (us)	20 threads average (us)
DEVDAX	0.66	193.24
FSDAX	2.11	38.67
SSD	748.12	106663.84

from the cache, such that the data is still accessible through the CPU's cache for fast access. Intel also provides persistent memory libraries such as libpmem [89], libpmem2 [90], libpmemobj [91] to help application programmers to access these storage devices and persist their data.

The users can configure the NVM in-memory mode, which allows the NVM to act as volatile memory and extend the DRAM, or in-app direct mode, which persists the data even after a power cycle. In-app the direct mode, users can configure the persistent memory as a character device (DEVDAX) or use it with a file system (FSDAX). Based on our knowledge, FSDAX is compatible with almost all libpmem libraries, while DEVDAX is only accessible through open and mmap system calls and the libpmem2 library. In this work, we implemented our code with libpmem2 to persist and access the data.

Compared with other storage technology, NVM the lowest latency for the NVM is achievable through the DEVDAX mode, which provides direct access to the NVM without a filesystem like a DRAM to applications. To compare the latency of NVM's DEVDAX mode, FSDAX mode, and SSD, we ran a simple Fio [92] sequential write test with 4 GB of data. Our test system uses Intel Xeon 5218 (16 cores, 32 threads), 2 DIMMS of 16 GB DDR4 memories, 2 DIMMS of Intel Optane DCPM NMA1XXD128GPS, and Micron 5200 480 GB SSD with 300 MBps sustained write throughput. The test system is running Ubuntu 20.04.2 LTS with Linux Kernel 5.5.0 74.

As shown in Table 4.1, the mean latency for DEVDAX is significantly lower for single thread (1T) sequential writes, and we recorded up to 4.23 GBps throughput with our 2 DIMMS interleaved Optane setup. The result also shows that latency degrades with more threads, especially for the NVM with DEVDAX mode. Additionally, as shown in Figure 4.2, only DEVDAX experiences degradation for bandwidth with a higher number of threads because of the write buffers limitation of the NVM [77]. These results indicate that DEVDAX mode can achieve the lowest latency for NVM devices, but it may suffer performance degradation when overwhelmed with too many write threads.

A unique characteristic for Intel PMem is that write throughput does not scale

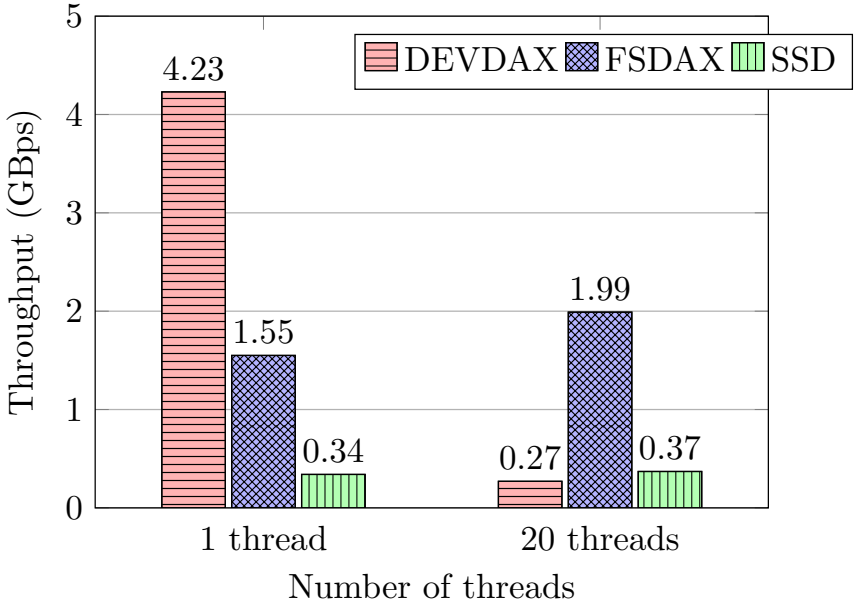


Figure 4.2: The Fio sequential write test's bandwidth results. The FSDAX uses Ext4 DAX and libpmem engine. The higher is better.

linearly with the number of access threads. As shown by Yang et al. [77], the write throughput stops to increase at 4 threads and starts to decline after 5 threads. This characteristic can be detrimental for most KVS that does not limit the number of writer threads such as WiscKey on write workloads that rely on multithreading to generate the data. For example, in our test system, using multiple YCSB threads is crucial to produce a high amount of data as shown in Figure 4.3. In this example, the data generation scales almost linearly with the number of threads. In such cases, limiting the number of client threads to avoid degrading the NVM performance can be inefficient in the system utilization for the job.

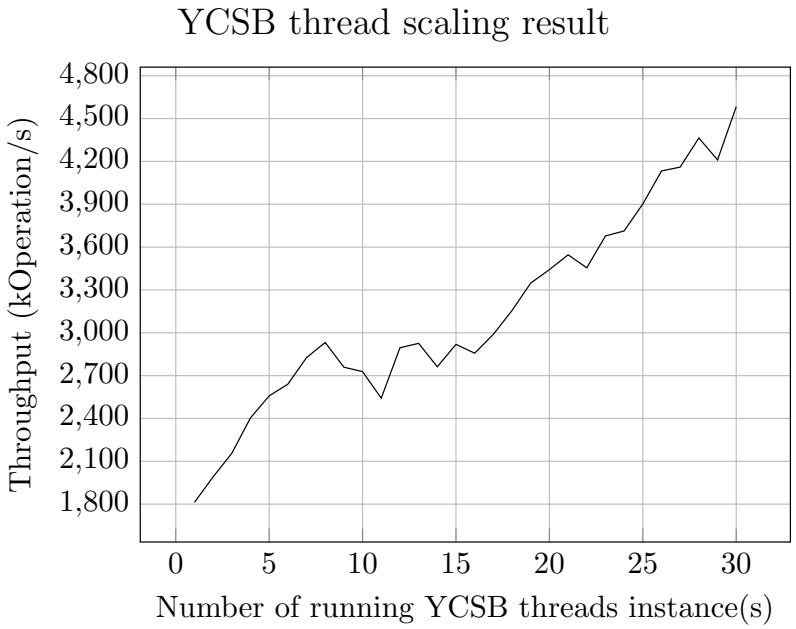


Figure 4.3: Test system's throughput for generating KV pairs without inserting the data into the KVS.

4.3 Non-volatile key-value store (NVKVS)

The main bottleneck of KVSs with KV separation such as WiscKey is the storage device because they write most of the data in the log files in the SSD. Replacing the storage device with a faster solution like NVS seems like a natural progression to these KVS. However, at the moment, no KVS with KV separation addresses the potential issue that may come from the client threads, which can be detrimental to the KVS throughput. In this paper, we propose a new NVM-based LSM-tree with KV separation called non-volatile key-value store (NVKVS). NVKVS decouples the client threads from the writer threads and stores the KV-pairs in the NVM to maximize the write throughputs. It can maximize the performance of NVM devices without degrading the write performance. Additionally, NVKVS can extend the system lifetime by performing most writes to the NVM, which has significantly better endurance than SSD drives with NAND flash.

Figure 4.4 shows the design of NVKVS. NVKVS maintains KV-pairs in a rotating log file stored in NVM to reduce the write latency and increase system endurance in write-intensive workloads. To provide fast reads, NVKVS utilizes RocksDB to index the keys and the offset of the values, which are stored in the NVM. NVKVS manages new writes by inserting key-value pairs into the log file first, and then into the LSM-tree. The read process is backward from the writing process. The read

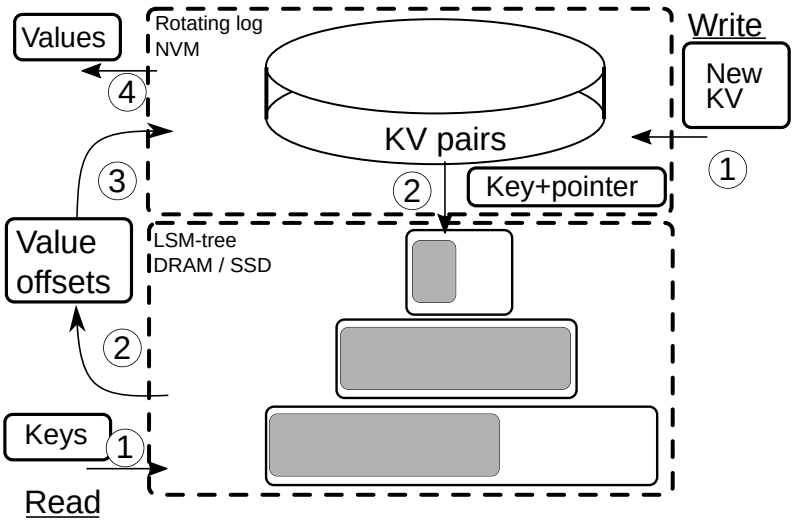


Figure 4.4: The design and supported operations in NVKVS. Write (1) inserts the data into the log file and (2) index it in the LSM-tree. Read (1) starts from the LSM-tree to retrieve (2 and 3) the offset to the value in the log file. (4) NVKVS returns the value in the log based on the offset.

starts by querying the LSM-tree and then reading the values from the log file based on the offsets from the LSM-tree.

4.3.1 NVKVS design challenges

NVM brings potential performance and endurance improvements over SSDs devices. For example, NVM has a significantly higher endurance and a lower access latency when compared to other persistent storage mediums. However, to achieve the maximum throughput of the NVMs, the system's design must consider the unique characteristics of NVM devices [77]. NVKVS uses several programming techniques and optimizations to maximize the performance of NVMs while ensuring atomicity.

Parallel data access

Similar to SSDs, NVMs require multiple threads to access the data to saturate the throughput. This can be solved easily by spawning multiple threads to execute multiple jobs in parallel. However, simply creating new threads without limits will degrade the performance of NVMs because of the limited capability of the NVM's underlying write buffers [77]. For example, when multiple threads try to write data into NVM, the throughput may decline after several parallel accesses, which depends on the number of DIMMs for the interleaved NVM setup.

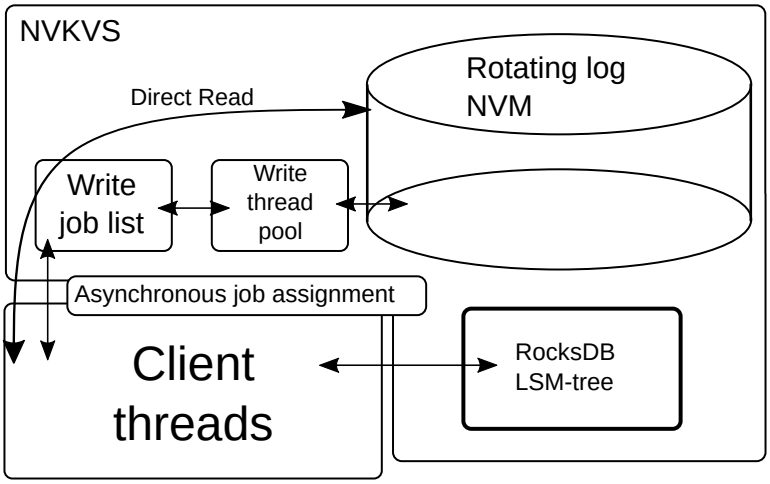


Figure 4.5: NVKVS's components that handle write-operations.

To avoid degrading the performance, NVKVS has a worker pool with a configurable number of threads that decouples the client threads with the NVM to ensure client-side multithreading does not harm the key-value store performance. This number of threads can be configured based on the configuration of the NVMs in their systems. For example, users with more interleaved NVM DIMMs can use a higher number of threads to maximize their system throughput. This system allows the client to submit a new job in a multithreaded manner without significantly harming the performance.

Figure 4.5 shows the multithreading aspect of NVKVS. NVKVS handles data reads from the log file in the NVM separately from the client threads. The client threads first submit the write jobs to the job list and notify the corresponding worker pool. The client threads then wait for the job status to change while the workers in the pool work do the job. After job completion, the worker threads notify the client threads regarding the job completion status. This process occurs asynchronously between the client threads and the NVKVS threads.

Atomicity

WAL is one of the most used techniques in databases to ensure that the durability of each write is atomic. However, in the case of separated KV-pairs databases such as WiscKey and NVKVS, this can be a wasteful process because they already write the KV-pairs to the log files in persistent storage devices such as NVMs and SSDs. Adding an extra log would be wasteful. In this case, NVKVS writes the WAL to

the NVM and reuses it as the log file. When a new KV-pair does not reach the LSM-tree, it is assumed as lost because the read always starts from the LSM-tree and not from the log file.

Write persistency

Because NVM shares some programming models with DRAM, data writes to the NVM are not always persistent because the data might still lie in the cache line of the processor [88]. NVKVS solves this by calling processor instruction called CLWB through libpmem2 persist function [90], which flushes the cache line of the processor to the NVM, to ensure the write but without evicting the data in the cache. This solution guarantees that each inserted KV-pair is in the NVM and does not harm the data access performance because the data is still accessible in the cache.

Data recovery

Unexpected events like system crashes from hardware or software errors may occur in any system. To prevent such events, KVSs like NVKVS must handle data recovery once the system is back online. Because each write in NVKVS is atomic and done in chronological order, NVKVS handles these events in two ways. First, if the LSM-tree is not lost, NVKVS can query the log files from the tail. Second, if the LSM-tree is lost, NVKVS can recover the data by reading the log file from the head and reconstructing the LSM-tree. NVKVS may lose any writes that occur during these events because it only updates the current offset data for completed writes. In such cases, NVKVS never has corrupted writes unless there is an error in the NVM device.

4.4 Implementation

To evaluate the performance of the proposed design, we implemented a prototype of NVKVS by extending the RocksDB 6.15.2. In the code, we modified the main DB class, which is the DBImpl class, to intercept and retrieve the KV-pairs before RocksDB stores them in the LSM-tree. Once NVKVS receives the KV-pairs, NVKVS registers an offset for the KV-pairs and returns the offset to the RocksDB to store in the LSM-tree. The implementation of NVKVS is illustrated in 4.6.

NVKVS currently supports write and simple read to ensure that KV-pairs writes are successful. Each operation, especially for writers, uses thread-safe implementation to ensure atomicity for the database, such that no writes can overlap each other.

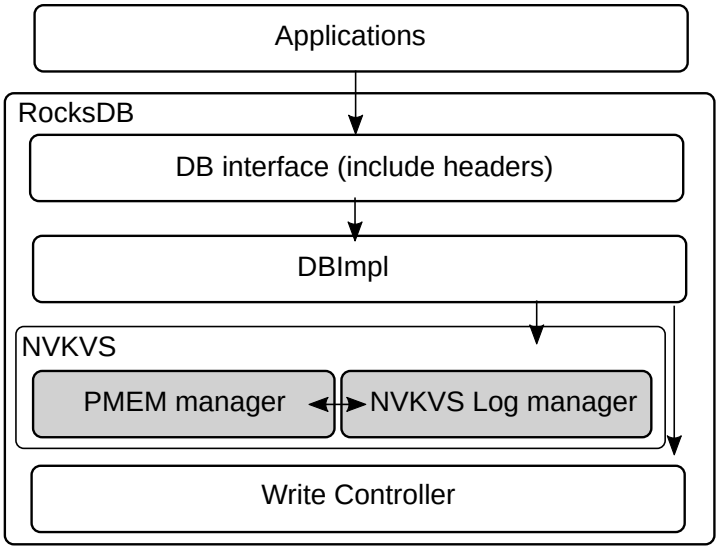


Figure 4.6: The implementation of NVKVS in RocksDB 6.15.2. The grayed part is the NVKVS specific components.

Once RocksDB indexes the KV-pairs, the client can access the KV-pairs through the RocksDB.

4.4.1 PMEM manager

To manage the writes and offsets in NVKVS, we included an NVM manager called PMEM manager as shown in 4.6. On startup, the PMEM manager loads the offset from the header of the log file, which contains the start offset, current write offset, current garbage collector offset, and the max offset. Additionally, it will try to get the NVM base offset, which is randomized by the operating system's address space layout randomization, by using the lpmem2 library. Based on these offsets, the PMEM manager can guide the writer threads in writing the data to the NVM by adding the current write offset to the base offset. It also provides persist function to the writer threads to ensure the atomicity of each write.

4.4.2 Log file

To store the KV-pairs, NVKVS uses a rotating log, which is a log file that is reusable when full by rewriting from the beginning. Inspired by WiscKey, NVKVS checks the validity of the KV-pairs before rewriting them. If the pairs are still valid in the LSM-tree, NVKVS reinserts the KV-pairs back to the log file. This solution may increase the write amplification of the log file. However, write amplification is less of a concern for NVM, which is more durable than SSDs.

Figure 4.7 illustrates the structure of the rotating log file. The log file contains two important structures, the header of the log file and the header of the KV-pair. The header of the log file contains the offset data, which are the maximum length of the log file, the last write offset, and the last garbage collector offset. NVKVS updates these offset information for every KV-pair write and when NVKVS closes the log file. The header of the KV pair contains the length for the trailing key and value. In this aspect, NVKVS is similar to WiscKey, which can store variable-sized values without any padding. NVKVS uses unsigned short, which is 16 bits in length, for the key and value. Based on this specification, the maximum length for a key or a value would be 64 kB, which is sufficient for most workloads that may have values varying from 10 B to 4 kB [76]. This limit can be changed to accommodate larger values by increasing the size of the header. For larger values, the cost of the larger header is insignificant compared to the value size.

4.4.3 WiscKey reimplementation

At the time of writing this paper, we do not have any access to the source code of WiscKey. Because of this issue, some parts of the implementation might be different from the original work. For example, we did not implement the write buffers for the value log because of our time limitation. However, our reimplementation retains the key features of WiscKey such as KV separation and log file in the SSD. Additionally, the reimplementation uses the same code as NVKVS to evaluate the benefit of NVM over SSD for KVS with KV separation.

4.5 Experimental results

In this section, we discuss the performance aspect of NVKVS for write workloads based on our experimental testing.

4.5.1 Test setups

To compare NVKVS in write workloads with vanilla RocksDB and our reimplementation of WiscKey, we conducted database load experiments where each tested KVS runs on the same test system described in Section 4.2. NVKVS uses the SSD to store the LSM-tree and the NVM to manage the log files. Vanilla RocksDB and WiscKey have two versions, which are SSD and NVM. In the test results, the storage medium is indicated in the name. Both NVKVS and WiscKey have their RocksDB's WAL disabled as explained in Section 4.3. All tests are run 3 times and the shown re-

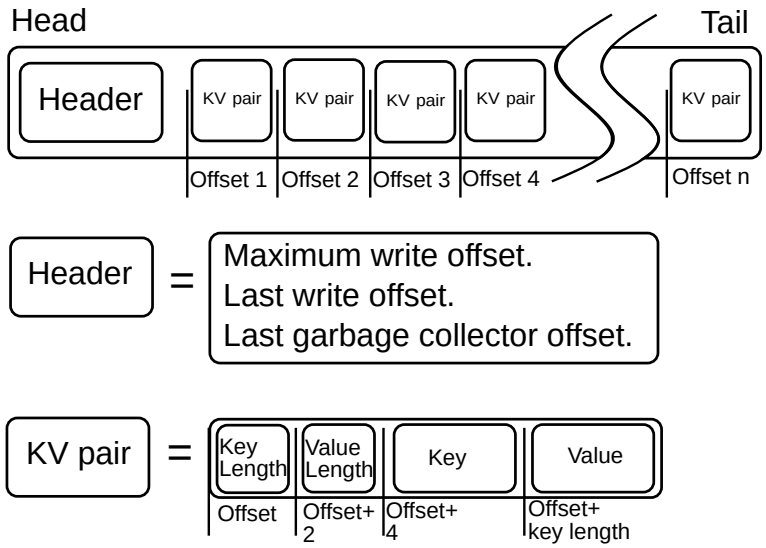


Figure 4.7: The structure of the log file in the persistent memory. The header of the log file contains offset information. Each KV-pair has length information for both the key and value.

sults are the average of the 3 runs' results. All tested KVSs use the same RocksDB configuration with compression disabled except for vanilla RocksDB with WAL and pipelined write enabled.

To test these KVSs, we use a popular database benchmark application called YCSB [93]. The default YCSB also tests the test system by performing serialization and deserialization. Additionally, it uses column family to store the values into the LSM-tree. These operations are compute-intensive, expensive, and may misrepresent the performance of the database under plain key-value insertion. To test the maximum potential of each KVS, we remove these extra operations for YCSB, which is similar to what is done in SpanDB [82].

Decoupled write experiment

One of the main issues with existing KVS with KV separation is that the client threads and the writer threads are coupled together, which can be detrimental for NVM. In this test, we compared NVKVS with coupled threads and NVKVS with decoupled threads. The NVKVS with decoupled threads uses 6 threads for the writer threads in the worker pool. We used YCSB and inserted 20 million keys with a 1000 bytes value each into the KVS with YCSB threads from 15 to 25 threads.

As shown in Figure 4.8, NVKVS with decoupled threads scales better with the

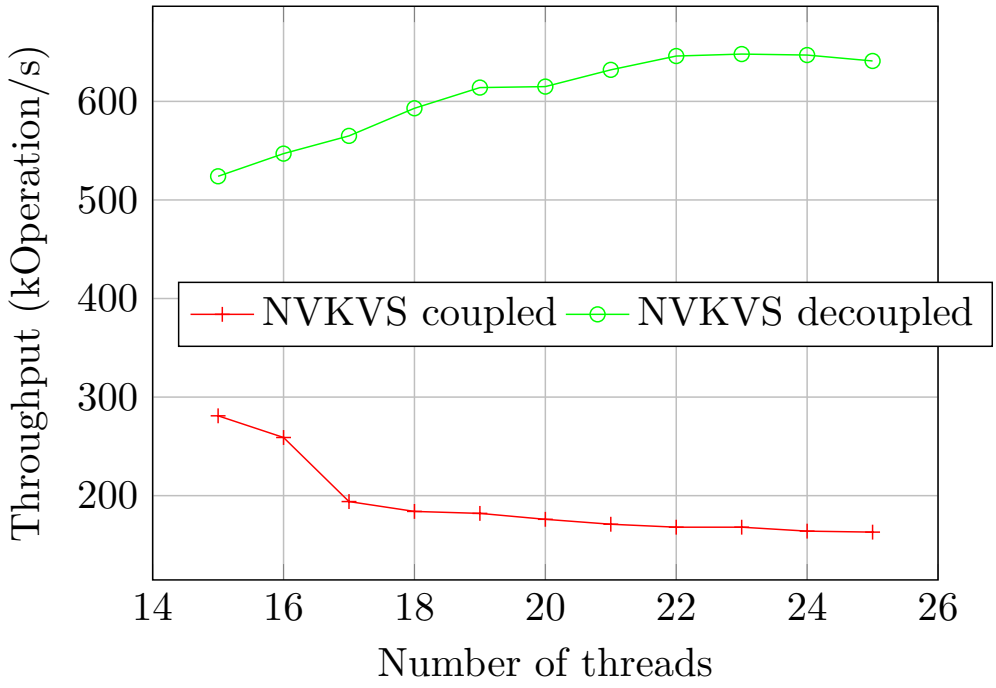


Figure 4.8: The comparison of NVKVS with coupled threads and NVKVS with decoupled threads. NVKVS with decoupled threads uses 6 writer threads. The higher is better.

number of client threads when NVKVS limits the number of writer threads through decoupling. When NVKVS does not separate the client's and writer's threads, the performance degrades more with a higher number of threads. This result indicates that simply replacing the SSD with NVM in WiscKey will degrade the performance because of the lack of writer threads decoupling.

Worker thread scaling

In this experiment, we tested the worker threads scaling for the worker pool in NVKVS and WiscKey with 20 million keys and 1000 bytes values. As shown in Table 4.2, NVKVS scales well with the number of YCSB threads because of the worker pool. Without the worker pool, the client's threads may overstress the write buffer, degrading the write throughput. However, when the YCSB thread's number is too low, the performance degrades because it is not enough to saturate the NVM. Additionally, when the number of YCSB threads is too high, the performance starts to degrade because of the overhead of each thread. These values may vary depending on the NVM configuration. With our NVM configuration, we saturated the performance at 23 YCSB worker threads and 6 threads in NVKVS's worker pool with 648 kOps/s.

Table 4.2: The throughput for thread number scaling test of NVKVS. The higher is better.

Number of threads in NVKVS's worker pool. \ Number of YCSB worker.	15	16	17	18	19	20	21	22	23	24	25
1	519	509	515	514	511	507	501	497	496	485	485
2	589	594	586	576	584	571	571	566	565	566	560
3	614	614	617	606	615	607	609	596	604	597	596
4	603	606	620	631	620	631	627	624	623	621	612
5	558	583	604	620	627	639	631	637	636	635	629
6	524	547	565	593	614	615	632	646	648	647	641
7	491	514	537	558	579	593	615	633	646	643	583
8	463	489	506	532	550	575	594	610	619	577	494
9	438	461	485	504	527	550	568	584	553	478	440
10	413	435	457	479	504	521	540	522	463	434	400

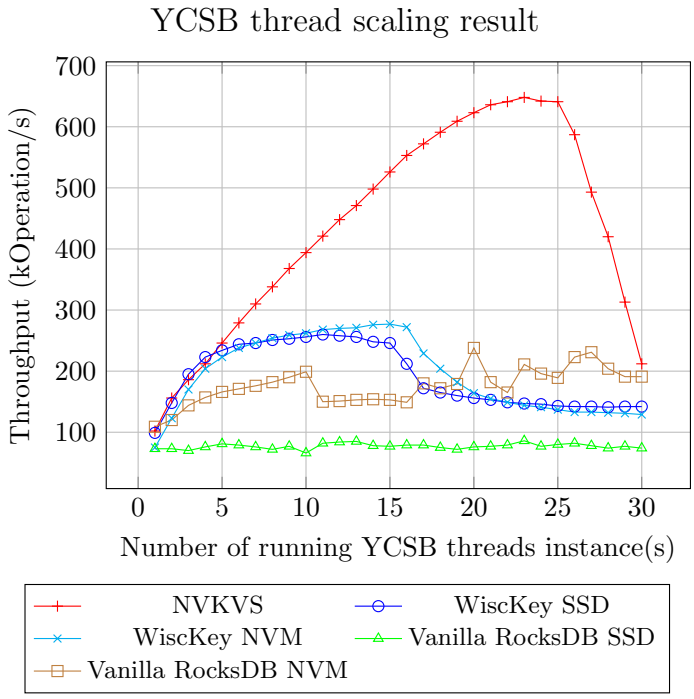


Figure 4.9: YCSB worker thread scaling for NVKVS, WiscKey, and vanilla RocksDB. The higher is better.

Next, to measure the best number of threads for YCSB for the tested KVSs, we used a fixed number of threads for the worker pools in NVKVS and WiscKey and changed the number of worker threads for YCSB. The test uses 20 million keys and a 1000 bytes value each. In this test, we use 6 threads for NVKVS's write threads. These numbers are obtained based on our experiment, where we varied the number of threads for YCSB and the KVSs in the worker pool to get the maximum throughput.

Figure 4.9 shows the result for YCSB scaling for the tested KVSs with a fixed number of worker pools for NVKVS. From the result, we observed that NVKVS performed well up to 25 YCSB threads with only 6 threads in the write thread pool. This result indicates that more direct solutions such as limiting the number of client threads to limit the number of access to the NVM would not saturate the system because the client cannot generate enough data. For example, if we restrict the number of client threads to 6, NVKVS would only achieve around 285 kOps in this test. With the asynchronous multithreading, NVKVS allows the application to generate more data by increasing the thread count without affecting the write throughput. However, over 25 threads, the overhead of the multithreading starts to degrade the performance of the KVS.

The results depicted in Figure 4.9 also indicate that the performance may change when using Vanilla RocksDB and WiscKey on NVM. For Vanilla RocksDB, the throughput increases by over 2-times at around 238 kOps for NVM and 76 kOps for SSD. However, we observed throughput inconsistencies with 10-threads or more. WiscKey on the other hand, performed more consistently than Vanilla RocksDB because it performs fewer merge operations. However, switching to NVM from SSD only slightly improved the performance of WiscKey.

Additionally, the results in Figure 4.9 show that WiscKey only scales well up to 5 threads. From 6 threads to 15 threads, we observed a small increase in the throughput for WiscKey. After 15 threads, the throughput of WiscKey started to degrade because of the parallel write limitation of the SSD and the multithreading overhead. We also observed that RocksDB does not scale well with over a single YCSB thread because it must write the data into the WAL and the extra overhead from threads synchronization [82].

YCSB load

One of the main usages of KVSs is to manage a live databases system [94, 95] where the KV-pairs are inserted individually without any type of batching from the client. This type of insertion is more sensitive to the overhead of each write unlike batched insertions, which amortizes the overhead through bulk writes. In

Table 4.3: The average, 99th percentile, and maximum latency results for the tested K-V stores for a short load (20 million keys). The values are 1000 bytes in size. The lower is better.

	Short load		
	Average (us)	99th percentile (us)	Max (us)
NVKVS	30	59	9843
WiscKey SSD	44	92	74986
WiscKey NVM	51	148	7275
V. RocksDB SSD	123	22	6671018
V. RocksDB NVM	79	364	16542

Table 4.4: The average, 99th percentile, and maximum latency results for the tested K-V stores for a long load (100 million keys). The values are 1000 bytes in size. The lower is better.

	Long load		
	Average (us)	99th percentile (us)	Max (us)
NVKVS	31	59	18671
WiscKey SSD	38	113	123391
WiscKey NVM	51	149	13162
V. RocksDB SSD	31	4552	15103316
V. RocksDB NVM	147	2536	56607

this test, we evaluated these three KVSs by using YCSB workload A load, which inserts the KV-pairs separately from each other with RocksDB's put command, with two different database sizes. The first one contains 20 million keys with a 1000 bytes value each. The second one has 100 million keys with a 1000 bytes value each. We used the best numbers of YCSB threads from the previous subsection, which are: 6 threads in the worker pool and 23 threads for NVKVS, 11 threads for WiscKey on SSD, 15 threads for WiscKey on NVM, 23 threads for vanilla RocksDB on SSD, and 20 threads for vanilla RocksDB on NVM.

As shown in Table 4.3 and 4.4, for short and long loads, NVKVS performed better than other tested KVSs for average and maximum latencies because of the characteristics of NVM. As for WiscKey, it also performed consistently because of the KV separation. However, the maximum latency becomes more significant because compaction in the LSM-tree is more often with long writes. The 99th-percentile latency of vanilla RocksDB for short load is better than WiscKey's because these KVS with KV separation have extra overhead for writing into the log files. However, the average latency is worse for vanilla RocksDB because the max latency

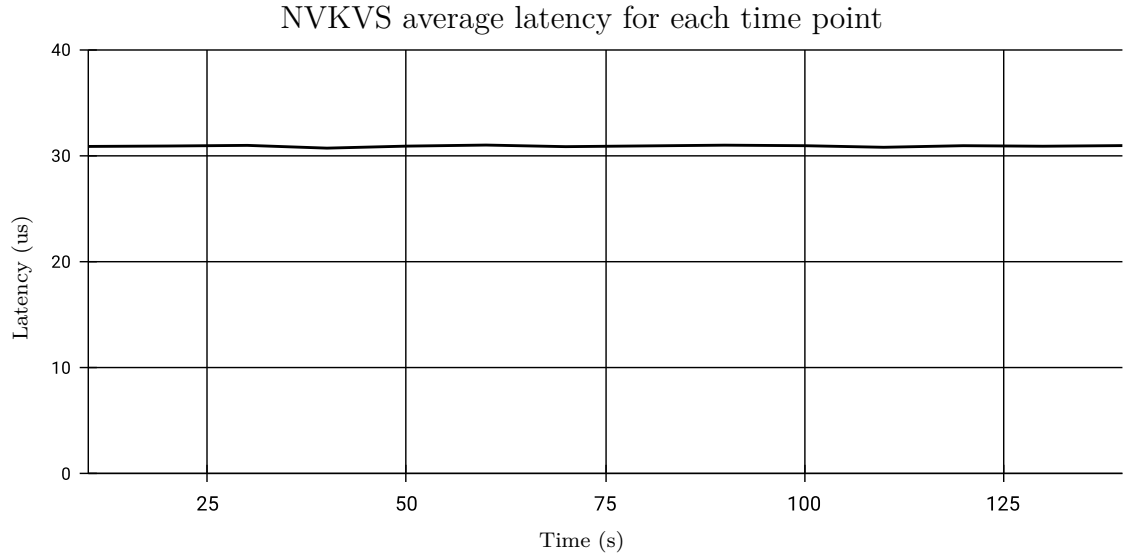


Figure 4.10: The average latency for each 10 seconds time point during 100 million keys upload by using YCSB load for NVKVS.

is significantly worse.

The results also indicate that for long load, vanilla RocksDB performed poorly because compaction is more often for larger LSM-trees. This reason is also supported by the results in Figure 4.10, 4.11, 4.12, 4.13, and 4.14, , which illustrates the average latencies for 10s durations during the 100 million keys YCSB load test. The smaller LSM-tree in NVKVS and WiscKey is more advantageous in terms of latency because they can deliver consistent latencies throughout the write workload. The results also indicate that NVM can provide more consistent latency for Vanilla RocksDB. However, WiscKey with NVM regressed in latency because it does not decouple the client threads from the writer threads, which is detrimental for NVM.

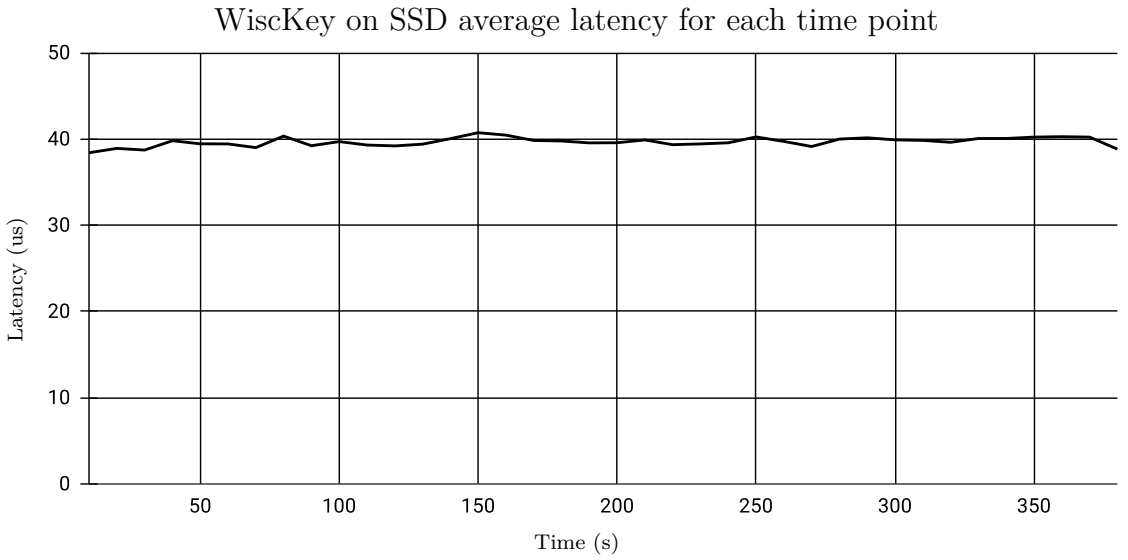


Figure 4.11: Average latency for each 10 seconds time point during 100 million keys upload by using YCSB load for WiscKey on SSD.

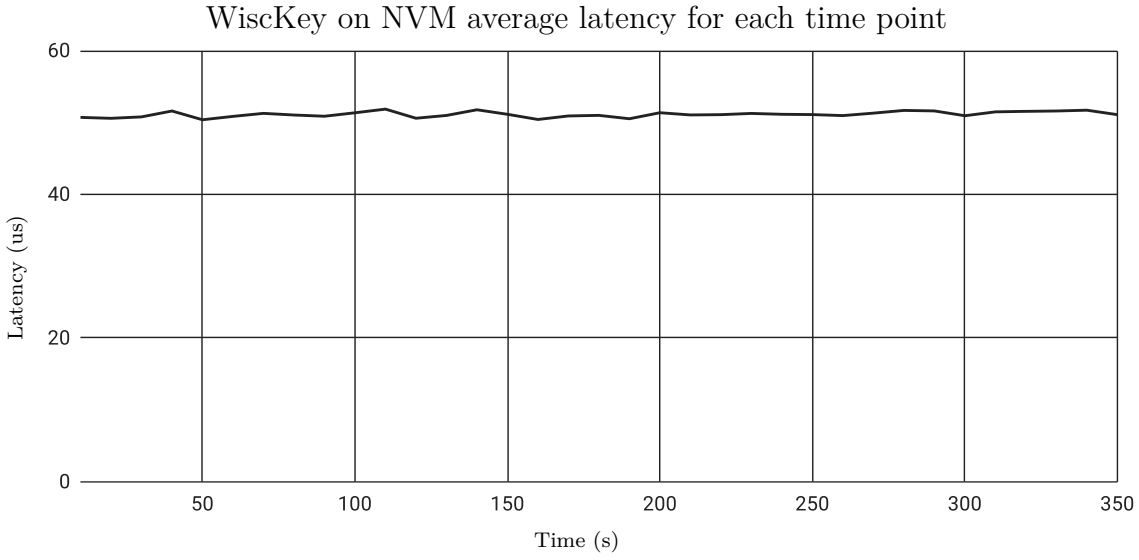


Figure 4.12: Average latency for each 10 seconds time point during 100 million keys upload by using YCSB load for WiscKey on NVM.

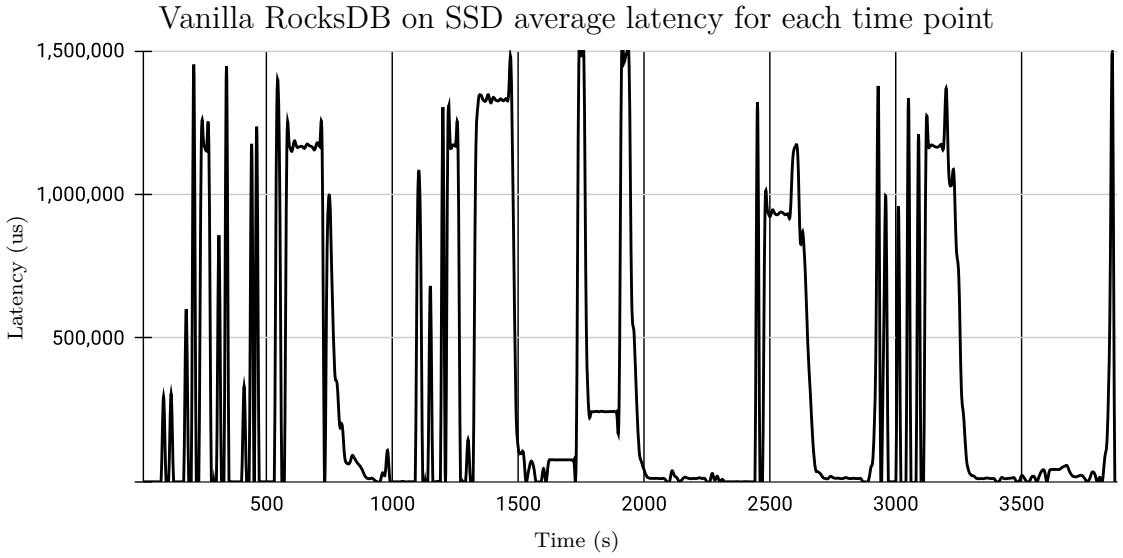


Figure 4.13: Average latency for each 10 seconds time point during 100 million keys upload by using YCSB load for Vanilla RocksDB on SSD.

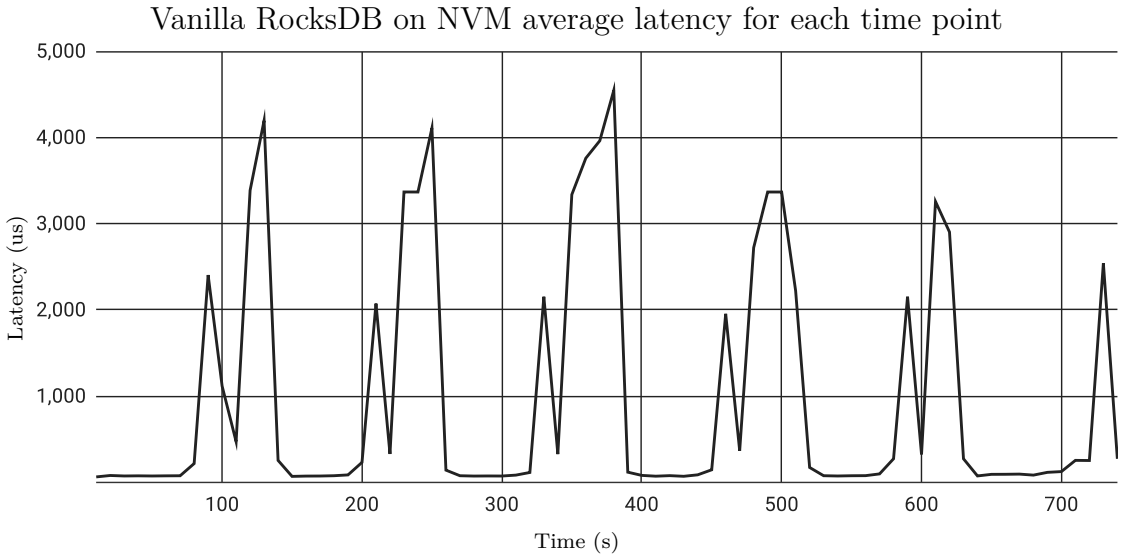


Figure 4.14: Average latency for each 10 seconds time point during 100 million keys upload by using YCSB load for Vanilla RocksDB on NVM.

As shown in Figure 4.15, with short writes of 20 million keys NVKVS outperforms other tested KVSs with 648 kOps/s, which is over double the throughput of our WiscKey reimplementation. During short writes, the throughput for all tested KVSs is stable because the burden of compaction is not significant. With long writes of 100 million keys, which has an approximate size of 100 GB, NVKVS performs better with long writes because the startup overhead becomes more apparent with the short

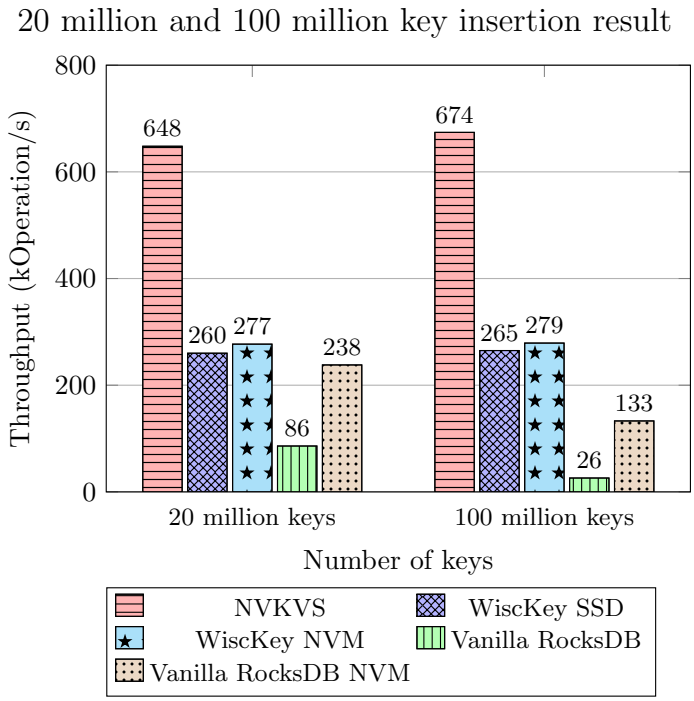


Figure 4.15: Write throughput for short write (20 million keys) and long write (100 million keys) with 1000 bytes values. The higher is better.

writes of 20 million keys. Similar to the latency, NVKVS and WiscKey do not suffer from long write sessions because of the smaller LSM-tree, unlike vanilla RocksDB.

4.5.2 Data recovery

Unwanted events like unexpected shutdowns from hardware failures or software failures may cause data loss for KVS systems like NVKVS. In NVKVS, data loss may occur in two ways: (1) partial loss from the current writes where NVKVS have not persisted some keys from the client in the log file, and (2) index loss, where the data is already in the log file, but not in the LSM-tree. NVKVS solves the former by reading the header and checking the written data in the log file and inserting it back into the LSM-tree. As for the latter, NVKVS simply reinserts keys in the log file into the LSM-tree starting from the last-known key in the LSM-tree.

We tested the indexing speed by indexing a 100 GB log file. NVKVS completed this indexing process in 202 seconds on a single thread, which translates to 495 MBps. This indexing process is slower than the YCSB load in the previous subsection because NVKVS indexes the keys in chronological order, which limits the number of threads to 1.

4.5.3 Write amplification

In terms of write amplification, we recorded 3 for both NVKVS and WiscKey and 7.4 for vanilla RocksDB from RocksDB stats. Vanilla RocksDB has a higher number of write amplification because the LSM-tree is significantly larger than other tested KVSs, thus increasing the occurrence of compaction. As for NVKVS and WiscKey, the write amplification number is lower and limited to the LSM-tree, which only takes 3% of the dataset's size. As for the log file, which is significantly larger than the LSM-tree, the write amplification is 1 because no garbage collection scheme is currently implemented. In the case of NVKVS, write amplification for the log file is less of a concern because NVM is more durable than SSD. In the future, we are planning to test other operations such as point query, range query, and garbage collection for NVKVS.

4.6 Related work

WiscKey [76] is the first work that uses KV separation in the LSM-tree. It introduces KV separation into the LSM-tree by chronologically writing the KV-pairs in log files and indexes the keys in the LSM-tree. The implementation of WiscKey uses LevelDB for the LSM-tree. The main drawback of WiscKey is the lack of write thread management, which is crucial for storage technologies like Optane NVM from Intel. These type of storage technologies has a limited write buffer [77], which may degrade the write throughput when overwhelmed by write-operations from a large number of threads.

One of the main concerns for the existing KVSs with LSM-tree is write-thread synchronization, which can take a significant amount of time when using high-performance storage devices. SpanDB [82] extends the existing RocksDB's code to accommodate low latencies writes when using high-performance NVMe SSDs. SpanDB replaces the synchronized write method in the RocksDB with asynchronous write-operations where the client can submit the KV-pairs and wait. SpanDB utilizes batching and parallel-writes methods to maximize the throughput of the storage devices.

NoveLSM [96] is an LSM-tree-based KVS system that is designed for NVM to achieve low latency and high throughput in applications. The main feature of NoveLSM is the use of a byte-addressable data structure to store the data in the NVM called persistent skiplist. This approach avoids the block I/O, minimizing the latency and data recovery time.

The work presented in this paper compares directly to WiscKey because it shares

the idea of the KV separation from WiscKey and extends it for NVM-based storage devices. This paper proposes a write decoupling method for LSM-tree with KV separation like WiscKey to exploit the high throughput and low latency of NVM. Our experimental results show that decoupling the write is crucial to maintain high throughput with a high number of client threads during write-intensive workloads.

4.7 Conclusion of this chapter

LSM-trees with KV separation such as WiscKey solve the issue with high write tail latency for LSM-tree especially for workloads with large datasets. However, the current design of WiscKey is only optimized for SSD by using in-memory write buffers, which does not guarantee atomicity and might run out during write-intensive workloads. Additionally, no mechanism limits the write to the SSD. Writes from a high number of client threads may exceed the buffer capacity and flush speed, degrading the system performance. Simply replacing the SSD with faster storage devices like NVM may degrade the performance of the KVS because of the unique characteristics of the NVM.

In this work, we propose an NVM-based KVS with KV separation that indexes the keys through LSM-tree called NVKVS. It features asynchronous multithreading that decouples the client and writer threads to maximize the storage device throughput, and reusable WAL to maximize the performance of NVM devices without sacrificing the atomicity of each write. Based on our experimental testings, NVKVS can offer over double the throughput and better write latencies when compared to other tested KVSs.

Chapter 5

Conclusion

5.1 Summary

Data reduction schemes play an important role in modern storage technologies including distributed file systems (DFSs), which is a scalable solution to store a massive amount of data. To improve the performance and ease of use of these schemes mainly in DFS environments, this study explores three important aspects to the application of data reduction schemes in DFSs, which are the data reduction schemes, the distributed file systems (DFSs), and the metadata system that supports many applications including the data reduction schemes and DFSs. First, it summarizes the challenges of chunking in deduplication and how it affects the performance of the deduplication scheme. Second, it reviews the challenges of applying data reduction schemes in a DFS environment, which may hinder the use of the schemes. Third, it discusses the issues of log-structured merge-tree (LSM-tree) based key-value stores (KVSs) when used with a non-volatile memory (NVM).

The Byte-values-based content-dependent chunking (CDC) algorithms' mainly use byte-value comparisons to determine the cut-point. Reducing the number of comparisons can directly translate to higher throughput. The proposed work called Rapid Asymmetric Maximum (RAM) improves upon the state-of-the-art algorithm called Asymmetric Extremum (AE) [14] by changing the conditions and window configuration. This new condition reduces the possibility of entering the condition and the overall number of comparisons. Additionally, RAM performs fewer variable assignments because the variable assignments to track the maximum-sized byte are located in the first if statement. However, RAM's window configuration is worse at detecting low entropy string, which can result in long chunks and lower duplicate finding performance. To solve this issue, this work also proposes RAML, which uses a size limit that slightly increases the number of comparisons. Compared to AE,

RAM can produce up to 48% higher throughput at the cost of 2% to 16% lower duplicate found.

Adding and enabling data reduction schemes in a DFS environment can be a challenge depending on the applications and the DFS environment, which often is inaccessible to the users. This work proposes a new DFS design that enables easy data reduction schemes to use and implement in the DFS through programming techniques such as the dynamic library. It evaluates the feasibility of the design by implementing it as a framework called Hadoop Data Reduction Framework (HDRF) in a DFS called Hadoop DFS (HDFS). HDRF enables file system-level reduction in HDFS without relying on the underlying file system's reduction scheme, making it compatible with all Hadoop applications. The experimental results show that HDRF can speed up data transfer when using a data transfer optimized application without affecting the compression ratio. The results also indicate that HDRF has minimal runtime and storage overhead compared to vanilla HDFS when running the same application.

LSM-trees with KV separation such as WiscKey solve the issue with high write tail latency for LSM-tree by writing the key-value pairs (KV-pairs) into a log file and indexing them in the LSM-tree. This separation reduces the LSM-tree's size and the number of merge processes, increasing the consistency of write latencies for write-intensive workloads. However, these KVS with KV separation does not limit the number of threads that access the storage devices, which can be detrimental for storage devices that do not scale well with a high number of threads such as NVM. This work proposes a KVS called Non-volatile KVS that controls the number of access to the NVM by using asynchronous multithreading. This solution limits the number of write-threads without limiting the number of client threads, which is crucial to generate the data. This study evaluates the proposed approach through the implementation of the design called NVKVS in RocksDB. The experimental results show that NVKVS can perform up to 2-times better than the reimplementation of WiscKey for write-intensive workloads with a high number of client threads.

The combination of the three works proposed in this study are not tied to each other and can work on top of each other to improve the DFS's performance and ease of use. For example, the metadata system can run in both the DFS and the deduplication scheme with the proposed CDC algorithm that runs in the DFS. This study can improve the existing DFS design and inspire future DFS design to be more easily accessible in the aspect of the data reduction scheme applications.

5.2 Future work

This study explores three different aspects of data reduction schemes usage in DFS environments and proposes solutions to improve the DFS in the terms of performance and ease of use when performing data reduction in the DFS. As a whole study, it can be expanded in several aspects. For example, evaluation of the DFS design in different DFS software such as Lustre to confirm its effectiveness.

The work presented in this study is also expandable as an independent study. For example, the use of RAM in performance limited environments such as deduplication in IoT devices. For HDRF, it has other potentials such as the data processing capability of HDRF, which can improve the compatibility of HDFS with other storage devices without relying on other solutions with high processing overhead such as FUSE [97]. For NVKVS, it still requires a garbage collection scheme to address the limited capacity of the NVM and to minimize the write amplification on the NVM by writing the data in an efficient data structure on more cost-effective storage mediums such as SSD.

Acknowledgement

It has been a pleasure for me to express my sincere gratitude to all people who have come along the way accompanying me during my study at the University of Tsukuba. I cannot complete this thesis by myself without their help. With their help, I was able to gain experiences that might have not been possible to achieve alone. I thank each of them.

First and foremost, I would like to express my gratitude to my advisors, Professor Kazuhiko Kato and Professor Hirotake Abe for allowing me to join the Operating Systems and System Software (OSSS laboratory). Professor Kazuhiko Kato and Professor Hirotake Abe also guide me throughout my study and provided the tools required for my research. Additionally, I also thank you for the relentless encouragement during my research and job hunting.

I would like to extend my gratitude to Professor Osamu Tatebe, Professor Toshiyuki Amagasa, and Professor Kazumasa Omote for providing me with useful comments and critics on the thesis. I was able to improve the thesis to the current state because of their comments.

Additionally, I would like to thank the co-authors, editors, and reviewers who are involved in my published articles. I feel blessed for receiving their encouragement and comments to improve the research. Their comments provide new points of view on projects and clues on how to progress with my research.

I thank all students and alumni of the OSSS laboratory for their comments during the seminar and in the laboratory. I was able to learn different research topics and take inspiration from their research.

Additionally, I thank The Ministry of Education, Culture, Sports, Science, and Technology (MEXT) for funding my study through their scholarship program. Because of the scholarship, I was able to focus on my research and study without any financial issues.

I also extend my gratitude to researchers and developers for the articles and open source software used in this study. In combination with the communities on the internet, I was able to use the software and experiment with new ideas to progress with my research. Without their contribution, this study might not exist.

I am also thankful to my family in Indonesia for their endless support in my entire life. Throughout my study, they have supported me mentally. Without them, I would not have become someone I would be proud of.

Last but not least, I would like to thank my Lord and Savior, Jesus Christ for His plan and grace to me. All my plans and hard work would be useless without His grace and help.

Bibliography

[1] Matthew A Russell. *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More.* " O'Reilly Media, Inc.", 2013.

[2] Paolo Giudici and Gianluca Passerone. Data mining of association structures to model consumer behaviour. *Computational Statistics & Data Analysis*, 38(4):533–541, 2002.

[3] Qin Yao, Yu Tian, Peng-Fei Li, Li-Li Tian, Yang-Ming Qian, and Jing-Song Li. Design and development of a medical big data processing system based on hadoop. *Journal of medical systems*, 39(3):23, 2015.

[4] Fernando Andreotti, Joachim Behar, Sebastian Zaunseder, Julien Oster, and Gari D Clifford. An open-source framework for stress-testing non-invasive foetal ECG extraction algorithms. *Physiological Measurement*, 37(5):627–648, apr 2016.

[5] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2003.

[6] U Cisco. Cisco annual internet report (2018–2023) white paper, 2020, Online, https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, (Accessed: 2021-12-15).

[7] Arne Holst. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025, 2021, Online, https://www.statista.com/statistics/871513/worldwide-data-created/, (Accessed: 2021-12-15).

[8] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. Logzip: Extracting hidden structures via iterative clustering for log compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 863–873. IEEE, 2019.

[9] 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.

[10] Dongzhan Zhang, Chengfa Liao, Wenjing Yan, Ran Tao, and Wei Zheng. Data deduplication based on hadoop. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 147–152, 08 2017.

[11] Dhruba Borthakur et al. HDFS architecture guide. *Hadoop Apache Project*, 53(1-13):2, 2008.

[12] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2010.

[13] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[14] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. Ae: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, 2015.

[15] 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)*, pages 101–114, Denver, CO, June 2016. USENIX Association.

[16] Nikolaj Bjørner, Andreas Blass, and Yuri Gurevich. Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences*, 76(3):154–203, 2010.

[17] Ryan N.S. Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. A new content-defined chunking algorithm for data deduplication in cloud storage. *Future Generation Computer Systems*, 71:145–156, 2017.

[18] Ryan NS Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. Sdm: Smart deduplication for mobile cloud storage. *Future Generation Computer Systems*, 70:64–73, 2017.

[19] Chao Yang, Jian Ren, and Jianfeng Ma. Provable ownership of files in deduplication cloud storage. *Security and Communication Networks*, 8(14):2457–2468, 2015.

[20] Keren Jin and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, 2009.

[21] Sung-hun Kim, Jinkyu Jeong, and Joonwon Lee. Selective memory deduplication for cost efficiency in mobile smart devices. *IEEE Transactions on Consumer Electronics*, 60(2):276–284, 2014.

[22] Jinwoo Ahn and Dongkun Shin. Optimizing power consumption of memory deduplication scheme. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, pages 1–2. IEEE, 2014.

[23] Anat Bremler-Barr, Shimrit Tzur David, Yotam Harchol, and David Hay. Leveraging traffic repetitions for high-speed deep packet inspection. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2578–2586. IEEE, 2015.

[24] MO Rabin. Fingerprinting by random polynomials. no. Technical report, TR-1581, 1981.

[25] H Khuern. A framework for analyzing and improving content-based chunking algorithms. *Intell. Enterp.*, 2005.

[26] USMA West Point. Data capture from national security agency (nsa), west point united states military academy, 2009., 2009, Online, http://www.westpoint.edu/crc/SitePages/DataSets.aspx, (Accessed: 2016-08-16).

[27] Levente Polyak. Wikimedia downloads: Xml dump for english wikipedia, 2016, Online, https://www.archlinux.org/download/, (Accessed: 2016-09-08).

[28] Google. Chromium os, google, 2016, Online, https://www.chromium.org/chromium-os, (Accessed: 2016-09-08).

[29] SPI Inc. Debian, 2016, Online, https://www.debian.org/CD/torrent-cd/, (Accessed: 2016-09-08).

[30] Inc elementary. elementary os, 2016, Online, https://elementary.io/, (Accessed: 2016-09-08).

[31] Clément Lefèbvre, Jamie B. Birse, Kendall Weaver, and community. Linux mint, 2016, Online, https://www.linuxmint.com/edition.php?id=217, (Accessed: 2016-09-08).

[32] Lubuntu Community. Lubuntu, 2016, Online, http://lubuntu.net/, (Accessed: 2016-09-08).

[33] Solus Project. Solus project, 2016, Online, https://solus-project.com/download/, (Accessed: 2016-09-08).

[34] Canonical Ltd. Ubuntu, 2016, Online, http://www.ubuntu.com/download/desktop, (Accessed: 2016-09-08).

[35] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, 2009.

[36] Yinjin Fu, Hong Jiang, 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 *2011 IEEE International Conference on Cluster Computing*, pages 112–120. IEEE, 2011.

[37] Kritwara Rattanaopas and Sureerat Kaewkeeree. Improving hadoop mapreduce performance with data compression: A study using wordcount job. In *2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 564–567. IEEE, 2017.

[38] Ryan Nathanael Soenjoto Widodo, Hirotake Abe, and Kazuhiko Kato. Hdrf: Hadoop data reduction framework for hadoop distributed file system. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 122–129, 2020.

[39] Bogdan Nicolae. High throughput data-compression for cloud storage. In *International Conference on Data Management in Grid and P2P Systems*, pages 1–12. Springer, 2010.

[40] Chenxi Tu, Eijiro Takeuchi, Chiyomi Miyajima, and Kazuya Takeda. Continuous point cloud data compression using slam based prediction. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1744–1751. IEEE, 2017.

[41] Ravneet Kaur, Inderveer Chana, and Jhilik Bhattacharya. Data deduplication techniques for efficient cloud storage management: a systematic review. *The Journal of Supercomputing*, 74(5):2035–2085, 2018.

[42] Kaium Hossain, Mizanur Rahman, and Shanto Roy. Iot data compression and optimization techniques in cloud storage: Current prospects and future directions. *International Journal of Cloud Applications and Computing (IJCAC)*, 9(2):43–59, 2019.

[43] Naga Malleswari Tyj and G Vadivu. Adaptive deduplication of virtual machine images using akka stream to accelerate live migration process in cloud environment. *Journal of Cloud Computing*, 8(1):1–12, 2019.

[44] Chuan Lin, Qiang Cao, Jianzhong Huang, Jie Yao, Xiaoqian Li, and Changsheng Xie. Hpdv: A highly parallel deduplication cluster for virtual machine images. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 472–481. IEEE, 2018.

[45] Jiwei Xu, Wenbo Zhang, Zhenyu Zhang, Tao Wang, and Tao Huang. Clustering-based acceleration for virtual machine image deduplication in the cloud environment. *Journal of Systems and Software*, 121:144–156, 2016.

[46] Muhammad Usama and Nordin Zakaria. Chaos-based simultaneous compression and encryption for hadoop. *PloS one*, 12(1):e0168207, 2017.

[47] A Ashu, Mir Wajahat Hussain, Diptendu Sinha Roy, and Hemant Kumar Reddy. Intelligent data compression policy for hadoop performance optimization. In *International Conference on Soft Computing and Pattern Recognition*, pages 80–89. Springer, New York City, USA, 2019.

[48] Michael Kuhn, Julian M Kunkel, and Thomas Ludwig. Data compression for climate data. *Supercomputing frontiers and innovations*, 3(1):75–94, 2016.

[49] Sergio De Agostino. The greedy approach to dictionary-based static text compression on a distributed system. *Journal of Discrete Algorithms*, 34:54–61, 2015.

[50] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 74–85. IEEE, 2005.

[51] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. Memzip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 638–649. IEEE, 2014.

[52] Mikhail Zarubin, Patrick Damme, Thomas Kissinger, Dirk Habich, Wolfgang Lehner, and Thomas Willhalm. Integer compression in nvram-centric data stores — comparative experimental analysis to dram. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–11, 2019.

[53] Google. Google snappy, 2019, Online, https://github.com/google/snappy, (Accessed: 2019-10-25).

[54] Ryan NS Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. A new content-defined chunking algorithm for data deduplication in cloud storage. *Future Generation Computer Systems*, 71:145–156, 2017.

[55] Keren Jin and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 7. ACM, 2009.

[56] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. A survey of secure data deduplication schemes for cloud storage systems. *ACM computing surveys (CSUR)*, 49(4):1–38, 2017.

[57] S Ranjitha, P Sudhakar, and KS Seetharaman. A novel and efficient deduplication system for hdfs. *Procedia Computer Science*, 92:498–505, 2016.

[58] Zhe Sun, Jun Shen, and Jianming Yong. A novel approach to data deduplication over the engineering-oriented cloud systems. *Integrated Computer-Aided Engineering*, 20(1):45–57, 2013.

[59] K Meena and J Sujatha. Reduced time compression in big data using mapreduce approach and hadoop. *Journal of medical systems*, 43(8):1–12, 2019.

[60] Yimu Ji, Houzhi Fang, Haichang Yao, Jing He, Shuai Chen, Kui Li, and Shangdong Liu. Fastdrc: Fast and scalable genome compression based on distributed and parallel processing. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 313–319. Springer, New York City, USA, 2019.

[61] Shanshan Huang, Jungang Xu, Renfeng Liu, and Husheng Liao. A novel compression algorithm decision method for spark shuffle process. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2931–2940. IEEE, 2017.

[62] Wang Fuzong, Guo Helin, and Zhao Jian. Dynamic data compression algorithm selection for big data processing on local file system. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, pages 110–114, 2018.

[63] Apache Software Foundation. Apache hadoop, 2019, Online, https://hadoop.apache.org/, (Accessed: 2019-10-25).

[64] Lustre. Lustre, 2019, Online, http://lustre.org/, (Accessed: 2019-10-25).

[65] Julian Kunkel. Analyzing data properties using statistical sampling – illustrated on scientific file formats. *Supercomputing Frontiers and Innovations*, pages 19–33, 10 2016.

[66] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[67] Salvatore Sanfilippo. Redis is an open source (bsd licensed), in-memory data structure store, used as a database, cache and message broker. Redis Ltd., Mountain View, California, USA, 2019, Online, https://redis.io/, (Accessed: 2019-10-25).

[68] Jonathan Leibiusky. Jedis is a blazingly small and sane redis java client, 2019, Online, https://github.com/xetorthio/jedis, (Accessed: 2019-10-25).

[69] Project Nayuki. Native hash functions for java, 2019, Online, https://github.com/nayuki/Native-hashes-for-Java, (Accessed: 2019-10-25).

[70] Hao Fan, Guangping Xu, Yi Zhang, Liming Yuan, and Yanbing Xue. Csf: An efficient parallel deduplication algorithm by clustering scattered fingerprints. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications,*

Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pages 602–607. IEEE, 2019.

[71] Wikimedia. Wikimedia downloads: Xml dump for english wikipedia, 2021, Online, https://dumps.wikimedia.org/backup-index.html, (Accessed: 2021-03-04).

[72] Mario Giovanni Terzano, Liborio Parrino, Arianna Smerieri, Ronald Chervin, Sudhansu Chokroverty, Christian Guilleminault, Max Hirschkowitz, Mark Mahowald, Harvey Moldofsky, Agostino Rosa, Robert Thomas, and Arthur Walters. Erratum to "atlas, rules, and recording techniques for the scoring of cyclic alternating pattern (CAP) in human sleep" [sleep med. 2(6) (2001) 537–553]. *Sleep Medicine*, 3(2):185, mar 2002.

[73] G.B. Moody and R.G. Mark. A database to support development and evaluation of intelligent intensive care monitoring. In *Computers in Cardiology 1996*. IEEE, 1996.

[74] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In *Computer Vision – ECCV 2014*, pages 740–755. Springer International Publishing, 2014.

[75] Bert Hubert, Jacco Geul, and Simon Séhier. Wondershaper: Command-line utility for limiting an adapter's bandwidth, 2021, Online, https://github.com/magnific0/wondershaper, (Accessed: 2021-03-25).

[76] 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):1–28, 2017.

[77] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[78] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[79] Facebook. RocksDB, 2013, Online, https://rocksdb.org/, (Accessed: 2021-06-30).

[80] Sanjay Ghemawat and Jeff Dean. LevelDB, 2011, Online, https://github.com/google/leveldb, (Accessed: 2021-06-30).

[81] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.

[82] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based {KV} Store on Hybrid Storage. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 17–32, 2021.

[83] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, July 2017. USENIX Association.

[84] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187, 2020.

[85] Intel. Intel Optane Persistent Memory PMem, 2019, Online, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, (Accessed: 2021-06-30).

[86] Takahiro Hirofuchi and Ryousei Takano. A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module. *IEICE TRANSACTIONS on Information and Systems*, 103(5):1168–1172, 2020.

[87] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[88] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.

[89] Intel. The libpmem library, 2021, Online, https://pmem.io/pmdk/libpmem/, (Accessed: 2021-06-30).

[90] Intel. The libpmem2 library, 2021, Online, https://pmem.io/pmdk/libpmem2/, (Accessed: 2021-06-30).

[91] Intel. The libpmemobj library, 2021, Online, https://pmem.io/pmdk/libpmemobj/, (Accessed: 2021-06-30).

[92] Jens Axboe. Fio-flexible io tester. *URL http://freecode.com/projects/fio*, 2014.

[93] 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, 2010.

[94] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[95] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.

[96] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[97] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To fuse or not to fuse: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.

List of publications

Publication in the thesis:

- Widodo, R. N., Lim, H., Atiquzzaman, M. (2017, June). A new content-defined chunking algorithm for data deduplication in cloud storage. Future Generation Computer Systems, vol. 71, pp. 145-156.
- Widodo, R. N. S., Abe, H., Kato, K. (2020, August). HDRF: Hadoop Data Reduction Framework for Hadoop Distributed File System. In Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems, pp. 122-129.
- Widodo, R. N. S., Abe, H., Kato, K. (2021, November) Hadoop Data Reduction Framework: Applying Data Reduction at the DFS Layer. IEEE Access journal, vol. 9, pp. 152704-152717.
- Widodo, R. N. S., Ohtsuji, H., Hayashi, E., Yoshida, E., Abe, H., Kato, K. NVKVS: Non-Volatile Memory Optimized Key-Value Separated LSM-Tree. IPSJ Journal of Information Processing. Accepted for publication.

Published papers that are not in the thesis:

- Widodo, R. N., Lim, H., Atiquzzaman, M. (2017, May). SDM: Smart deduplication for mobile cloud storage. Future Generation Computer Systems, vol. 70, pp. 64-73.
- Widodo, R. N., Lim, H. (2017, January). RDM: Rapid Deduplication for Mobile Cloud Storage. In Proceedings of the 8th International Conference on Computer Modeling and Simulation, pp. 14-18.

Other publications:

- Widodo, R. N. S., Abe, H., Kato, K. (2020, November). DFS on a Diet: Enabling Reduction Schemes on Distributed File Systems. SC20 Research Poster.