USING DEDUPLICATION TO IMPROVE STORAGE EFFICIENCY IN DISTRIBUTED FILE SYSTEMS

By:

Paul Bartus

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTING AND INFORMATION SCIENCES AND ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

2018

Approved by:

Emmanuel Arzuaga, Ph.D. President, Graduate Committee	Date
Manuel Rodríguez Martínez, Ph.D. Member, Graduate Committee	Date
Pedro I. Rivera Vega, Ph.D. Member, Graduate Committee	Date
Wilson Rivera Gallego, Ph.D. Member, Graduate Committee	Date
Daniel Rodríguez, Ph.D. Representative of Graduate School	Date
Jaime Seguel, Ph.D. CISE Ph.D. Program Coordinator	Date

Abstract of Thesis Dissertation Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of DOCTOR OF PHILOSOPHY

USING DEDUPLICATION TO IMPROVE STORAGE EFFICIENCY IN DISTRIBUTED FILE SYSTEMS

By

Paul Bartus

July 2018

Chair: Emmanuel Arzuaga, Ph.D.
Major Department: Computing and Information Sciences and Engineering

Storage systems contain redundant copies of data such as identical files or within sub-file regions. Using deduplication technology, we can take advantage of this redundancy and reduce the space needed to store files in the file system. Scalable, highly reliable distributed systems supporting data deduplication have recently become popular for storing backup and archival data. There is potential for this technology to be adapted to primary storage.

This dissertation is focused on solving the storage problem, designing and developing HD2FS, improving data storage capacity and efficiency in distributed file systems.

Resumen de Disertación Doctoral Presentado a Escuela Graduada
de la Universidad de Puerto Rico como requisito parcial de los
Requerimientos para el grado de DOCTOR EN FILOSOFIA

USO DE DEDUPLICACIÓN PARA MEJORAR LA EFICIENCIA DE ALMACENAMIENTO EN SISTEMAS DE ARCHIVOS DISTRIBUIDOS

Por

Paul Bartus

Julio 2018

Consejero: Emmanuel Arzuaga, Ph.D.
Departamento: Ciencias e Ingeniería de la Información y la Computación

Los sistemas de almacenamiento contienen copias redundantes de datos, como archivos idénticos o dentro de regiones de subarchivos. Utilizando la tecnología de deduplicación sobre esta redundancia, reducimos el espacio necesario para almacenar archivos en el sistema de archivos. Recientemente se han popularizado los sistemas distribuidos escalables y altamente confiables que respaldan la deduplicación de datos para almacenar datos de copia de seguridad y archivado. Existe la posibilidad de que esta tecnología se adapte al almacenamiento primario.

Esta disertación se enfoca en resolver el problema de almacenamiento de datos mediante el diseño y desarrollo del sistema HD2FS, logrando mejorar la capacidad y la eficiencia del almacenamiento en el sistema de archivos distribuidos.

Dedicated to my parents.

ACKNOWLEDGMENTS

First, I would like to highly thank to my advisor, Dr. Emmanuel Arzuaga for his guidance, support, advice as well as dedication and help during my doctoral studies at the University of Puerto Rico Mayagüez.

I would like to thank to my Graduate Committee members, Dr. Manuel Rodríguez Martínez, Dr. Pedro I. Rivera Vega, and Dr. Wilson Rivera Gallego for their dedication and help with the preparation of my dissertation.

I would like to thank to Dr. Bienvenido Vélez, Director of Computer Science and Engineering Department and to Dr. Jaime Seguel, Associate Director of Computer Science and Engineering Department and Computing and Information Sciences and Engineering (CISE) Ph.D. Program Coordinator. I would like to thank to CSE, CISE Faculty members and Administrative staff.

I would also like to thank to Mr. Luis Lugo, Communications Technologies Specialist at Electrical and Computer Engineering (ECE) Department and to Mr. Victor Asencio Casiano, Computing and Telecommunications Equipment Specialist at Department of Electrical and Computer Engineering (ECE) and Laboratory for Applied Remote Sensing and Image Processing (LARSIP) for their help in providing the equipments such as the servers and the Virtual machines used for my research and also for their technical support.

Very special thanks to Dr. Gloria Isidro for her unconditional support and help during all these time in Puerto Rico, and to my family for their support during my doctoral studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS LIST

BP	Base Pairs
CDC	Content Defined Chunking
DBMS	Database Management System
DE	Deduplication Engine
DFS	Distributed File System
FD-HDFS	File Deduplicated HDFS
GFS	Google File System
HDFS	Hadoop Distributed File System
HD2FS	Hadoop Distributed and Deduplicated File System
HPC	High Performance Computing
PDF	Portable Document Format
RFD-HDFS	Reliable File Deduplicated HDFS
SHA	Secure Hash Algorithm

Chapter 1
INTRODUCTION

In recent years, the storage needs are increasing faster than the price drop in the storage systems. The production and sharing of massive amounts of data has become an everyday necessity. Cloud and Big Data are pushing storage scale to new high levels. There will be about 44 ZB of digital data by 2020 and it is expected to grow exponentially. The amount of money companies have to invest every year in new storage systems is increasing. There is a need of new methods that are capable of managing such data scales efficiently.

1.1 Justification

Storage systems typically contain redundant copies of data such as identical files or within sub-file regions. Using deduplication technology can take advantage of this redundancy and reduce the space needed to store files in the file system. Scalable, highly reliable distributed systems supporting data deduplication have recently become popular for storing backup and archival data. There is potential for this technology to be adapted to primary storage.

Deduplication systems divide files into chunks (data blocks) and identify redundant chunks by comparing their chunk identifiers (fingerprints) [1]. Typical deduplication solutions rely on fixed-size chunking methods. Although it provides a simpler implementation, the benefits of deduplication in those scenarios limit the use of this technique mostly to backup data [2–4]. The file recipe for a file is a synopsis that contains a list of chunk identifiers (fingerprints) that comprise the file. Each chunk identifier can be created using a collision resistant hash like SHA-1 or SHA-256 over the contents of the block [5]. Once the chunk identifiers in a file recipe have been obtained, they can be combined as prescribed in the file recipe to reconstruct the file.

1.2 Problem Statement

The price drop in the storage systems can not compensate for the continuous increase in the storage needs. Also, cloud and Big Data are pushing storage scale to new high levels. It is estimated that there will be about 44 ZB of digital data by 2020 and it is expected an exponential grow. Distributed file systems are widely used and store huge amounts of redundant data.

Despite lots of research have been done in deduplication, there is no research in finding a relation between the percentage of duplicate content and the percentage of duplicate chunks for any type of files. By having a file-aware chunking mechanism [6] there is potential to integrate this technology into primary storage solutions [7].

1.3 Purpose

The purpose of this research is to improve data storage capacity and efficiency in distributed file system environments using deduplication. The Hadoop Distributed File System (HDFS) is the one most used distributed file systems and it is open sourced. These two factors make it an ideal candidate to integrate deduplication. Thus, in this thesis we extend HDFS to include deduplication.

1.4 Distributed File Systems

A distributed file system (DFS) is a distributed implementation of a traditional model of a file system, where multiple users share files and storage resources through a network such as the Internet.

1.4.1 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is used to solve the storage problem of huge data, but does not provide a handling mechanism of duplicate content. HDFS is based on Google File System (GFS) and it operates on top of the operating system.

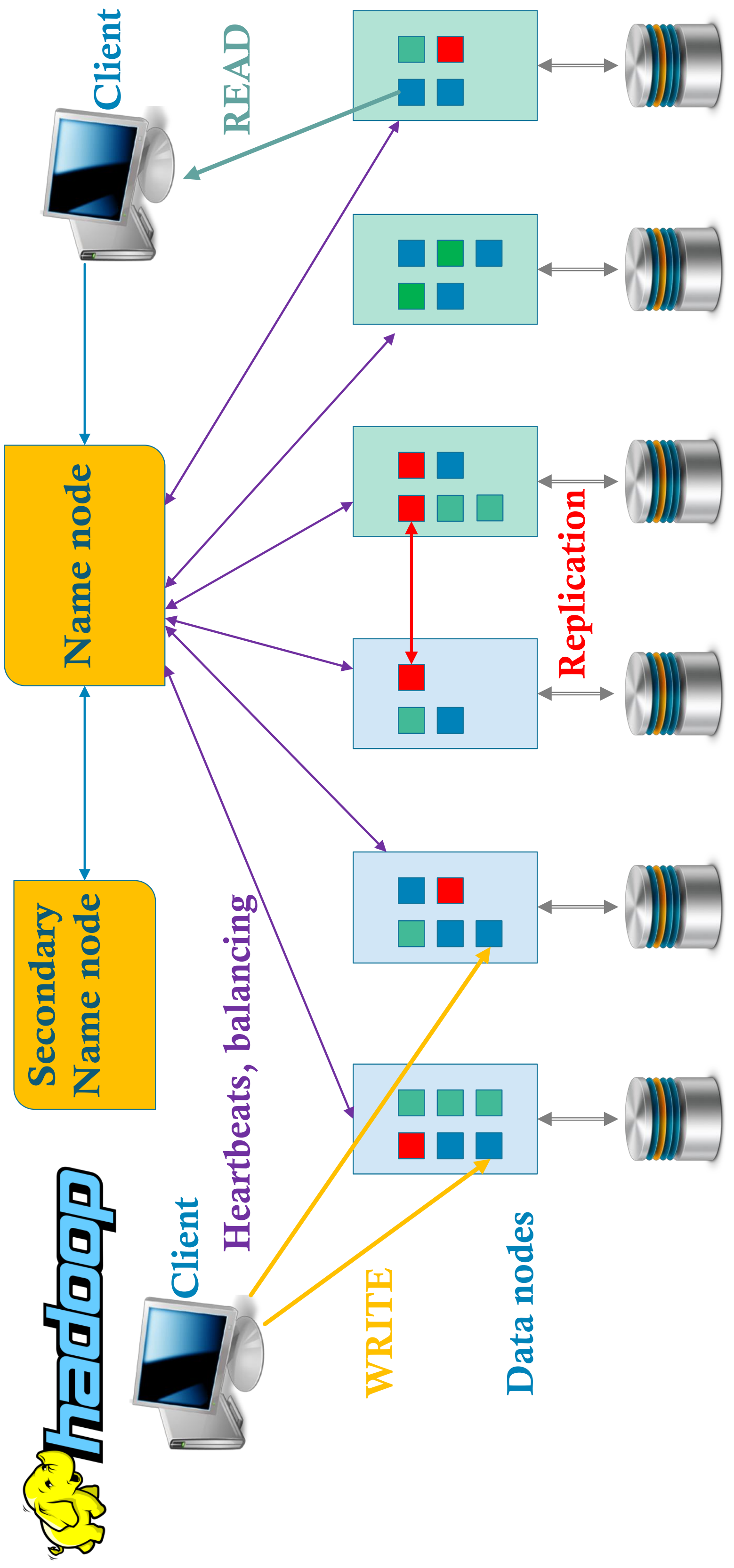


Figure 1–1: Typical Hadoop Distributed File System configuration.

Figure 1–1 shows a typical Hadoop Distributed File System (HDFS) configuration. HDFS has a name node, an optional secondary name node, and several data nodes. The name node is managing access and storing all metadata, such as file names, file attributes, and block locations. For fast lookup, the metadata is kept in RAM. Files contents are stored as blocks distributed across data nodes. It provides reliability throw replication, allowing copies of blocks to be stored on several data nodes. Client writes data directly to the data nodes. The data nodes will forward the data to the next data node for replication. The default replication is 3. Client reads the data directly from the nearest data nodes. Periodically, the name node is exchanging heartbeats with the data nodes, signals that indicate that the data node is alive. If no response is received, the corresponding data node is considered lost. In this case, the blocks from the lost data node will be replicated to another data nodes. In the case of a name node failure, if a secondary name node is configured, it will assume it's role as name node. For safety, it is highly recommended the secondary name node to run on a different machine.

1.5 Deduplication

Recently, the use of deduplication has shown potential to remove storage redundancy in similar files across file systems. The concept of a file can be adapted to refer to chunks (data blocks) and file recipes [3].

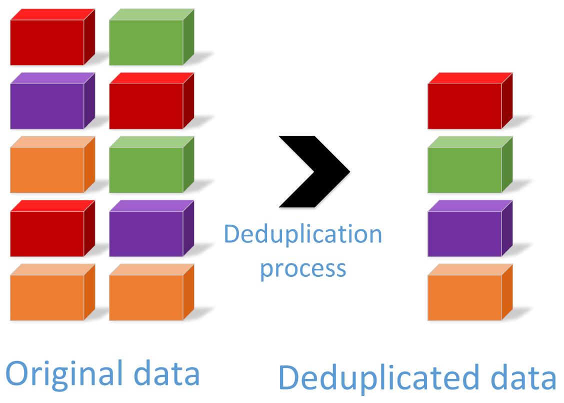


Figure 1–2: Deduplication process.

As Figure 1–2 shows, deduplication systems divide files into chunks (data blocks) and identify redundant chunks by comparing their chunk identifiers [1]. The chunk index (fingerprint) contains the chunk identifier of the stored chunk. Each chunk identifier can be created using a collision resistant hash like SHA-1 or SHA-256 over the contents of the block [5]. The file recipe for a file is a synopsis that contains a list of chunk identifiers (fingerprints) that comprise the file. Once the chunk identifiers in a file recipe have been obtained, they can be combined as prescribed in the file recipe to reconstruct the file.

There are several deduplication systems already available. The majority of the deduplication systems are for backup data and they are using fixed-size chunks.

1.6 Concepts

The most important concepts in this research are:

Deduplication. Deduplication systems divide files into chunks (data blocks) and identify redundant chunks by comparing their chunk identifiers [1].

Chunk index (fingerprint). The chunk index (fingerprint) contains the chunk identifier of the stored chunk. Each chunk identifier can be created using a collision resistant hash like SHA-1 or SHA-256 over the contents of the block [5].

File recipe. The file recipe contains a list of chunk identifiers (fingerprints).

Distributed file system (DFS). A distributed file system (DFS) is a distributed implementation of a traditional model of a file system, where multiple users share files and storage resources through a network such as the Internet.

Hadoop Distributed and Deduplicated File System (HD2FS)

A new deduplicated and distributed file system using Hadoop Distributed File System (HDFS) by adding deduplication.

1.7 Contributions

The main contribution of this research is to integrate deduplication technology in distributed file system environments to improve efficiency in the storage of large amounts of data. Our prototype system uses Hadoop Distributed File System (HDFS). HDFS is used to solve the storage problem of large data sets, but does not provide a mechanism to handle duplicate content within file boundaries. The experimental results show that deduplication provides superior efficiency compared to file compression for both (file read-decompress or write-compress) I/O patterns, highlighting the potential for this technology to be effectively adapted to improve storage systems capacity.

Other contributions of this research include:

- The design and implementation of a distributed datastore using deduplication, which we called *Smartstorage: a deduplicated and distributed datastore* [8],
- A characterization of file attributes that help determine the appropriate chunk size in primary deduplication systems, *The use of file attributes to determine the best chunk size in primary deduplication* [9],
- The development of a file-aware deduplication storage system. *Using file-aware deduplication to improve capacity in storage systems* [10],
- The design and implementation of a deduplication storage system for genomics data, in order to improve data storage capacity and efficiency in distributed file systems without compromising I/O performance. *GDedup: Distributed file system level deduplication for genomic big data* [11],
- Integrating deduplication to a distributed file system environment such as the Hadoop Distributed File System (HDFS). *Distributed file system level deduplication* [12].

1.8 Outline

This thesis is organized as follows: Chapter 2 presents the background and related works. Chapter 3 describes the design and implementation of a deduplicated and distributed file system using Hadoop Distributed File System (HDFS), the experiments done so far with the corresponding results and the new contributions, discuss the advantages and disadvantages of such strategy. Chapter 4 describes an application of the newly created file system, HD2FS and how to efficiently store data in cloud. Chapter 5 describes an application of HD2FS to manage large scale genomic data. Chapter 6 contains the conclusions.

Chapter 2
RELATED WORK

This chapter presents the related work addressing deduplication. The main topics are: Classification of Storage Deduplication Systems, Fixed-sized Deduplication, Inline Deduplication, Primary Storage Deduplication, and Hadoop-based systems. Deduplication systems divide files into chunks (data blocks) and identify redundant chunks by comparing their chunk identifiers (fingerprints) [1]. Different indexes are used to manage the relations between files and chunks, which require additional capacities beside the deduplicated data. The chunk index contains the chunk identifiers of the stored chunks. Every deduplication system has an additional persistent index to store the information that is necessary to rebuild file contents based on file recipes.

A file recipe contains a list of chunk identifiers. Each of these chunk identifiers represents a unique identifier for a particular chunk. Using the file recipe, the original file contents (denoted as logical data) can be reconstructed by using the uniquely identifiable chunked data [13]. To reconstruct the logical data, the chunk identifiers are read and their associated data chunks are loaded and concatenated in the specified order.

2.1 Classification of Storage Deduplication Systems

Paulo and Pereira [14] present a classification of deduplication systems according to six criteria that correspond to key design decisions: *granularity, locality, timing, indexing, technique, and scope. Granularity* refers to the method used for partitioning data into chunks. *Locality* [15] is used to support caching strategies can be temporal or spatial. *Timing* refers to when duplicates are eliminated. Duplicates are eliminated before storing the file to the file system for *inline deduplication* [16, 17], and after storing the file to the file system for *offline deduplication. Indexing* provides an efficient data structure for duplicate discovery. *Technique* can be chunk-based, or if eliminates duplicate content among two similar but not fully identical chunks, delta encoding. The *scope* for distributed deduplication systems can be local or global.

Deduplication can be regarded as bidirectional mapping between the logical view (containing identifiable duplicates) and the physical view (stored in actual devices from which duplicates have been removed).

2.2 Fixed-sized Deduplication

Zhao et al. [18] proposed Liquid, a distributed file system particularly designed to simultaneously address the above problems faced in large-scale VM deployment. However, they are considering chunk sizes multiples of 4 KB, between 256 KB and 1 MB. Their idea was to delay fingerprint calculation and use multiple threads. Liquid delays fingerprint calculation for recently modified data blocks. In order to speed up the deduplication process, a group of four fingerprint calculation threads is used. One of the concurrent threads calculates the fingerprint for one data block at one time, so multiple threads will process different data blocks concurrently. Data blocks are split into groups according to their fingerprints. Frequently accessed data blocks are cached in memory. It uses the copy-on-read technique to bring data blocks from data servers and peer clients to local cache on demand as they are being accessed by a VM. Liquid also provides fault tolerance through data replication, data migration, and hot backup of the meta server. It is compiled with block size as a parameter and it offers fault tolerance by mirroring the meta server, and by replication on stored data blocks.

Fu et al. [19] introduces Near-exact Deduplication, where a small number of duplicate chunks are allowed for higher backup performance and lower memory footprint. Data deduplication is disassembled into a large N-dimensional parameter space. Each point in the space is of various parameter settings, and performs a tradeoff among backup and restore performance, memory footprint, and storage cost. They propose a general purpose framework to evaluate various deduplication solutions in the space. Their goal was to find some reasonable solutions that have sustained backup performance and perform a suitable tradeoff between deduplication ratio, memory footprints, and restore performance. The fingerprint index consists of two submodules: a *key-value store* and a *fingerprint prefetching/caching* module. According to the use of the *key-value store*, the fingerprint index is classified into: *Exact Deduplication (ED)* if all duplicate chunks are eliminated for highest deduplication ratio (data size before deduplication / data size after deduplication), and *Near-exact Deduplication (ND)* if a small number of duplicate chunks are allowed for higher backup performance and lower memory footprint. According to the fingerprint prefetching policy, the fingerprint index is classified into exploiting: *Logical Locality (LL)* if the chunk (fingerprint) sequence of a backup stream before deduplication is preserved in recipes, and *Physical Locality (PL)* if the physical layout of chunks (fingerprints), namely the chunk sequence after deduplication is preserved in containers.

Mao et al. [20] introduce SAR, an SSD (solid state drive) Assisted Read scheme, effectively exploits the high random-read performance properties of SSD's, and the unique data-sharing characteristic of deduplication-based storage systems by storing in SSD's the unique data chunks with high reference count, small size, and non-sequential characteristics. Many read requests to HDD's are replaced by read requests to SSD's, significantly improving the read performance of the deduplication-based storage systems [21]. It transforms many small HDD-bound read I/Os to SSD-bound I/Os to fully leverage the significant random-read-performance and energy-efficiency advantages of the latter over the former. Also, improves system reliability and availability by significantly shortening the restore window. SAR outperforms the traditional deduplication-based system significantly in read operations by a speedup factor of up to 28.2, with an average factor of 5.8 in terms of the average response time while reduces the user response time of the traditional deduplication-based storage system by an average of 83.4 % and up to 176.6 %.

Clements et al. [22] introduces DEDE, a block-level deduplication system for live cluster file systems that does not require any central coordination, tolerates host failures, and takes advantage of the block layout policies of an existing cluster file system. Their idea was to use shared on-disk logs. Hosts keep summaries of their own writes to the cluster file system in shared on-disk logs, uses content hashes to identify potential duplicates, detects and eliminates duplicates introduced since the last index update, and stores summaries of recent modifications in on-disk write logs. DEDE must be resilient to stale index entries that do not reflect the latest content of recently updated blocks. However, the correctness does not depend on its ability to monitor every write to the file system and it uses only 4 KB fixed-size blocks. Also, requires the file system to be block oriented and to support file-level locking, block-level copy-on-write support. The file system block size must also align with the deduplication block size.

2.3 Inline Deduplication

Kim et al. [23] and Xie et al. [24] propose a content-based chunk placement scheme to increase deduplication rate on the DFS. To avoid performance overhead caused by deduplication process, Lessfs, a block-level and inline deduplication file system is used in each chunk server. Their idea was to use Lessfs in each chunk server. Consistent hashing for chunk allocation and failure recovery is used. Their experimental results show that the proposed system reduces the storage space by 60 % compared with the system without consistent hashing. Chunks are stored on the chunk server that has more available space than the other servers. All chunks of chunk server are stored in a mount point of Lessfs and the duplicated chunk data are eliminated by Lessfs. However, if the deduplication process is performed in a master server, those overheads cause bottleneck of a master server. Also, the data deduplication process is performed in each chunk server separetely.

Matsumiya et al. [25] develope ifarm, an implementation of Gfarm, a distributed file system used in the field of High Performance Computing (HPC) with deduplication of file data. Their idea was to compute the fingerprints asynchronously with client requests. It was considered only the case in which the computation nodes read and write too large data to store to disk spaces provided by the distributed file system (DFS). Also, DFS must load/store from/into the huge storage multiple times, and these operations can cause overheads on file access performance. However, it is required to reduce the amount of data stored in I/O servers and decreases data transfer operations between a distributed file system and huge storage. The inline deduplication method was used, where clients can access the storage during deduplication. ifarm is using content defined chunking (CDC), that divides a file into variable size chunks based on their content. Fast execution of CDC is required to archive practical inline deduplication. Their experiments show that ifarm is 117 % faster when sequentially reading a zero-data file (best case), and it is 99.8 % slower when sequentially reads a random-data file (worst case).

Jones [26, 27] defines a *sequence* as a group of consecutive file data blocks in an incoming file system write request. Their goal was to maximize the amount of data read per seek with the smallest impact to deduplication possible. Their idea was to use sequences. A sequence is considered a duplicate if a group of consecutive data blocks is found to be in the same consecutive order on disk. By using sequences, deduplicated file data will not be fragmented over the disk. A sequence is not deduplicated unless there is an exact match found on disk for the consecutive group of blocks in the sequence. It needs to exist on disk in the same consecutive order as the sequence in question. Three algorithms were compared: the Naive algorithm, the Sliding window algorithm, and the Minimum threshold algorithm.

Wildani et al. [28] implement HANDS, a framework that dynamically pre-fetches chunk identifiers from disk into memory cache according to working sets statistically derived from access patterns. Their idea was to use neighborhood grouping to dynamically pre-fetch fingerprints. It generates correlated groups of segments, or working sets, based on usage patterns. Loading groups of data into the index cache significantly reduces the number of accesses to the on-disk index cache while realizing most of the data reduction potential. HANDS can reduce the percentage of the index cache that is stored in memory to 1 % while still achieving 90 % of the optimal space savings. Working sets are groups of blocks that are likely to be accessed together. Instead of trying to predict the next access based on popularity, however, co-locate elements on disk if they are likely to be accessed together regardless of whether the elements have a high probability of being accessed at all. It applies a statistical analysis to a training set to establish initial groupings, and then alter these groupings based on their observed predictive capacity. Neighborhood Partitioning (NP) is a statistical method to compare data across multiple dimensions with a definable distance metric. Also, it introduce N-Neighborhood Partitioning (NNP), which merges several NP groupings without the memory overhead of a single large partitioning.

2.4 Primary Storage Deduplication

El-Shimi et al. [29] present a large scale study of primary data deduplication and use the findings to drive the design of a new primary data deduplication system. However, they are not considering chunk sizes below 4 KB.

Tarasov et al. [30, 31] implements Dmdedup, a versatile and practical primary deduplication platform operating at block layer. It has sequential and random read, write behaviour. Dmdedup was used to evaluate the benefits of hints [32] like *nodedup*(deduplication should only be performed when there is a potential benefit) and *prefetch* (inform the deduplication system of I/O operations that are likely to generate duplicates).

Koller and Rangaswami [33] propose *content similarity* to improve I/O performance, and Aronovich et al. [34] present the design of a similarity based deduplication system.

Chen and Shen [35] implement OrderMergeDedup, deduplicating writes to the primary flash storage with failure-consistency and high efficiency.

Klonatos et al. [36] implements ZBD, a block-layer driver transparently compressing/decompressing data between the file system and the storage device.

Jin and Miller [37] show the effectiveness of deduplication on VMDI's by loading groups of data into the index cache, significantly reducing the number of accesses to the on-disk index cache while realizing most of the data reduction potential.

Kumar et al. [38] present a genetic optimized data deduplication for distributed big data storage systems. The proposed genetic evolution algorithm, an optimized Two Thresholds Two Divisors content-defined chunking algorithm by significantly reducing the computing operations.

Kamboj et al. [39] present a deduplication system for encrypted data in cloud. The proposed Dedup App, a client side application for deduplication interface based on challenge of ownership and re-encryption. However, they are implementing file deduplication only, instead of finding duplicates within the file system.

2.5 Hadoop-based systems

The Hadoop Distributed File System (HDFS) is used to solve the storage problem of huge data, but does not provide a handling mechanism of duplicate files.

Ku et al. [17] use the middle layer file system in the HBASE virtual architecture to do file deduplicate in HDFS, with two proposed architectures: RFD-HDFS (Reliable File Deduplicated HDFS) where no errors are permitted, and FD-HDFS (File Deduplicated HDFS) where very few errors are permitted. In the case of RFD-HDFS, stream comparison is used to partially retrieve data fragments to conduct binary compare in the sequential serial method. Three phases are used to determine whether the files are the same: if the hash value exists, if the file sizes are the same, and gradually and sequentially executing stream comparison. For the collision policy of the duplicate files, if the SHA value of the file is the same, the file size is the same. In the case of FD-HDFS, the hashes with the same fingerprints are regarded as the duplicate files.

Sun et al. [40] present a deduplication cloud storage system, named DeDu, which runs on commodity hardware. The idea is to use link files and file level deduplication. At the front end, it has a deduplication application. At the back end, there are two main components, which are HDFS and HBase, respectively used as a mass storage system and a fast index. Both the MD5 and SHA-1 algorithm are used to make a unique fingerprint for each fIle, and set up a fast fingerprint index to identify the duplicates. A distribution file system DFS is developed to store data and develop "link files" to manage files in DFS. There are two ways to identify duplications in a cloud storage system: by comparing blocks or files bit to bit [41] (accurate, but time consuming), or by comparing blocks by hash values [42] (very fast, with a chance of accidental collision). The combination of MD5 and SHA-1 will greatly reduce this probability. The existing approaches for identifying duplicates work on two different levels: file level (decreases the quantity of hash values significantly), and chunk level (convenient for DFS to store chunks). The deduplication method in DeDu is at file level and based on comparing by hash values. HDFS and HBase were used as storage mechanisms. The advantage of HDFS is that it can be used under high throughput and large dataset conditions, and it is stable, scalable, and fault-tolerant. HBase is a Hadoop database. The hash value is calculated at the client side before data transmission while the lookup function is executed in HBase. When duplication is found, real data transmission will not occur.

Zhang et al. [43] present an inline data deduplication for SSD-based distributed storage architecture based on existing protocol of HDFS using two routing algorithms to assign chunks to data nodes that are most likely to contain similar blocks. However, they are not considering chunk sizes below 4 KB and they are not comparing the results obtained with the SSD-based distributed storage to regular HDD-based storage which is by far cheaper and do not present the write limit and disk endurance limitations.

Zhang et al. [44] propose an integrated deduplication approach by using Hadoop and levaraging parallelism based on Mapreduce and HBase. The hashes are stored in HBase and the file content in the HDFS. They are using the default setting of data block size of Hadoop (64 MB). Files are aggregated into a Hadoop sequence file, then the sequence file is evenly divided into chunks. However, this approach is more similar to file deduplication, their chunks are the Hadoop blocks.

Liu et al. [45] present a Hadoop based scalable cluster deduplication for big data, Halodedu. They use MapReduce and HDFS for parallel deduplication processing and managing data stores. MySQL is used to store the metadata, and the unique chunks are managed by HDFS. With their approach, the deduplication ratio is decreasing as the cluster size is increasing. However, with our HD2FS system, the deduplication ratio is not altered by the cluster size.

Chapter 3
HADOOP DISTRIBUTED AND DEDUPLICATED FILE SYSTEM (HD2FS)

This chapter presents the methodologies to design and implement a deduplicated and distributed file system. Our implementation relies on the Hadoop Distributed File System (HDFS). The primary reason for using HDFS as the base is that it is a widely used and open source distributed file system and by integrating our design into it, would increase adoption by the global community. Preliminary study was done on the relationship between the percentage of duplicate content and the percentage of duplicate chunks for the most common file types in order to determine the best chunk size for each type of file [10].

3.1 Preliminary Study

The study on the relationship between the percentage of duplicate content and the percentage of duplicate chunks for the most common types of files was performed. Given a file type, its best chunk size for deduplication was determined. The objective of the experiments was to understand the relationship between chunk size and file type in order to select a chunk size that can be used to add deduplication at the filesystem level. Focused on the amount of duplicate content, and given the duplicate content, the amount of duplicate chunks for different chunk sizes were computed. The results of the experiments provided us with sufficient information to identify the best chunk size for different file types. Finally, by knowing the best chunk size for a given file type, we used this information to build our deduplicated and distributed file system by integrating deduplication into Hadoop Distributed File System (HDFS).

3.1.1 Experimental Setup

The experiments were done on a DELL PowerEdge R220 server equipped with an Intel(R) Xeon(R) CPU E3 – 1286 v3 @ 3.70 GHz 8 core CPU, 32 GB of RAM, 400 GB SSD, and 12 TB of extra storage using regular hard drives. The experiments were run on the SSD and the results stored on the extra storage. Taking advantage of its low-latency access, the SSD was also used for storing the metadata. Depending on the chunk size used for deduplication, the metadata ranges from one percent of the total data for 4 KB chunks, to around ten percent of the total data for 512 B chunks. The operating system used was Ubuntu Linux 14.04 LTS with kernel Linux 4.2.0-42.

3.1.2 Data sets

The reference sets

The first step in this experiment was to build the data sets. Using random words from the English dictionary and the fact that the average sentence length in the English language is about 14 words, sentences were generated. Using the fact that in the English language there are about four or five sentences per paragraph, sentences were combined into paragraphs. In this way, with 3,600 paragraphs per file, 5,000 text (TXT) files with an average file size of 1.82 MB were generated. Also, using 380,000 random numbers between 1 and 500,000, comma separated values (CSV) files with an average file size of 2.45 MB were generated. In this way, 4,000 comma separated values (CSV) files were generated, each file with 2,500 rows and 152 columns. This two sets of files will be used as references throughout the experiments.

The modified sets

The second step was to modify the original sets of files. Let *sizeMB* denote the size of the modify block, a block used to introduce modified content, and *DupCont* denote the percentage of duplicate content. We used 256, 512, and 1,024 as *sizeMB* values, corresponding to 256 B, 512 B, and 1 KB modify block sizes, respectively. The *DupCont* values were 20, 40, 60, 80, 90, 95, and 98. For the cases of inserting new content, we also added the additional values of 30, 50, and 70.

Let Δ denote the file modification percentage. If a file is modified Δ percent, the corresponding *DupCont* will be $(100 - \Delta)$ percent. To obtain the same percentage of duplicate content as with a modify block size of 1 KB, we need to modify twice as many blocks of size 512 B and four times as many blocks of size 256 B.

The modifications were done at random locations, and within a set of modified files, *sizeMB* and *DupCont* values are fixed and unique.

For the text (TXT) files, three cases were considered. In the first case, blocks with the same size were interchanged, no new data was added to the files. In the second and third case, blocks were replaced with new random data without taking into consideration the English letters frequencies, and by taking into consideration the English letters frequencies, respectively. Figure 3–1 and Figure 3–2 show the character frequency distribution if blocks were replaced with new random data without taking into consideration the English letters frequencies, and the character frequency distribution if blocks were replaced with new random data by taking into consideration the English letters frequencies.

Table 3–1 shows the number of different sets of files, the total number of files, and the total size for the TXT files. The first line corresponds to the unmodified (reference) set of files.

For the comma separated values (CSV) files, blocks with the same size were interchanged, no new data was added to the files.

In the third step, from all the modified text (TXT) files, including the reference ones, Microsoft Word Documents (DOC), Microsoft Word Open XML Documents (DOCX), and Adobe Portable Document Format (PDF) files were generated. From

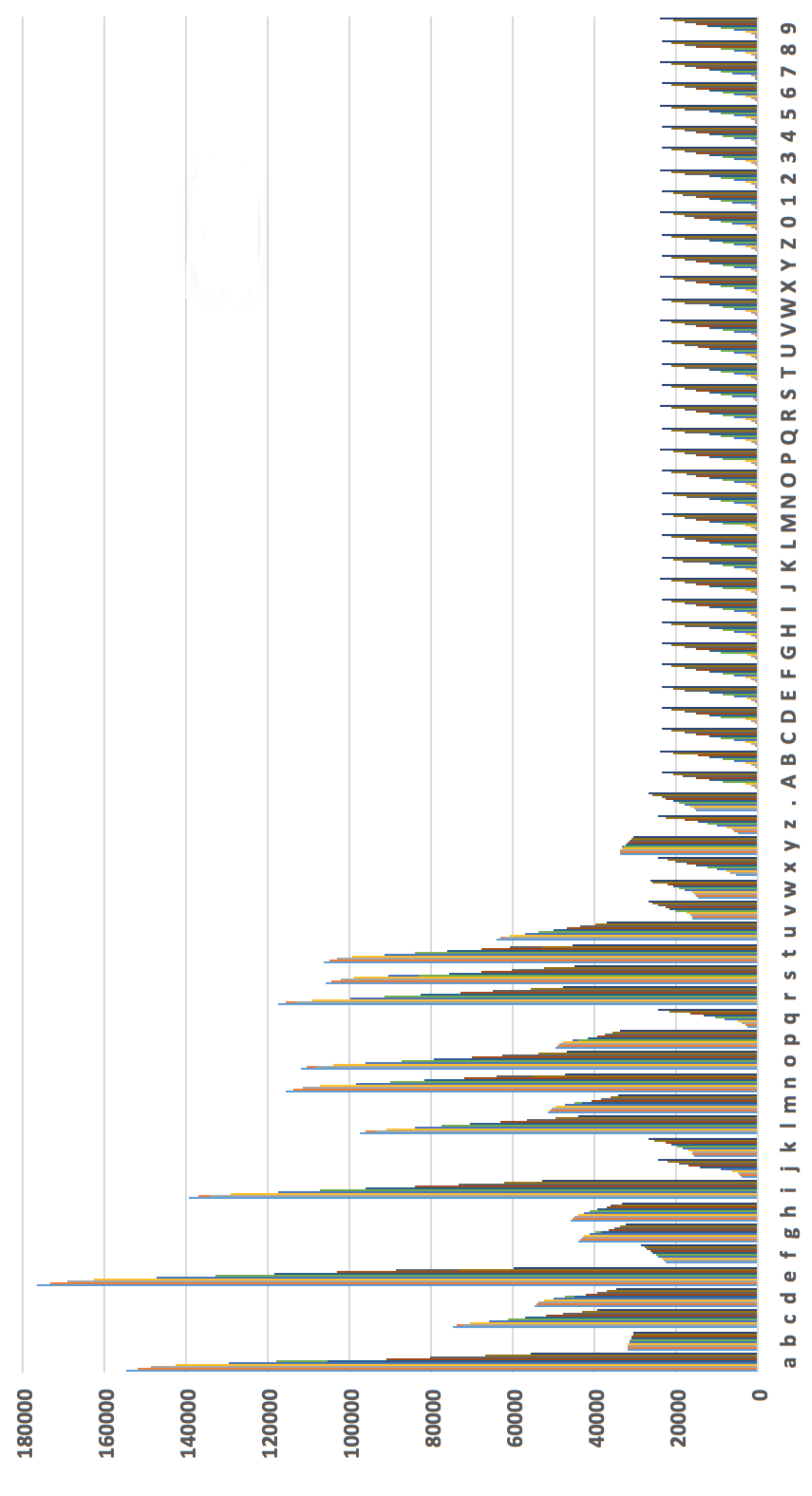


Figure 3–1: Character frequency distribution if blocks were replaced with new random data without taking into consideration the English letters frequencies. The X axis represent the characters and the Y axis represent the number of times a character appears in each file.

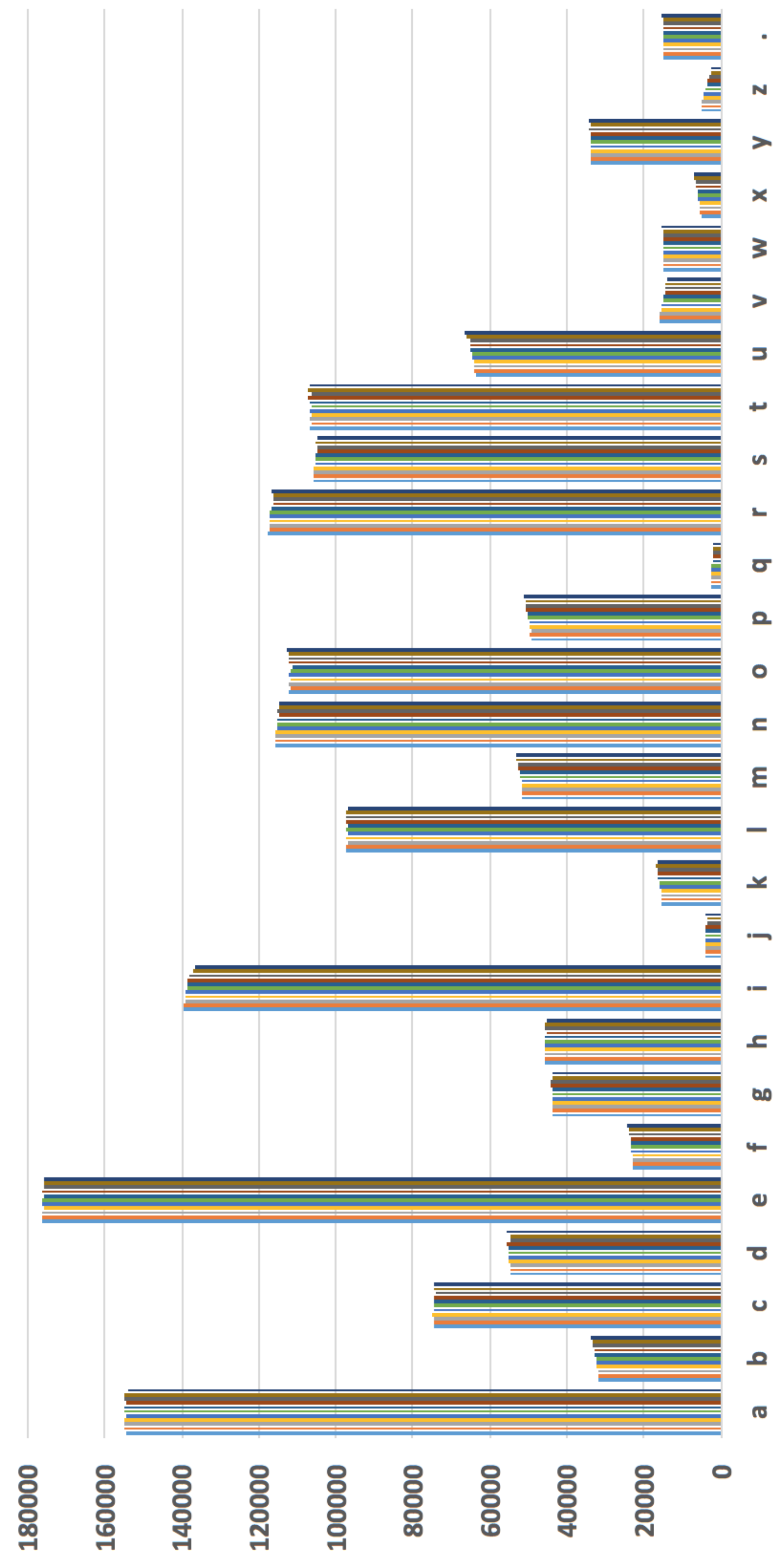


Figure 3–2: Character frequency distribution if blocks were replaced with new random data by taking into consideration the English letters frequencies. The X axis represent the characters and the Y axis represent the number of times a character appears in each file.

Table 3–1: The modified TXT files by interchanging blocks, and by inserting new random content. The first line corresponds to the unmodified (reference) set of files.

Modify Type	**Sets**	**Files**	**Size (GB)**
Unmodified (reference)	1	5,000	9
Interchanged blocks	21	105,000	187
Insert new content	60	300,000	534
Total:	**82**	**410,000**	**730**

Table 3–2: Statistics of the complete data set used for the preliminary study.

File tye	**Average size (MB)**	**Sets**	**Total size (GB)**	**Percentage of Total (%)**
DOC	3.89	82	1,558	42
TXT	1.82	82	730	20
PDF	1.60	82	640	17
DOCX	0.88	52	224	6
CSV	2.45	22	211	6
XLS	2.28	22	196	5
JPG	2.47	22	106	3
PPT	1.33	22	57	2
Total:		**386**	**3,722**	**100**

all the comma separated values (CSV) files, including the reference ones, Microsoft Excel (XLS) files were generated. LibreOffice command line converter was used to generate the Microsoft Word Documents (DOC), the Adobe Portable Document Format (PDF) files, and the Microsoft Excel (XLS) files. In order to generate the Microsoft Word Open XML Documents (DOCX), Pandoc universal document converter was used.

The Joint Photographic Experts Group files (JPG) and the Microsoft Power Point files (PPT) were real files, not generated ones.

Table 3–2 shows the statistics of the complete data sets. There are 386 different sets of files for a total of 3.6 TB of data. The Percentage of Total column shows the percentage of each file type taking into consideration their size.

3.1.3 Compare Two Files Using Deduplication

With the created data sets and using deduplication, we compared each file with its original one in order to compute the number of duplicate chunks between them. Fixed-length deduplication was used, each file being divided into equally sized chunks. Using SHA-1 secure hash algorithm, a unique chunk identifier (fingerprint) was generated for every chunk and sent to the database. If the identifier already existed, meaning that this chunk was already stored, the corresponding chunk counter was incremented by one. If the chunk identifier was not found in the database, it was added and the chunk was stored on the disk. In order to find the optimal chunk size for different file types, we considered the following potential chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB. The percentage of duplicate chunks between every modified set of files and its corresponding reference set of files was computed. In order to be able to compute the percentage of duplicate chunks for each case, deduplication was used. Each file type was considered separately because the probability of finding duplicates in files of the same type is significantly higher than between files of all types.

Let $chunkSize$ denote the chunk size and $DupCh$ denote the percentage of duplicate chunks. The values for $chunkSize$ were 512 for 512 B chunks, 1,024 for 1 KB chunks, 2,048 for 2 KB chunks, and 4,096 for 4 KB chunks, respectively. To compute the percentage of duplicate chunks ($DupCh$), the chunk identifiers from the original set of files were compared with the chunk identifiers from every modified set of files.

Figure 3–3 shows an example of the resulting chunks from two files, after deduplication. The first file is the reference file, and the second file is the modified file.

In this example, it was assumed that after deduplication, each file was divided into ten chunks of the same size. The first line contains the unmodified file, and the second line the modified file. Because the files were generated from random words,

A	*B*	**C**	*D*	*E*	*F*	**G**	*H*	*I*	**J**
A	*K*	**C**	*L*	*M*	*N*	**G**	**J**	*O*	*P*

Figure 3–3: Compare two files using deduplication. Each file contains ten chunks of the same size. Chunks A, C, G, and J are duplicates, even if chunk J appears at different locations in the files.

there are no duplicate chunks within a file nor within any set of files, except in the cases where the files have special formats and were generated using converters (DOC, DOCX, PDF, XLS). Fixed chunk sized deduplication was used. Every chunk identifier was recorded into a database, and using SQL queries the total number of chunks per file and the number of duplicate chunks were computed. In this example, there are four duplicate chunks (A, C, G, and J) from a total of ten chunks in each file. To compute the percentage of duplicate chunks between these two files, we used the following formula:

$$DupCh = \frac{Total\,Chunks - Unique\,Chunks}{Total\,Chunks} \tag{3.1}$$

where $DupCh$ represents the percentage of duplicate chunks, $Unique\,Chunks$ the number of chunks having count=1, and $Total\,Chunks$ represents the total number of chunks. Therefore, the percentage of duplicate chunks in this example is:

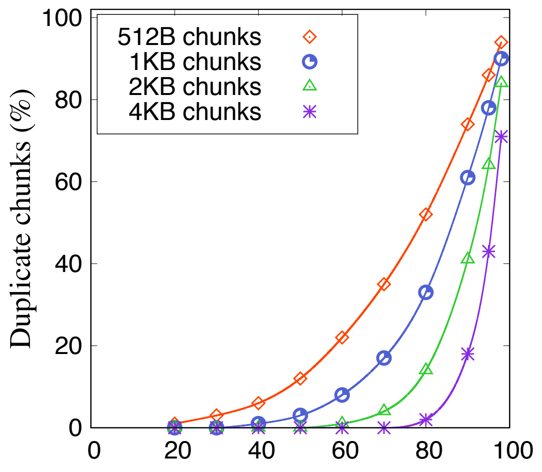
$$DupCh = \frac{20 - 12}{20} = 40\,\%.$$

If the modify block size (*sizeMB*) was smaller than or equal the chunk size (*chunkSize*), by changing one block, one or two chunks were altered. The bigger the difference between the chunk size and the modify block size, the higher the probability that only one chunk will be altered. If the modify block size was bigger than the chunk size, by changing one block, more than two chunks were altered. The later case applies only when the value of *sizeMB* is 1 KB and the value of *chunkSize* is 512 B, when two or three chunks were altered.

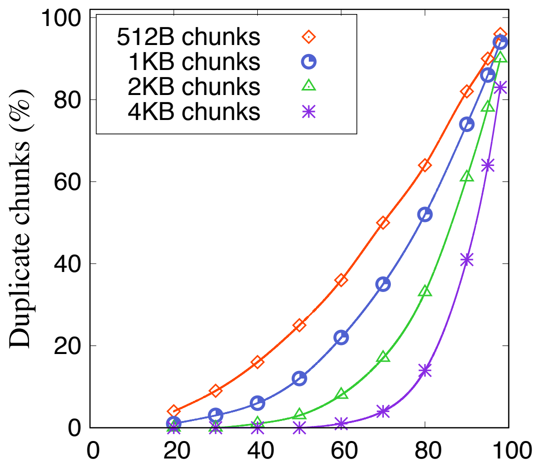
3.1.4 Results Analysis by File Type

Text files

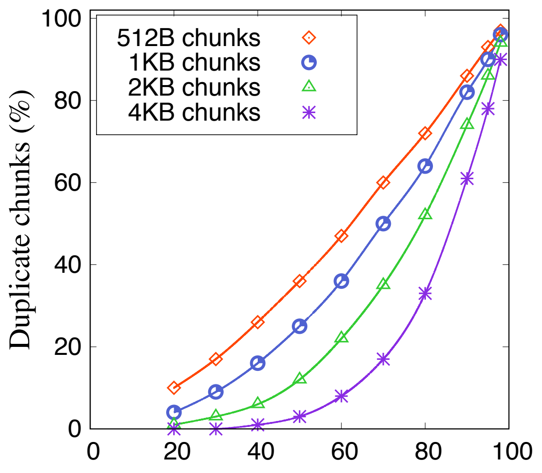
Figure 3–4 shows the percentage of duplicate chunks as function of the percentage of duplicate content for TXT, CSV, PPT, and JPG files for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB.



(a) 256 B modify blocks.



(b) 512 B modify blocks.



(c) 1 KB modify blocks.

Figure 3–4: Duplicate content analysis for TXT, CSV, PPT, and JPG files. The X axis represents the duplicate content percentage, and the Y axis represents the duplicate chunk percentage.

The three cases correspond to the following modify block sizes: 256 B in Figure 3–4a, 512 B in Figure 3–4b, and 1 KB in Figure 3–4c, respectively. We can see that as the chunk size increases, the percentage of duplicate chunks decreases. It decreases even faster as the modify block size decreases.

For 256 B modify blocks in Figure 3–4a, the percentage of duplicate chunks decreases from 94 % to 1 % ($DupCont$ = 20, $chunkSize$ = 512), from 90 % to 1 % ($DupCont$ = 40, $chunkSize$ = 1,024), from 83 % to 1 % ($DupCont$ = 60, $chunkSize$ = 2,048), and from 71 % to 1 % ($DupCont$ = 80, $chunkSize$ = 4,096), respectively.

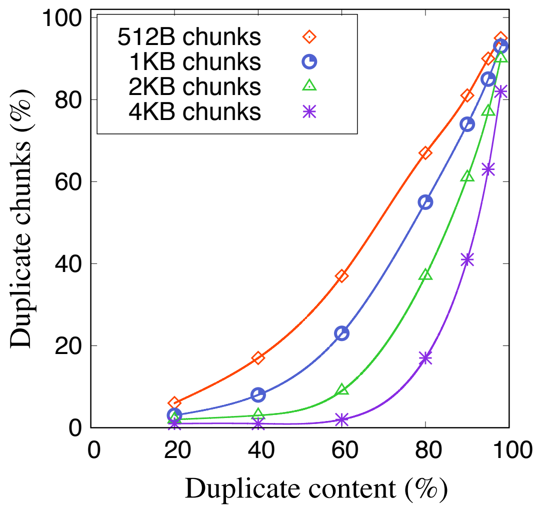
For 512 B modify blocks in Figure 3–4b, the percentage of duplicate chunks decreases from 96 % to 4 % ($DupCont$ = 20, $chunkSize$ = 512), from 94 % to 1 % ($DupCont$ = 20, $chunkSize$ = 1,024), from 90 % to 1 % ($DupCont$ = 40, $chunkSize$ = 2,048), and from 83 % to 1 % ($DupCont$ = 60, $chunkSize$ = 4,096), respectively.

For 1 KB modify blocks in Figure 3–4c, the percentage of duplicate chunks decreases from 97 % to 9 % ($DupCont$ = 20, $chunkSize$ = 512), from 96 % to 4 % ($DupCont$ = 20, $chunkSize$ = 1,024), from 94 % to 1 % ($DupCont$ = 20, $chunkSize$ = 2,048), and from 90 % to 1 % ($DupCont$ = 40, $chunkSize$ = 4,096), respectively.

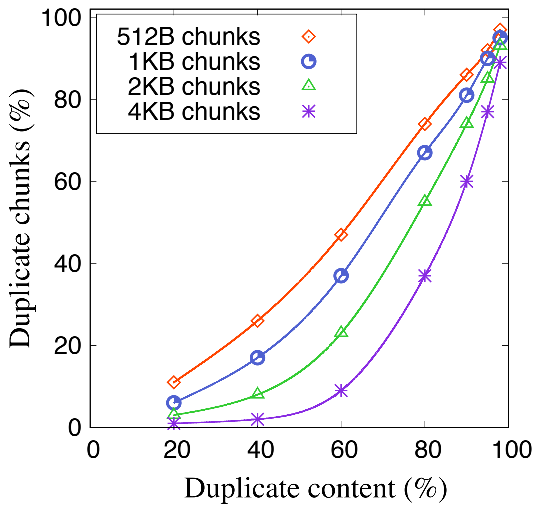
Microsoft Word Document files

Figure 3–5 shows the percentage of duplicate chunks as function of the percentage of duplicate content for Microsoft Word Document DOC files for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB, respectively.

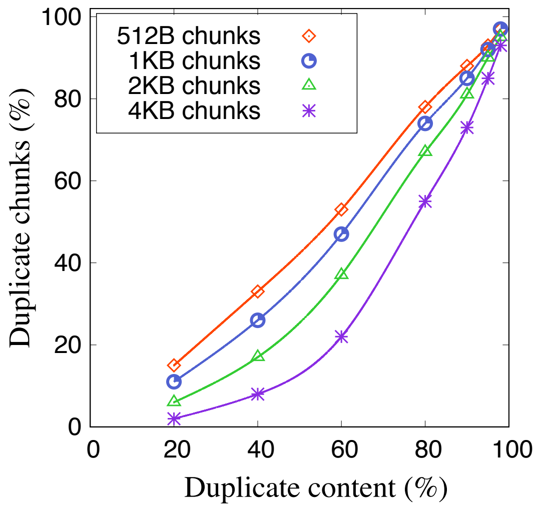
The three cases correspond to different modify block sizes: 256 B in Figure 3–5a, 512 B in Figure 3–5b, and 1 KB in Figure 3–5c, respectively. We can see that as the chunk size increases, the percentage of duplicate chunks decreases. It decreases even faster as the size of the modify block decreases. Figure 3–6 shows the percentage



(a) 256 B modify blocks.



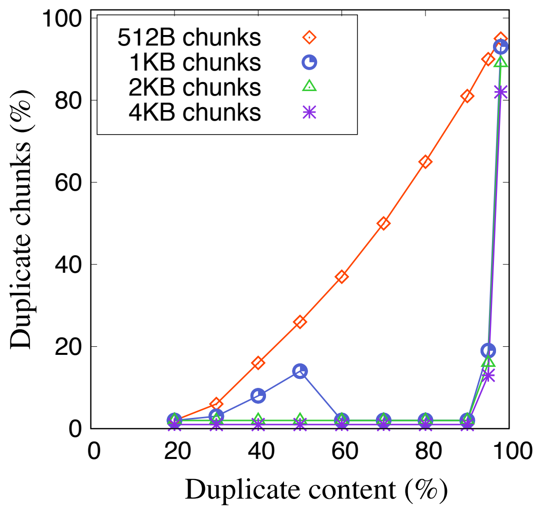
(b) 512 B modify blocks.



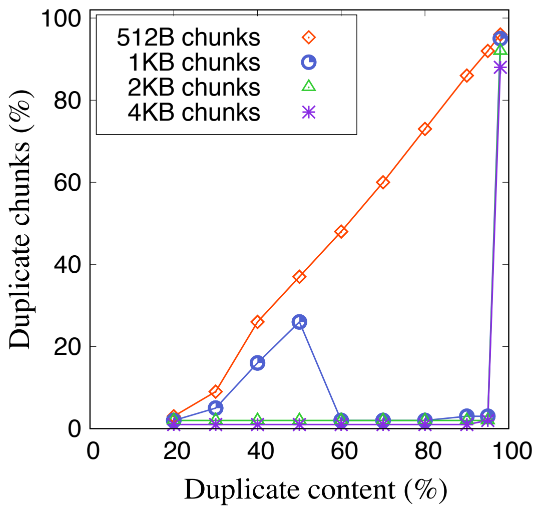
(c) 1 KB modify blocks.

Figure 3–5: Duplicate content analysis for DOC files with interchanges. The X axis represents the duplicate content percentage, and the Y axis represents the duplicate chunk percentage.

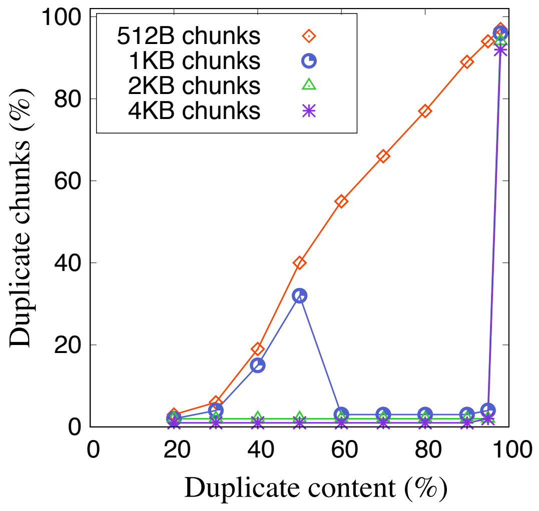
of duplicate chunks as function of the percentage of duplicate content for Microsoft Word Document DOC files corresponding to the case where the modifications were done by inserting new random data. We can see that for 512 B chunk size, the relationship between the percentage of duplicate chunks and the percentage of duplicate content is approximately linear. Up to 50 % duplicate content, the 1 KB chunks have almost the same behavior as the 512 B chunks, after that the percentage of duplicate



(a) 256 B modify blocks.



(b) 512 B modify blocks.



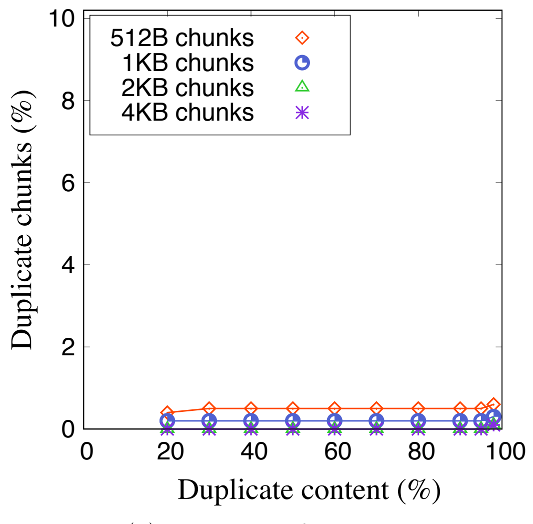
(c) 1 KB modify blocks.

Figure 3–6: Duplicate content analysis for DOC files with completely / controlled random content inserted. The X axis represents the duplicate content percentage, and the Y axis represents the duplicate chunk percentage.

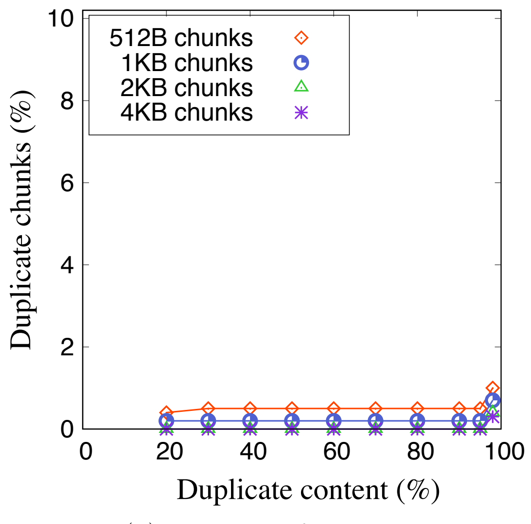
chunks is almost zero. For 2 KB and 4 KB chunk sizes, the percentage of duplicate chunks is almost zero if the percentage of duplicate content is less than 98 %. We found that if new data is inserted to Microsoft Word Document DOC files, 512 B chunk size should be used for the deduplication. Also, the use of English letters frequency had no effect on the percentage of duplicate chunks.

Microsoft Word Open XML Document files

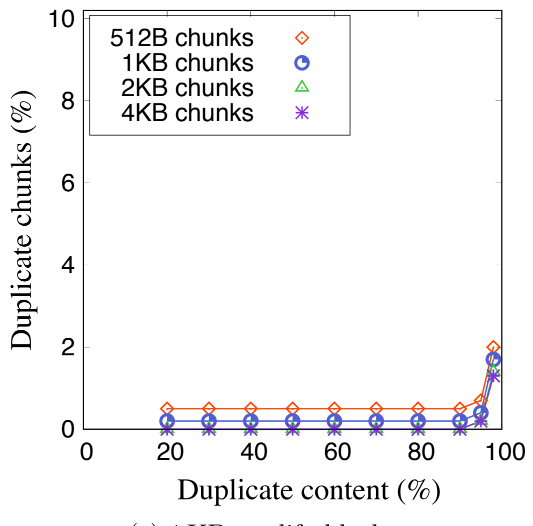
Figure 3–7 shows the percentage of duplicate chunks as a function the percentage of duplicate content for Microsoft Word Open XML Documents DOCX files for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB, respectively.



(a) 256 B modify blocks.



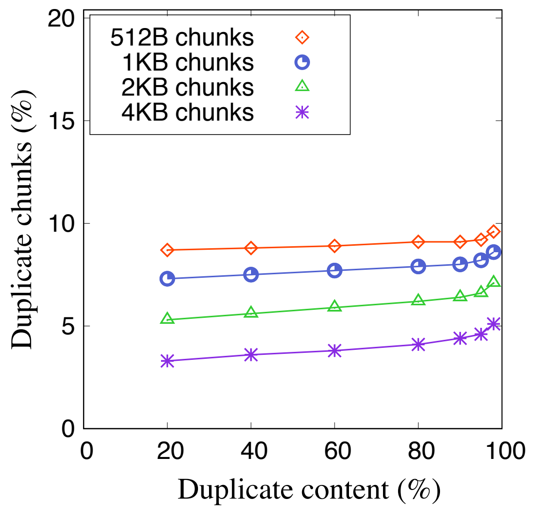
(b) 512 B modify blocks.



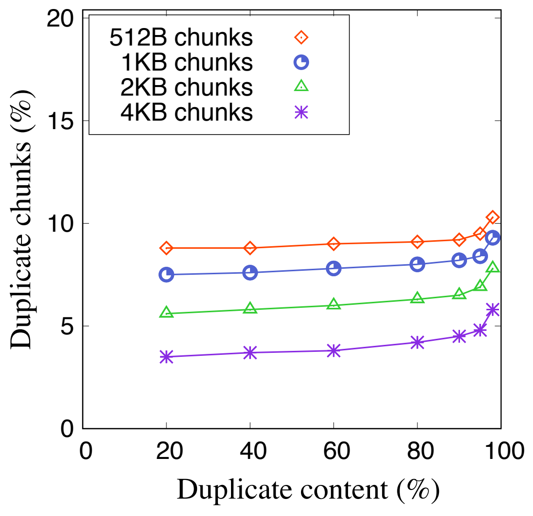
(c) 1 KB modify blocks.

Figure 3–7: Duplicate content analysis for DOCX files. The X axis represents the duplicate content percentage, and the Y axis represents the duplicate chunk percentage.

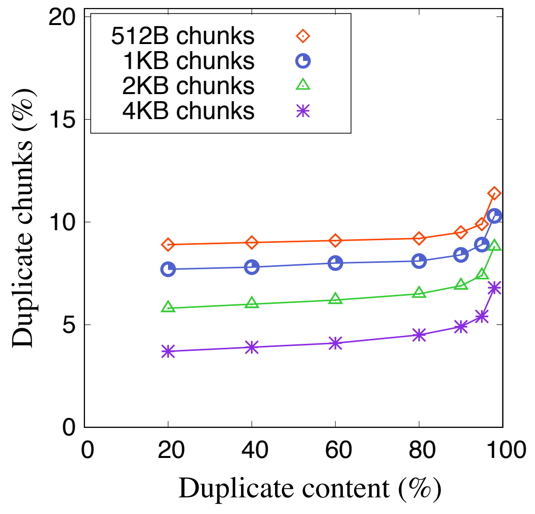
The three cases correspond to different modify block sizes: 256 B in Figure 3–7a, 512 B in Figure 3–7b, and 1 KB in Figure 3–7c. Because the DOCX files are already compressed, the percentage of duplicate chunks is almost zero, except the



(a) 256 B modify blocks.



(b) 512 B modify blocks.



(c) 1 KB modify blocks.

Figure 3–8: Duplicate content analysis for PDF files. The X axis represents the duplicate content percentage, and the Y axis represents the duplicate chunk percentage.

case when the size of modify block is 1 KB (*sizeMB*= 1KB) and the percetage of duplicate content (*DupCont*) is greater than 98 %, the percentage of duplicate chunks is about 2 %.

Portable Document Format files

Figure 3–8 shows the percentage of duplicate chunks as a function of the percentage of duplicate content for Portable Document Format (PDF) files for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB, respectively.

The three cases correspond to different modify block sizes: 256 B in Figure 3–8a, 512 B in Figure 3–8b, and 1 KB in Figure 3–8c, respectively. The percentage of duplicate chunks is around nine percent for 512 B chunk size and decreases to 4 % for 4 KB chunk size.

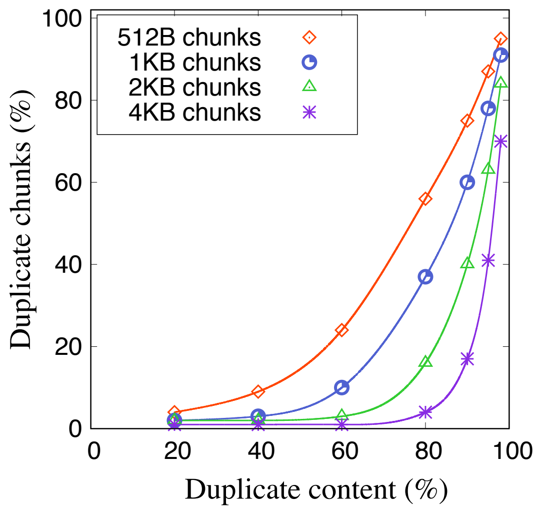
Microsoft Excel files

Figure 3–9 shows the percentage of duplicate chunks as function of the percentage of duplicate content for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB, respectively. The three cases correspond to different modify block sizes: 256 B in Figure 3–9a, 512 B in Figure 3–9b, and 1 KB in Figure 3–9c, respectively. We can see that as the chunk size increases, the percentage of duplicate chunks decreases. It decreases even faster as the size of the modify block decreases.

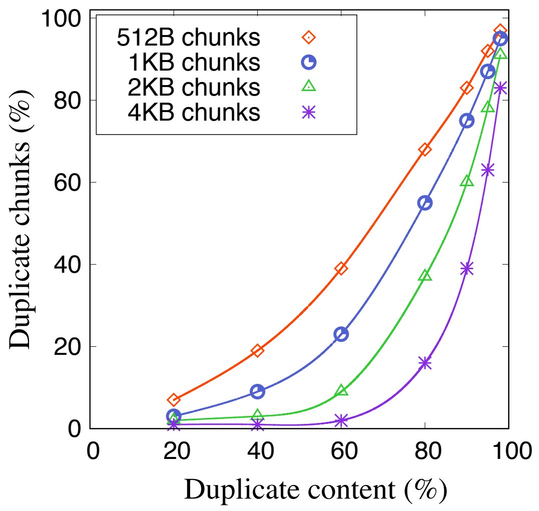
For 256 B modify blocks in Figure 3–9a, the percentage of duplicate chunks decreases from 95 % to 4 % ($DupCont$ = 20, $chunkSize$ = 512), from 91 % to 2 % ($DupCont$ = 20, $chunkSize$ = 1,024), from 84 % to 2 % ($DupCont$ = 40, $chunkSize$ = 2,048), and from 70 % to 1 % ($DupCont$ = 60, $chunkSize$ = 4,096), respectively.

For 512 B modify blocks in Figure 3–9b, the percentage of duplicate chunks decreases from 97 % to 7 % ($DupCont$ = 20, $chunkSize$ = 512), from 95 % to 3 % ($DupCont$ = 20, $chunkSize$ = 1,024), from 91 % to 2 % ($DupCont$ = 40, $chunkSize$ = 2,048), and from 83 % to 1 % ($DupCont$ = 40, $chunkSize$ = 4,096), respectively.

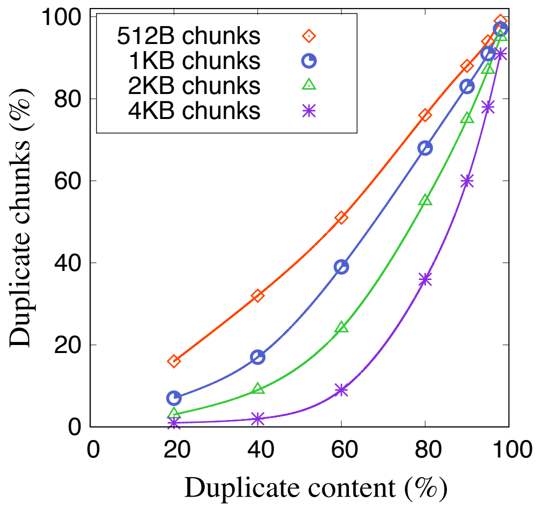
For 1 KB modify blocks in Figure 3–9c, the percentage of duplicate chunks decreases from 98 % to 16 % ($DupCont$ = 20, $chunkSize$ = 512), from 97 % to 7 % ($DupCont$ = 20, $chunkSize$ = 1,024), from 95 % to 3 % ($DupCont$ = 20, $chunkSize$ = 2,048), and from 91 % to 1 % ($DupCont$ = 40%, $chunkSize$ = 4,096), respectively.



(a) 256 B modify blocks.



(b) 512 B modify locks.



(c) 1 KB modify blocks.

Figure 3–9: Duplicate content analysis for XLS files. The X axis represents the duplicate content percentage, and the Y axis represents the duplicate chunk percentage.

Microsoft Powerpoint files and JPG images

Figure 3–4 shows the percentage of duplicate chunks as function of the percentage of duplicate content for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB, respectively. The Microsoft Powerpoint files and the JPG image files were modified without taking into consideration their format. Therefore, they have the same behavior as the already presented TXT files.

3.2 Metrics

3.2.1 Deduplication Ratio and Percentage of Duplicate Chunks

Let *chunkSize* denote the chunk size, *DedupRatio* denote the deduplication ratio, and *DupCh* denote the percentage of duplicate chunks. The deduplication ratio is defined as the ratio between the original data size and the non redundant data size. A higher *DedupRatio* value shows a high redundancy in the file content while a lower ratio shows a high number of unique chunks. The deduplication ratio is given by the following formula:

$$DedupRatio = \frac{Original\ Data\ Size}{Non\ Redundant\ Data\ Size} = \frac{Total\ chunks}{Distinct\ chunks} \tag{3.2}$$

The percentage of duplicate chunks is given by the following formula:

$$DupCh = \frac{Total\ chunks - Unique\ chunks}{Total\ chunks} \tag{3.3}$$

In our experiments, we used the following *chunkSize* values: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB.

3.2.2 Chunking and Reconstruction Times

The **chunking time** represents the time needed to write a file to the deduplicated system. As the diagram in Figure 3–10 shows, the file is divided in equally sized chunks, and the chunk indexes are computed. The chunk indexes are saved to the file recipe, which will be the new file. The chunks content are sent to the deduplication database.

The **reconstruction time** represents the time needed to read a file from the deduplicated system. As Figure 3–11 shows, for reading a file, the corresponding file recipe is first read from the file system, then the corresponding chunk contents are retrieved from the deduplication database and concatenated in order to reconstruct the original file.

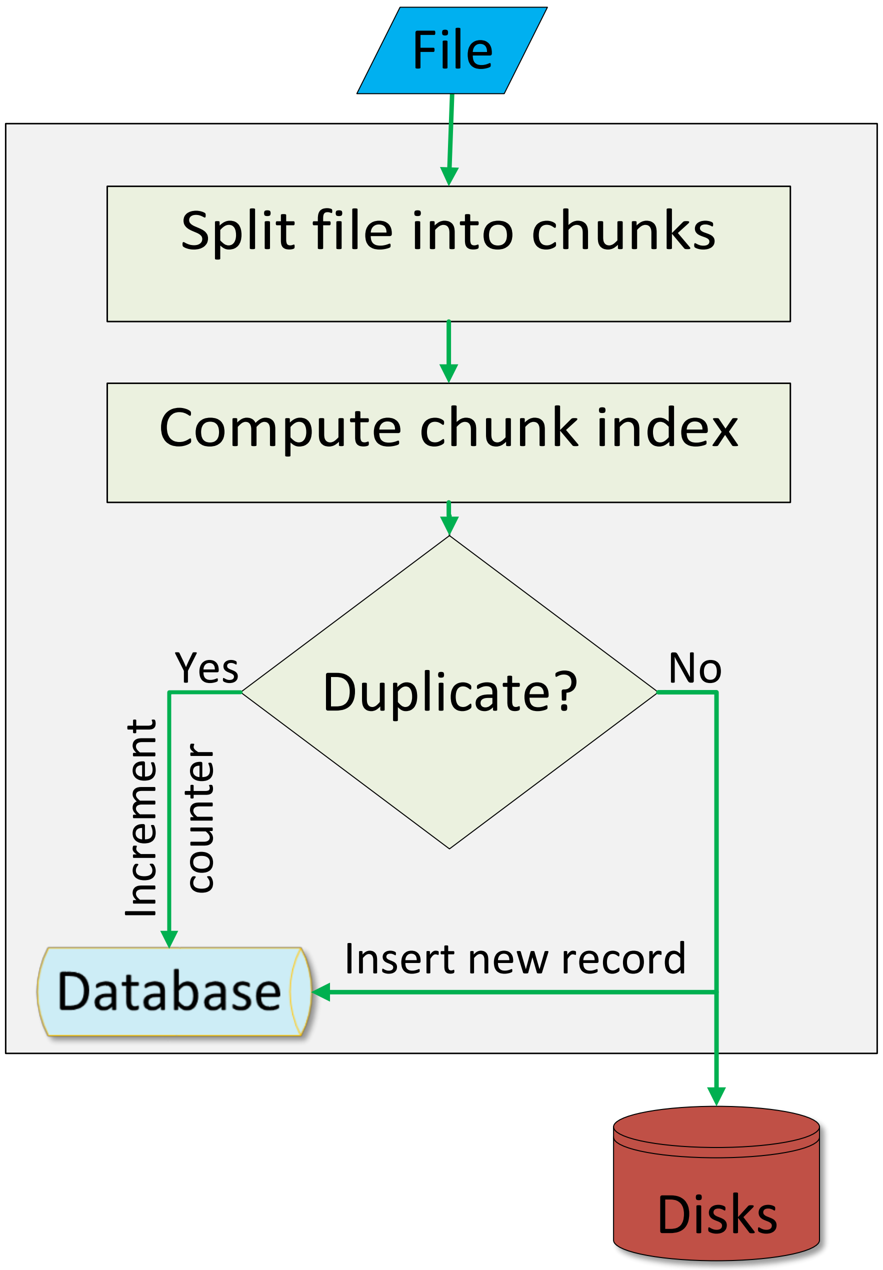


Figure 3–10: Deduplication (chunking) diagram.

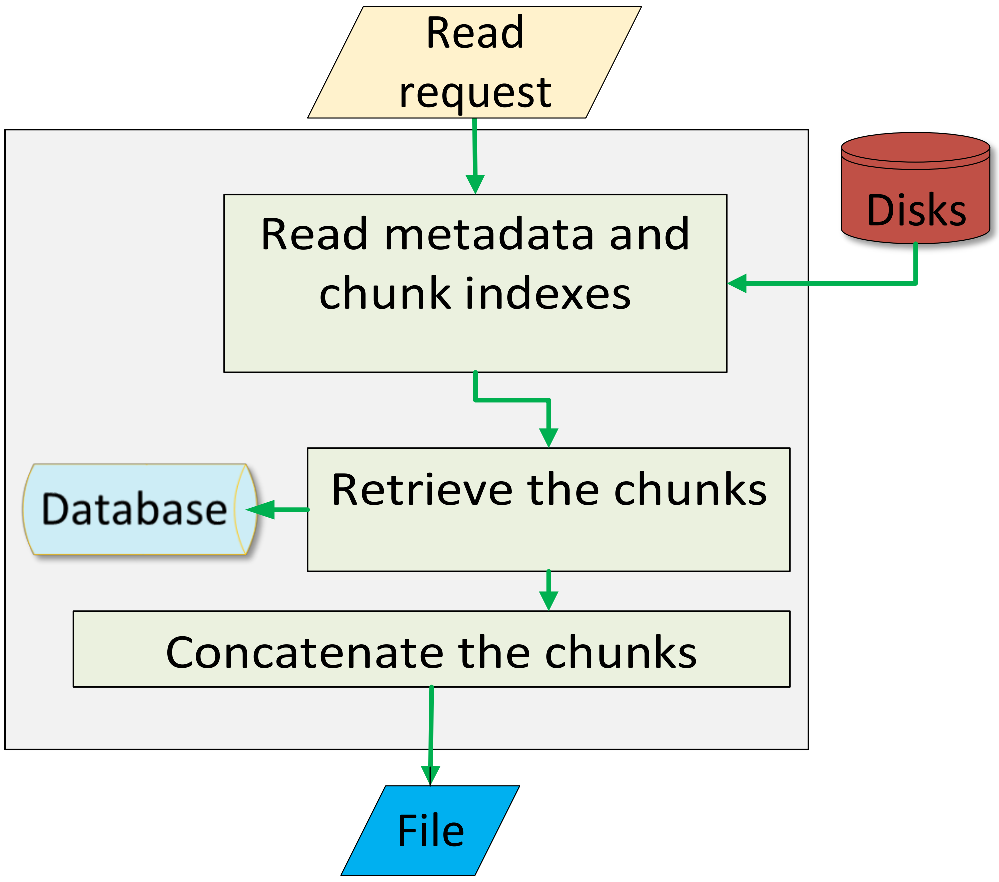


Figure 3–11: File reconstruction diagram.

3.3 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is used to solve the storage problem of huge data, but does not provide a handling mechanism of duplicate files. HDFS is based on Google File System (GFS) and it operates on top of the operating system.

As Figure 1–1 shows, HDFS has a name node, an optional secondary name node, and several data nodes. The name node is managing access and storing all metadata, such as file names, file attributes, and block locations. For fast lookup, the metadata is kept in RAM. Files contents are stored as blocks distributed across data nodes. It provides reliability throw replication, blocks are stored on several data nodes. Client writes data directly to the data nodes. The data nodes will forward the data to the next data node for replication. The default replication is three. Client reads the data directly from the nearest data nodes. Periodically, the name node is exchanging heartbeats with the data nodes. If no response is received, the corresponding data node is considered lost. In this case, the blocks from the lost data node will be replicated to another data nodes. In the case of a name node failure, if a secondary name node is configured, it will assume it's role as name node. For safety, it is highly recommended the secondary name node to run on a different machine.

3.4 Database Management Systems selection

The objective of this section is to find the optimal Database Management System in order to implement a deduplicated and distributed file system using Hadoop Distributed File System(HDFS). The following potential Database Management Systems were considered: MariaDB, MongoDB, PostgreSQL, and SQLite. For the case of MariaDB, the experiments were done for the following storage engines: ARIA, InnoDB, and MyISAM.

3.4.1 Data set

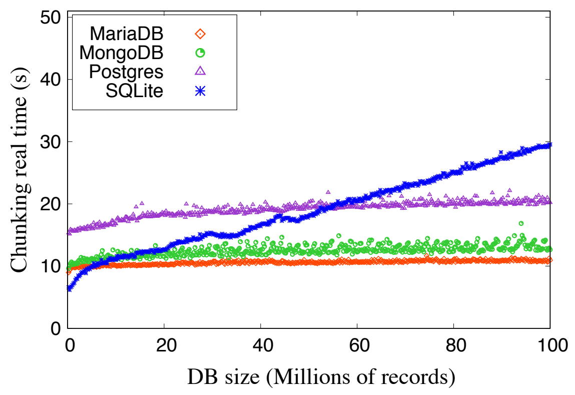
In this experiment we used 500 CSV files of size 100 MB each and measured the file chunking and reconstruction times as function of the number of records in the database. Each file was divided into 204,800 chunks of size 512 B and each chunk was sent to the database.

3.4.2 MariaDB

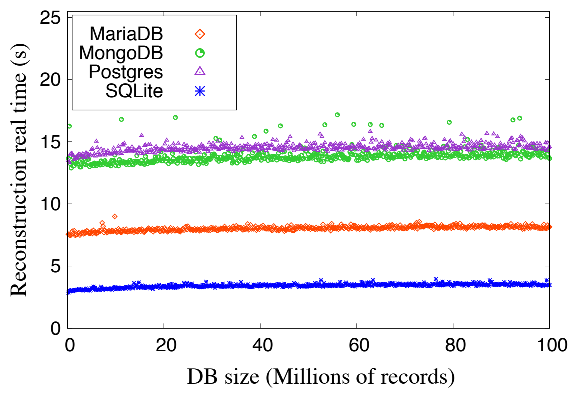
MariaDB is the most popular Relational Database Management System. We used the following data types: INT, CHAR, VARCHAR, TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT, BLOB (for binary data). Two tables were created: **files** and **chunks**. MariaDB has two major storage engines, the default XtraDB / InnoDB and ARIA. There is also MyISAM, a non-transactional storage engine, also tested for comparison, not as a candidate storage engine. ARIA is faster for reads and writes, XtraDB / InnoDB is faster for reads. Each ARIA table is stored on disk in three files: the frm file stores the table format, the data file has an MAD extension, and the index file has an MAI extension. MariaDB is easy to work with, is feature rich, secure, scalable, and powerful. It has some disadvantages as known limitations, reliability issues, and stagnated development. MariaDB is used for distributed operations, for high security, for web-sites and web-applications, and for custom solutions. Appendix A.1 contains implementation details, including SQL command for creating the tables A.1.1, inserting file information and chunks into the database A.1.2, and reconstructing the files A.1.3.

3.4.3 MongoDB

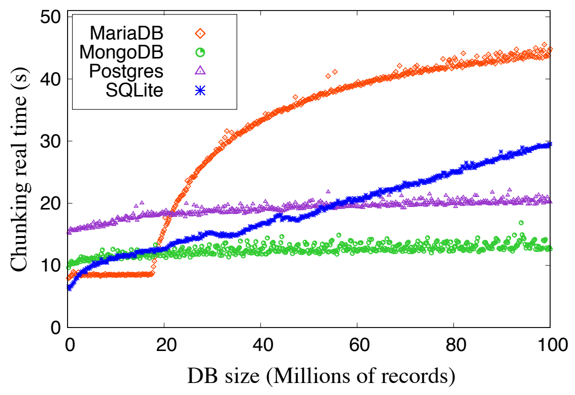
MongoDB is a NoSQL Database Management System. It uses JSON-like documents (rows in mysql) with schemas. The following data types were used: string (must be UTF-8 valid), integer, binary data (used to store binary data). Instead of tables, MongoDB uses collections. The following collections were used in our experiments: **files** and **chunks**. MongoDB has two storage engines: the default WiredTiger and MMAPv1. MMAPv1 is better for heavy reads, WiredTiger is better for heavy writes but has some known issues with ext4. MMAPv1 is better choice for deduplication. MongoDB stores data as documents in a binary representation called BSON (Binary JSON). MongoDB documents tend to have all data for a given record in a single document, whereas in a relational database information for a given record is usually spread across many tables. The database files are broken up into 2 GB extents. Documents in a collection are stored as a doubly linked list within extents. Index data is also stored in these files, but they are stored as B-Trees. Appendix A.2 contains implementation details, including command for creating the collections A.2.1, inserting file information and chunks into the database A.2.2, and reconstructing the files A.2.3.



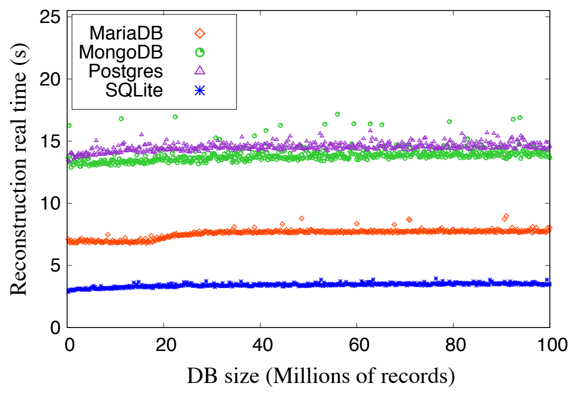
(a) Chunking using Aria storage engine.



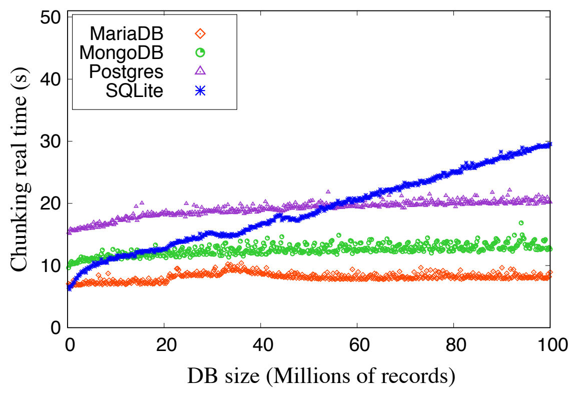
(b) Reconstruction using Aria storage engine.



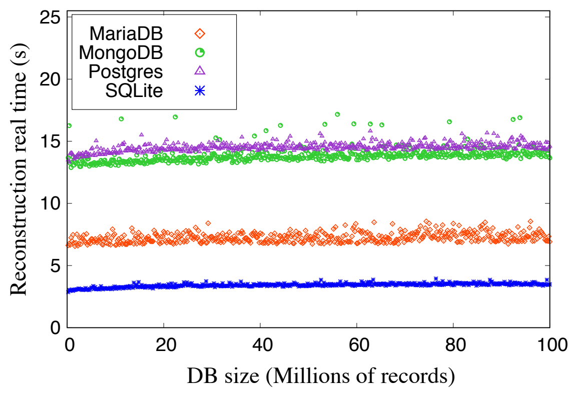
(c) Chunking using Innodb storage engine.



(d) Reconstruction using Innodb storage engine.



(e) Chunking using MyIsam storage engine.

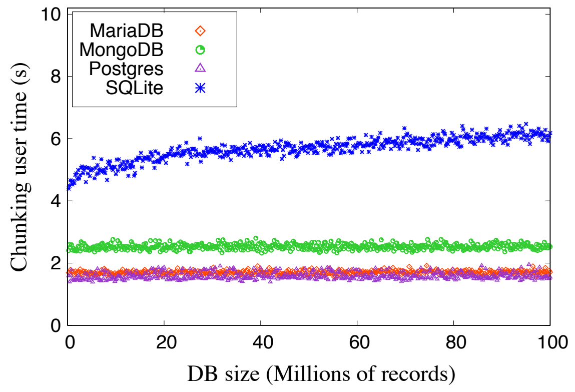


(f) Reconstruction using MyIsam storage engine.

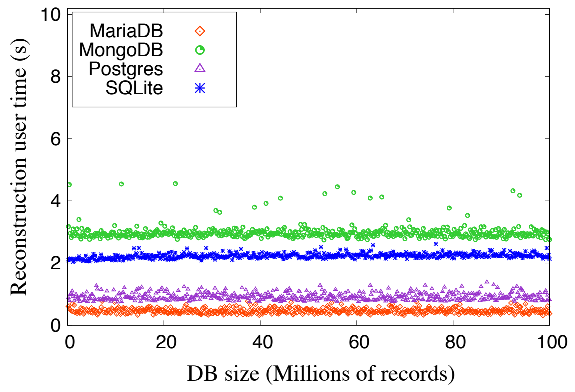
Figure 3–12: Chunking and reconstruction real times for MariaDB using ARIA engine, MariaDB using Innodb engine, MariaDB using MyISAM engine, MongoDB, PostgreSQL, and SQLite.

3.4.4 Postgres

Postgres is an Object-relational Database Management System. Provides the following data types: int, char, varchar, text , bytea (binary data). Two tables



(a) Chunking user times.



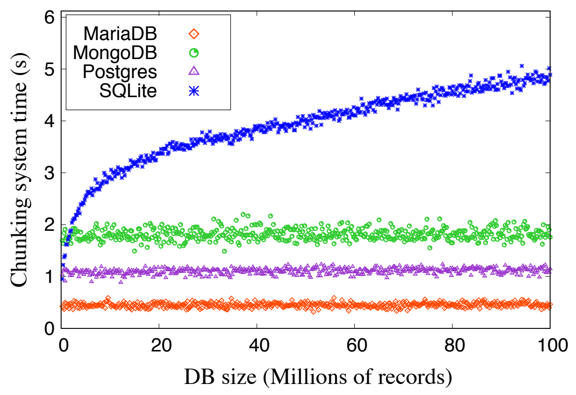
(b) Reconstruction user times.

Figure 3–13: Chunking and reconstruction user times for MariaDB, MongoDB, PostgreSQL, and SQLite.

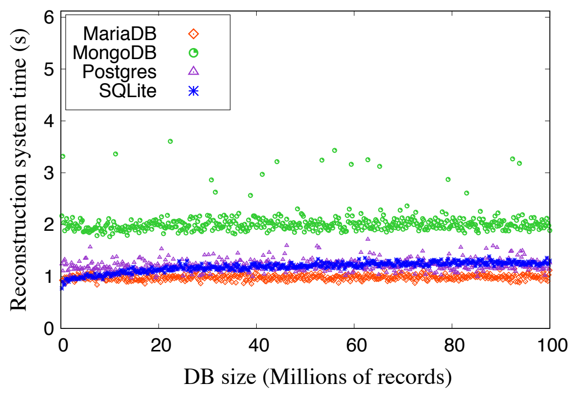
were created: **files** and **chunks**. Postgres supports and uses just only one storage engine - PostgreSQL. The configuration and data files used by a database cluster are stored together within the cluster's data directory. For each database there is a subdirectory, this is the default location for the database's files. Each table and index is stored in a separate file. Tables and indexes are divided into segments of size 1 GB. Advantages of Postgres are: Open-source, Strong community, strong third-party support, extensible, objective. It has some disadvantages as performance, popularity, and hosting. Postgres is used when data integrity is an absolute necessity, for complex, custom procedures, complex designs, and when integration is needed. Postgres is not recommended if fast read operations, or replication are required. Appendix A.3 contains implementation details, including SQL command for creating the tables A.3.1, inserting file information and chunks into the database A.3.2, and reconstructing the files A.3.3.

3.4.5 SQLite

SQLite is a very powerful, embedded relational database management system. The following data types are available: INT, CHAR, TEXT (stored using the database encoding), BLOB (stored exactly as it was input). Two tables were created: **files** and **chunks**. SQLite stores the data server less, in a single, stable



(a) Chunking system times.



(b) Reconstruction system times.

Figure 3–14: Chunking and reconstruction system times for MariaDB, MongoDB, PostgreSQL, and SQLite.

cross-platform database file. It is compact, has manifest typing, variable-length records, and readable source code. SQL statements compile into virtual machine code. SQLite has Zero-configuration. Some advantages of SQLite are: it is file based (the entire database consists of a single file on the disk), standards-aware (uses SQL). It is great for developing and even testing (simplicity of working with a single file and a linked C-based library). However, has no user management, lack of possibility to tinker with for additional performance, DBMS allows only one single write operating to take place at any given time. SQLite is used for single-user local applications, and for testing. It is not recommended for multi-user applications and for applications requiring high write volumes. Appendix A.4 contains implementation details, including SQL command for creating the tables A.4.1, inserting file information and chunks into the database A.4.2, and reconstructing the files A.4.3.

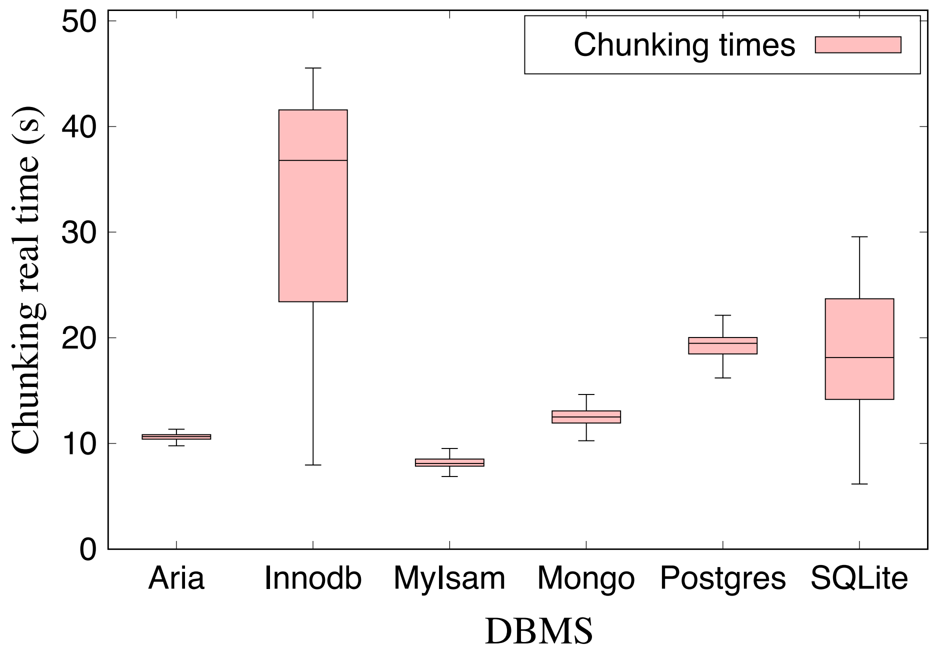
Table 3–3: Database Management System (DMS) comparison. Chunking and reconstruction times for files of size 100 MB using databases containing between 50 and 100 million records.

DMS	**Engine**	**Type**	**Storage**	**Chunking time (s)**	**Reconstruct time (s)**
MariaDB	ARIA	Relational	3 files per table	10.4	8.5
MariaDB	InnoDB	Relational	3 files per table	37.2	7.8
MariaDB	MyISAM	Relational	3 files per table	8.2	7.3
MongoDB	MMAPv1	Not relational	2 GB extents	13.0	14.0
Postgres	PostgreSQL	Relational	1 GB segments	20.1	14.6
SQLite		Relational	1 file per database	24.0	3.5

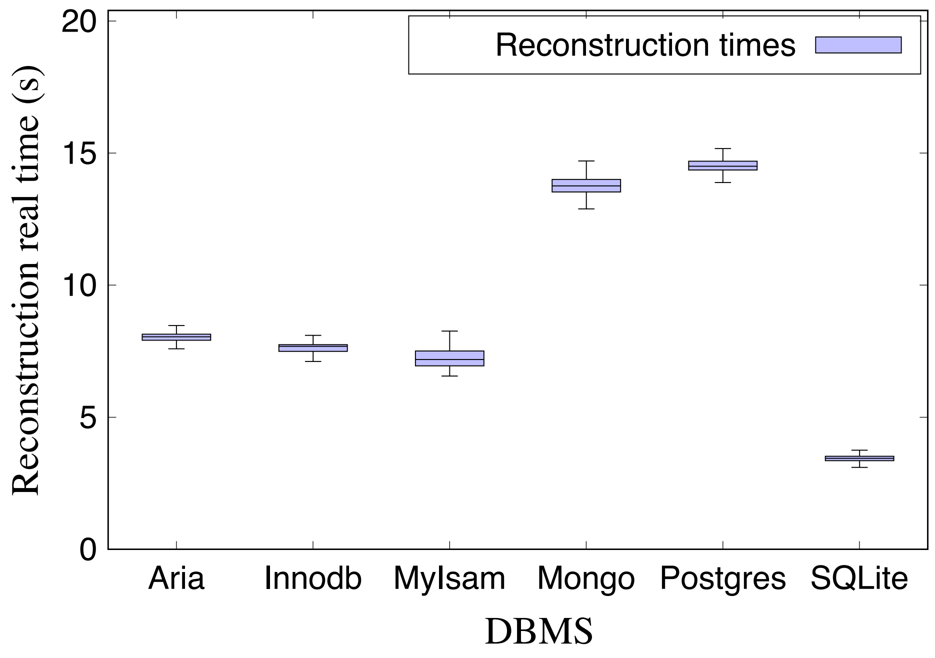
3.4.6 DBMS Comparison Results

Table 3–3 contains our results from the Database Management System (DMS) comparison.

The best chunking (writing) times were obtained for MariaDB with ARIA and MyISAM storage engines. The best reconstruction (read) time was obtained for SQLite, followed by MariaDB with InnoDB engine. Because SQLite is not an option for our deduplicated and distributed file system, we selected MariaDB with InnoDB storage engine for our implementation.

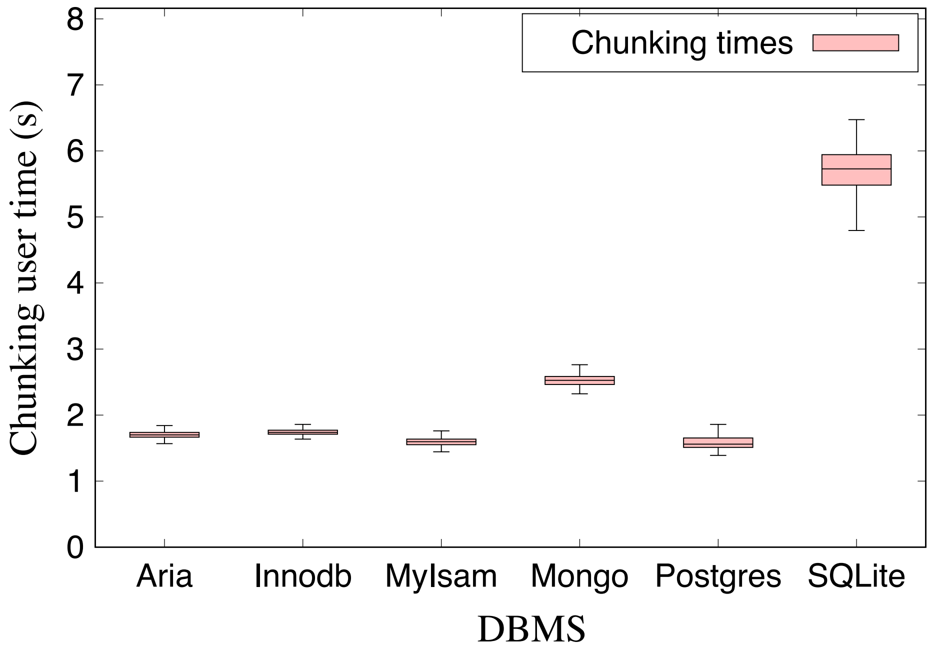


(a) Chunking real times.

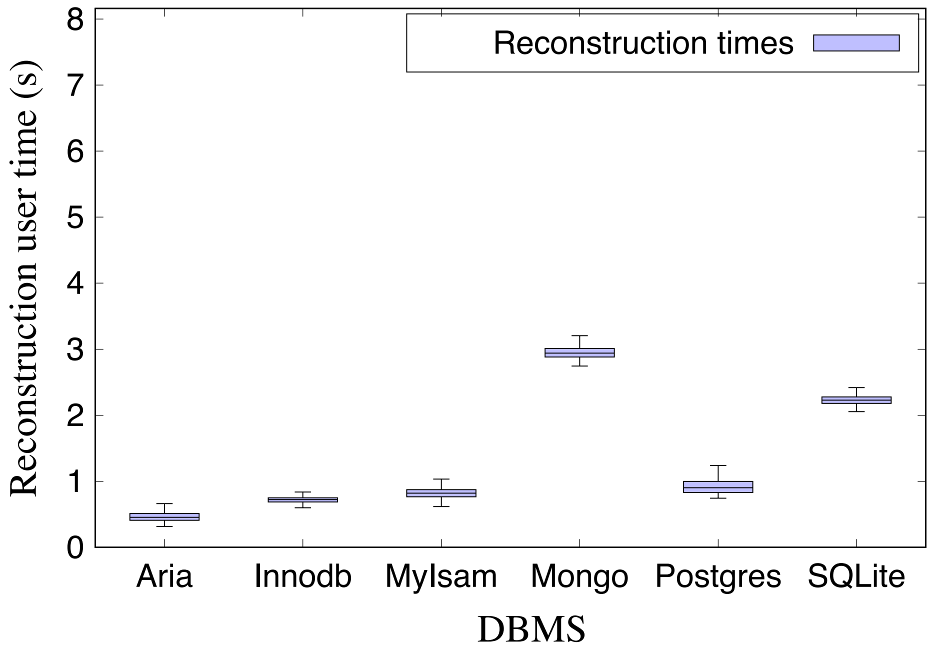


(b) Reconstruction real times.

Figure 3–15: Comparative box plots for chunking and reconstruction real times for MariaDB using ARIA engine, MariaDB using InnoDB engine, MariaDB using MyISAM engine, MongoDB, PostgreSQL, and SQLite.

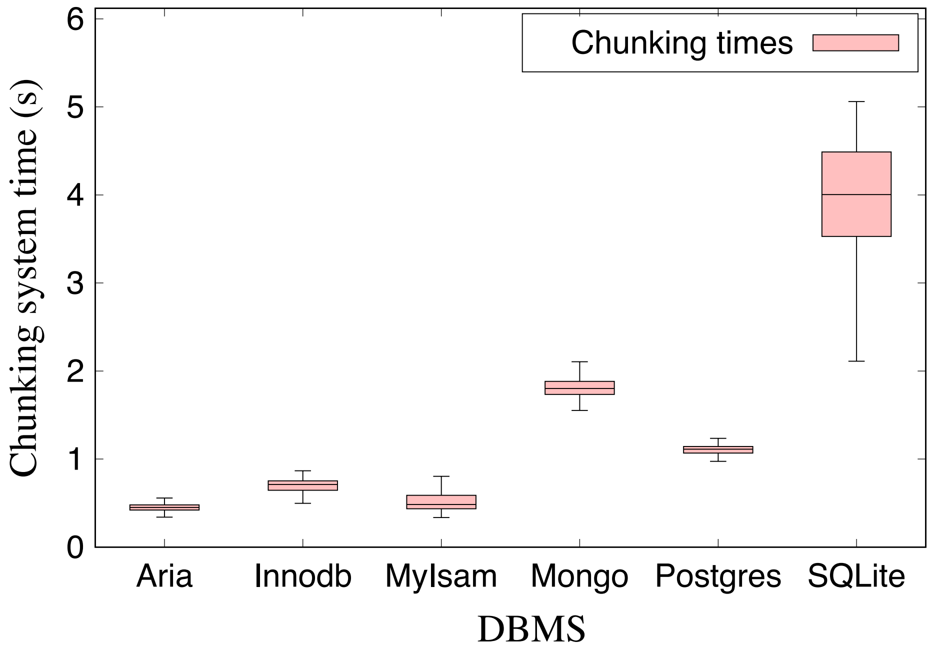


(a) Chunking user times.

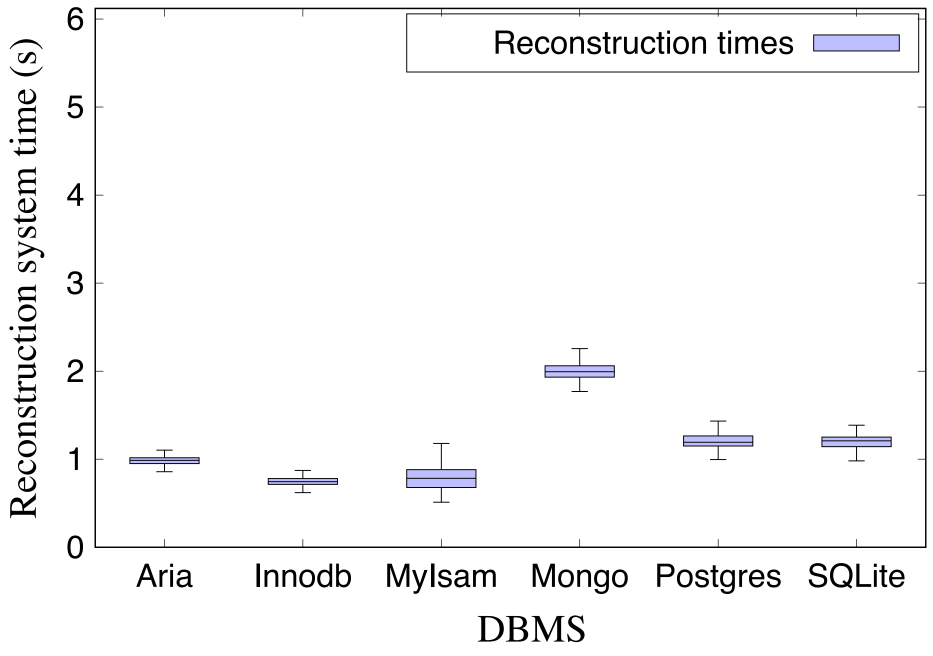


(b) Reconstruction user times.

Figure 3–16: Comparative box plots for chunking and reconstruction user times for MariaDB using ARIA engine, MariaDB using InnoDB engine, MariaDB using MyISAM engine, MongoDB, PostgreSQL, and SQLite.



(a) Chunking system times.



(b) Reconstruction system times.

Figure 3–17: Comparative box plots for chunking and reconstruction system times for MariaDB using ARIA engine, MariaDB using InnoDB engine, MariaDB using My-ISAMengine, MongoDB, PostgreSQL, and SQLite.

3.5 HD2FS Implementation

In order to improve data storage capacity and efficiency in distributed file system, using the results obtained from the preliminary study and the experiments done to find the best Database Management System, the design and implemention of the new deduplicated and distributed file system, HD2FS, was done by integrating deduplication into Hadoop Distributed File System (HDFS), as Figure 3–19 shows. HDFS is used to solve the storage problem of huge data, but does not provide a handling mechanism of duplicate files.

Hadoop 2.9.0 source code downloaded from

http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-2.9.0

was modified to integrate deduplication. The following new keys were defined in hdfs-site.xml:

- *dfs.dedupchunksize*, used to set the deduplication chunk size, with a default value of 512 corresponding to 512 B deduplication chunk size.
- *dfs.dedupchunk.location.db*, used to set where the chunk content will be stored, in the deduplication database (*true*, default) or stored as files (*false*).
- *dfs.dedupchunk.dir*, used to set the directory for the chunks in the case they will be stored as files, with the default value */home/hadoop/hadoopdata/dedupchunk/*.

For writing a file to HD2FS, the stream containing the file content is used as input in one of the following chunking methods:

dedupFromByteArray(Connection, OutputStream, byte, long, int), and

dedupFromByteArray(OutputStream, byte, long, int).

The first method is used if the chunk content will be stored in the database, and the second in the case the chunk content will be stored as files.

The stream is processed by dividing it into chunks, computing the corresponding chunkId by using the SHA-256 hash function, and writing the obtained chunkId to the file recipe. The file recipe will be the new file in HD2FS, sent to the

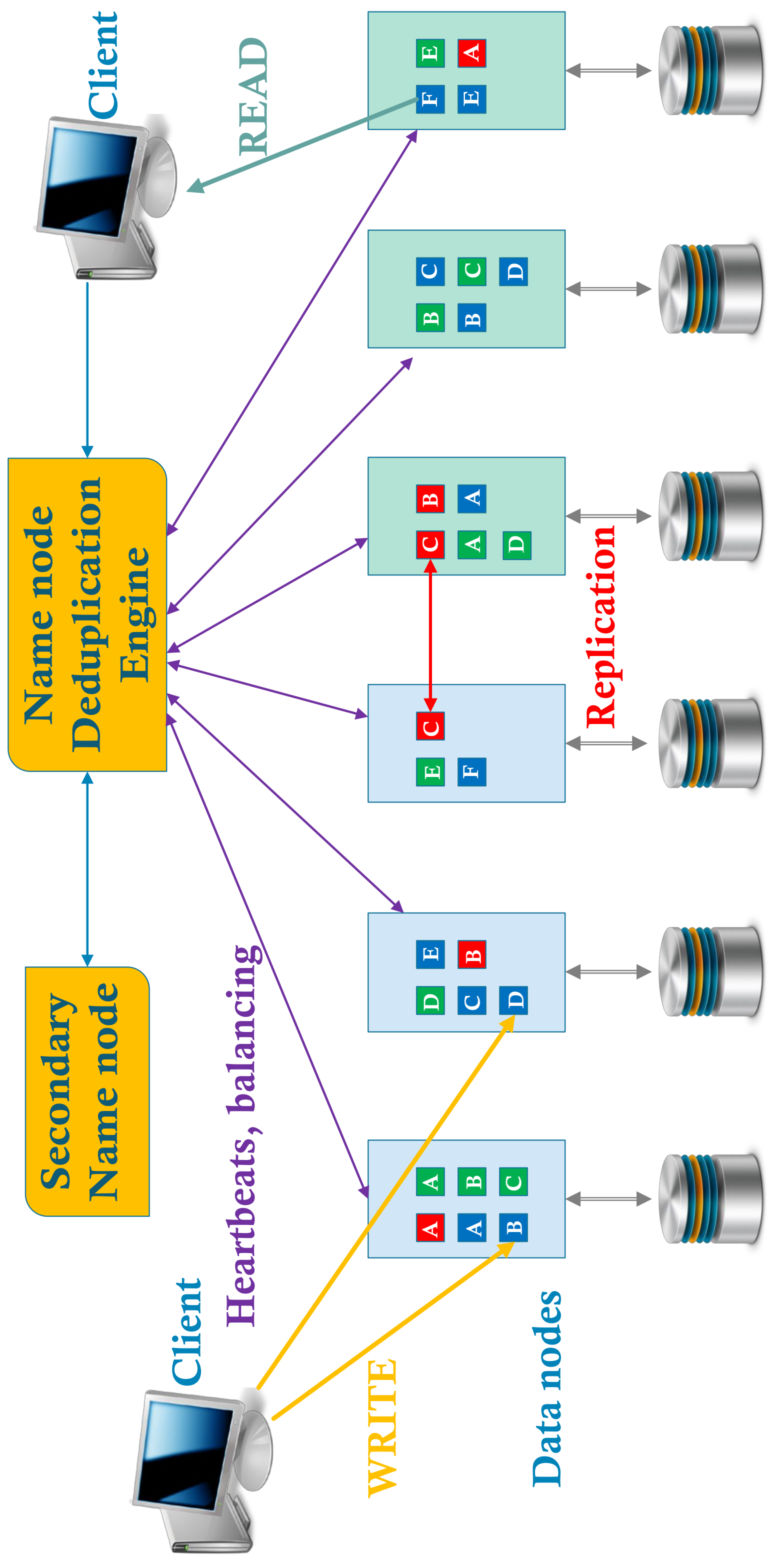


Figure 3–18: The modified Hadoop Distributed File System to include deduplication.

data nodes in HDFS blocks. After computing the chunkId, the stream content is sent to the database, or stored as files, respectively. The deduplication chunk size *dedupchunksize* can be modified in hdfs-site.xml, using the new key *dfs.dedupchunksize*. Appendix E.1 contains the working hdfs-site.xml for HD2FS. For reading a file from HD2FS, the corresponding reconstruction method will be used, depending where the chunk contents are stored in the file system:

reconstructFromStreamToFile(Connection, InputStream, PathData), and

reconstructFromStreamToFile(dedupChunkFolder, dedupChunkSize, InputStream, PathData).

The first method is used if the chunk content was stored in the database, and the second in the case the chunk content was stored as files. In both cases, the corresponding file recipe is obtained from HD2FS data nodes. The chunk ID's are read one by one and the corresponding content is obtained from the database or from the stored file. In this way, using the unique data stored, the file is reconstructed.

The chunking methods can be found in Appendix C.1 and D.1, and the reconstruction methos in Appendix C.2 and D.2, respectively.

3.5.1 Deduplication and Reconstruction Algorithms

Algorithm 1 and Algorithm 2 contain the deduplication algorithms for the two cases, and Algorithm 3 and Algorithm 4 contain the reconstruction algorithms used in our implementation. MariaDB with InnoDB storage engine was selected for the implementation, and for replication purposes, MariaDB Galera Cluster was used.

Algorithm 1 Deduplication algorithm - chunk content stored in the database.

```
procedure DEDUP(connectMariaDB,out,b,dedupChunkSize,numBytes)
    sql ← "INSERT IGNORE INTO chunk(chunkId,numBytes,count,content)
    VALUES(?,?,1,?) ON DUPLICATE KEY UPDATE count=count+1;"
    chunkSize ← dedupChunkSize
    bytesToChunk ← numBytes
    while $bytesToChunk > 0$ do
        if $chunkSize < bytesToChunk$ then
            bytesToChunkNext ← chunkSize
        else
            bytesToChunkNext ← bytesToChunk
        end if
        chunk ← Arrays.copyOfRange(b, numBytes - bytesToChunk,
        numBytes - bytesToChunk + bytesToChunkNext)
        chunkId ← DigestUtils.sha256Hex(chunk)
        sqlInsert ← connectMariaDB.prepareStatement(sql)
        sqlInsert ← setString(1, chunkId)
        sqlInsert ← setInt(2, bytesToChunkNext)
        sqlInsert ← setBinaryStream(3, new ByteArrayInputStream(chunk),
        bytesToChunkNext)
        out ← write(chunkId.getBytes(), 0, chunkId.getBytes().length)
        try
            sqlInsert ← executeUpdate()
        finally
            sqlInsert ← close()
        end try
        $bytesToChunk \leftarrow bytesToChunk - bytesToChunkNext$
    end while
end procedure
```

Algorithm 2 Deduplication algorithm - chunk stored as file in the file system.

```
1: procedure DEDUP(out,b,dedupChunkSize,numBytes)
2:     $chunkSize \leftarrow dedupChunkSize$
3:     $bytesToChunk \leftarrow numBytes$
4:     while $bytesToChunk > 0$ do
5:         if $chunkSize < bytesToChunk$ then
6:             $bytesToChunkNext \leftarrow chunkSize$
7:         else
8:             $bytesToChunkNext \leftarrow bytesToChunk$
9:         end if
10:        chunk ← Arrays.copyOfRange(b, numBytes - bytesToChunk,
11:        numBytes - bytesToChunk + bytesToChunkNext)
12:        chunkId ← DigestUtils.sha256Hex(chunk)
13:        bytesToChunkNext)
14:        out ← write(chunkId.getBytes(), 0, chunkId.getBytes().length)
15:        $bytesToChunk \leftarrow bytesToChunk - bytesToChunkNext$
16:        $outputStream \leftarrow FileOutputStream(dedupChunkFolder + chunkId)$
17:        $outputStream \leftarrow write(chunk)$
18:    end while
19: end procedure
```

Algorithm 3 Reconstruction algorithm - chunk content stored in the database.

```
procedure RECONSTRUCT($connectMariaDB, in, target$)
    $sqlStatement \leftarrow connectMariaDB$.createStatement()
    $newFile \leftarrow target$
    $out \leftarrow$ new FileOutputStream($newFile$.toFile())
    try
        $buf \leftarrow$ new byte[64]
        $bytesRead \leftarrow$ in.read($buf$)
        $chunkId \leftarrow$ new String($buf$)
        while $bytesRead > 0$ do
            $sql \leftarrow$ “SELECT content, numBytes from chunk where chunkId = ’”
            + chunkId + “’ LIMIT 1;”
            $statement \leftarrow connectMariaDB$.prepareStatement($sql$)
            $chunkContentAndSize \leftarrow statement$.executeQuery()
            $chunkContentAndSize \leftarrow$ next()
            $inContent \leftarrow chunkContentAndSize$.getBinaryStream(“content”)
            $chunkSize \leftarrow chunkContentAndSize$.getInt(“numBytes”)
            $chunkByte \leftarrow$ new byte[$chunkSize$]
            while $inContent$.read($chunkByte$) >= 0 do
                $out \leftarrow$ write($chunkByte$)
            end while
            $bytesRead \leftarrow$ in.read($buf$)
            $chunkId \leftarrow$ new String($buf$)
        end while
        $out \leftarrow$ close()
        $in \leftarrow$ close()
    finally
        $sqlStatement \leftarrow$ close()
    end try
end procedure
```

Algorithm 4 Reconstruction algorithm - chunk stored as file in the file system.

```
1: procedure RECONSTRUCT(dedupChunkFolder,dedupChunkSize,in,target)
2:     newFile ← target
3:     out ← new FileOutputStream(newFile.toFile())
4:     try
5:         buf ← new byte[64]
6:         bytesRead ← in.read(buf)
7:         chunkId ← new String(buf)
8:         while bytesRead > 0 do
9:             chunkStream ← new FileInputStream(dedupChunkFolder + chunkId);
10:            chunkByte ← new byte[chunkStream.available()];
11:            while inContent.read(chunkByte) >= 0 do
12:                out ← write(chunkByte)
13:            end while
14:            bytesRead ← in.read(buf)
15:            chunkId ← new String(buf)
16:        end while
17:        out ← close()
18:        in ← close()
19:    end try
20: end procedure
```

3.5.2 MariaDB Galera Cluster

After analyzing the chunking (writing) times in Figure 3–13a , Figure 3–13c, and Figure 3–16a, the reconstruction (read) times in Figure 3–13b, Figure 3–13d, and Figure 3–16b, MariaDB showed the best performance. Therefore, MariaDB Galera Cluster was selected to store and replicate the chunks in our new deduplicated and distributed file system HD2FS.

As Figure 3–19 shows, the **Deduplication Engine (DE)** in HD2FS is running on the name node. During the deduplication process, the file recipes are stored in HD2FS blocks in the data nodes, and DE uses MariaDB cluster to store the unique chunk contents in the database. When reading a file, the file recipe is obtained from HD2FS data nodes. The chunk ID's are read one by one and the corresponding chunk is obtained from the database. In this way, using the unique data stored in the database, the file is reconstructed.

The deduplication database *dedup* contains the following table:

$$chunk(chunkId, numBytes, count, content)$$

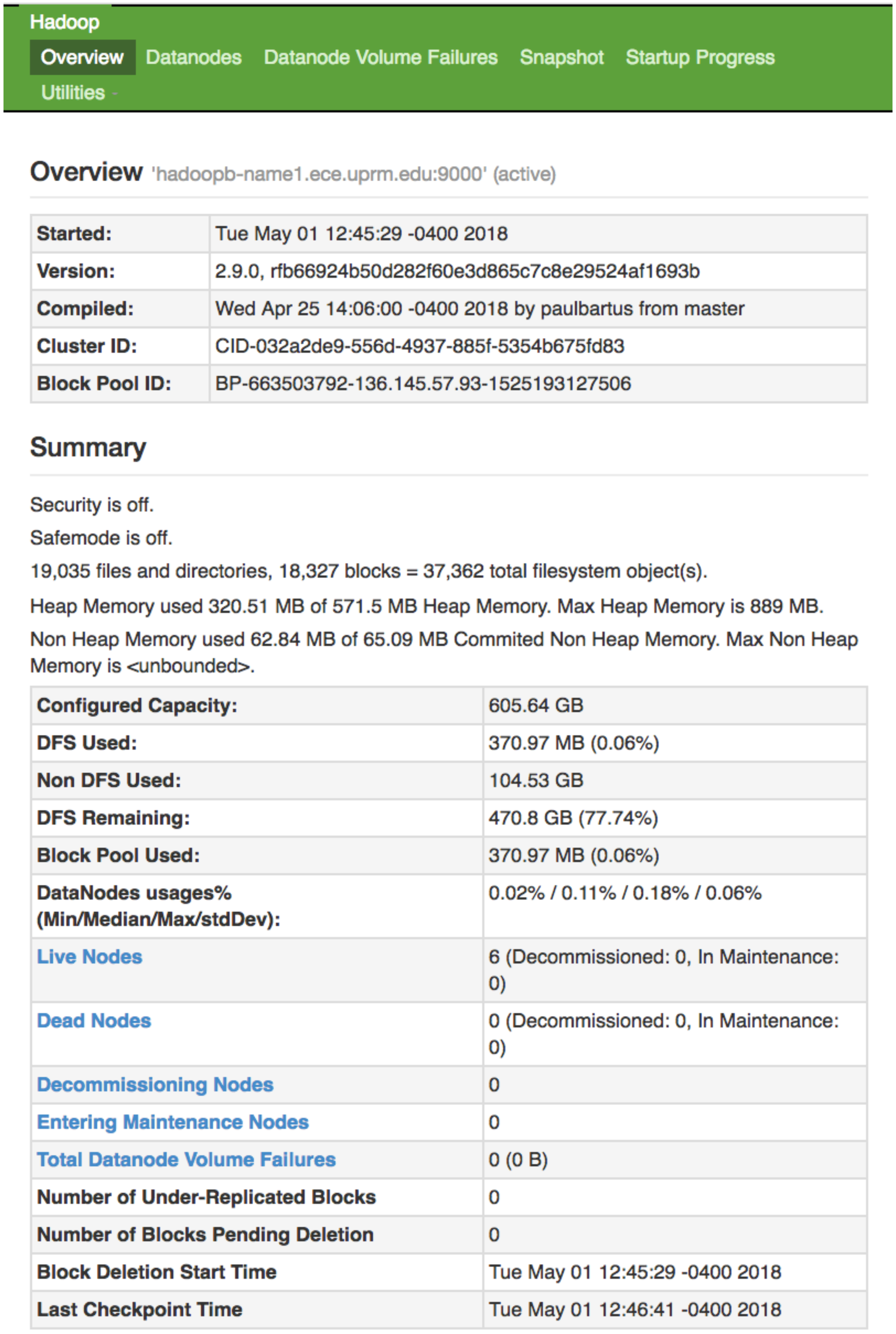
where $chunkId$ is the primary key, $numBytes$ represents the number of bytes in the chunk (the chunk size), $count$ contains the number of times the chunk is referenced, and $content$ stores the binary content of the chunk as a BLOB data type. If the chunk index already exists in the database, the corresponding reference count is incremented by one. If the chunk index is not found, a new record is added with the corresponding reference count equal one.

Table 3–4: HD2FS cluster with deduplication.

Host	IP Address	Node Role	CPU	Memory (GB)	Storage (GB)
hadoopb-name1.ece.uprm.edu	136.145.57.93	Name Dedup engine	8	32	200 (SSD)
hadoopb-name2.ece.uprm.edu	136.145.57.186	Secondary Name	4	16	200 (SSD)
hadoopb-data1.ece.uprm.edu	136.145.57.187	Data	4	16	200 (SSD)
hadoopb-data2.ece.uprm.edu	136.145.59.104	Data	4	16	200 (HDD)
hadoopb-data3.ece.uprm.edu	136.145.59.105	Data	2	4	50 (HDD)
hadoopb-data4.ece.uprm.edu	136.145.59.106	Data	2	4	50 (HDD)
hadoopb-data5.ece.uprm.edu	136.145.59.107	Data	2	4	50 (HDD)
hadoopb-data6.ece.uprm.edu	136.145.59.108	Data	2	4	50 (HDD)

3.5.3 HD2FS Cluster

Table 3–4 shows the HD2FS cluster configuration. For better performance, as Table 3–4 shows, the MariaDB Galera cluster runs on three servers having solid state drives (SSD): the name node, the secondary name node, and also on a data node. Figure 3–20 shows a screenshot of our new deduplicated and distributed file system HD2FS overview page and Figure 3–21 shows information about the data nodes.



Overview 'hadoopb-name1.ece.uprm.edu:9000' (active)

Started:	Tue May 01 12:45:29 -0400 2018
Version:	2.9.0, rfb66924b50d282f60e3d865c7c8e29524af1693b
Compiled:	Wed Apr 25 14:06:00 -0400 2018 by paulbartus from master
Cluster ID:	CID-032a2de9-556d-4937-885f-5354b675fd83
Block Pool ID:	BP-663503792-136.145.57.93-1525193127506

Summary

Security is off.

Safemode is off.

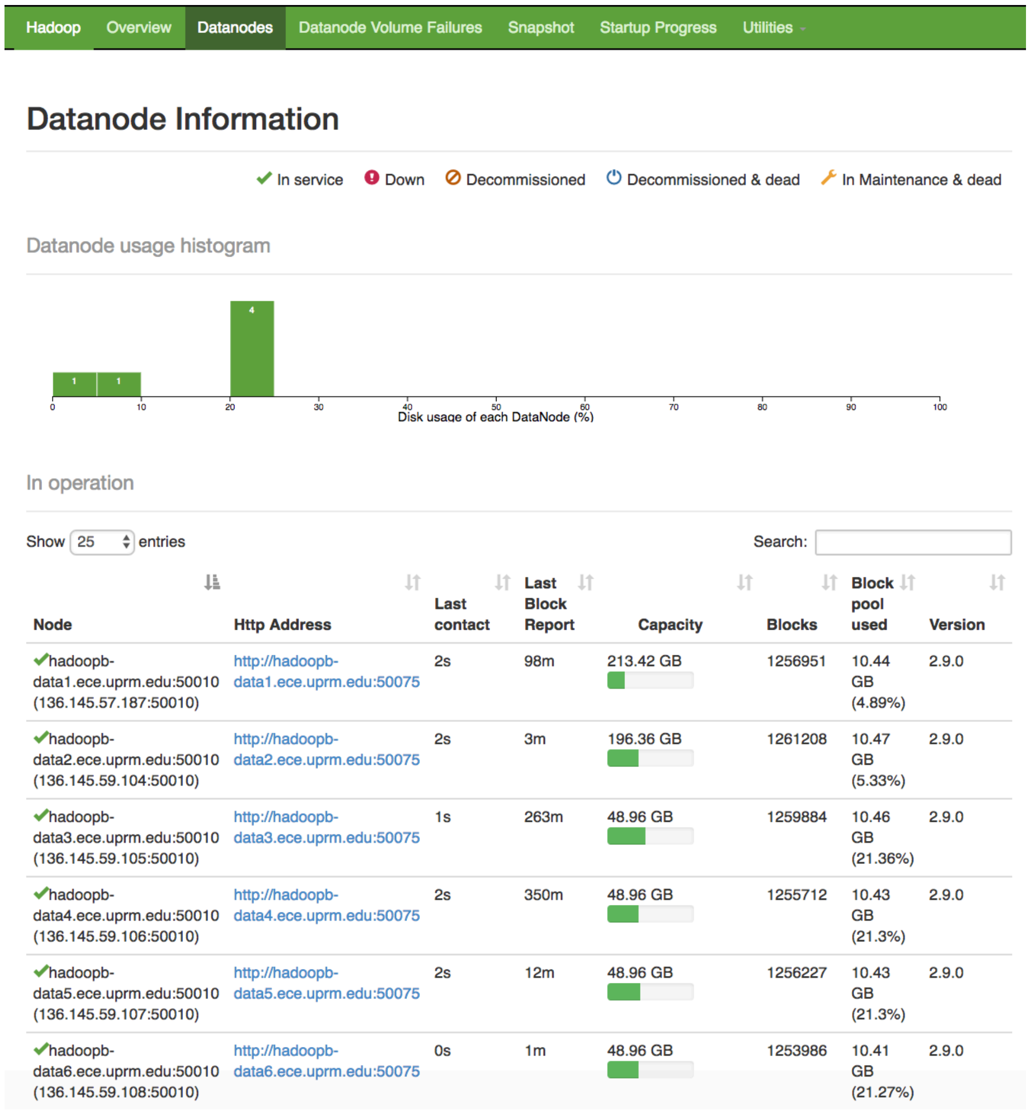
19,035 files and directories, 18,327 blocks = 37,362 total filesystem object(s).

Heap Memory used 320.51 MB of 571.5 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 62.84 MB of 65.09 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	605.64 GB
DFS Used:	370.97 MB (0.06%)
Non DFS Used:	104.53 GB
DFS Remaining:	470.8 GB (77.74%)
Block Pool Used:	370.97 MB (0.06%)
DataNodes usages% (Min/Median/Max/stdDev):	0.02% / 0.11% / 0.18% / 0.06%
Live Nodes	6 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	Tue May 01 12:45:29 -0400 2018
Last Checkpoint Time	Tue May 01 12:46:41 -0400 2018

Figure 3–19: HD2FS cluster overview.



Datanode Information

Datanode usage histogram

In operation

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓hadoopb-data1.ece.uprm.edu:50010 (136.145.57.187:50010)	http://hadoopb-data1.ece.uprm.edu:50075	2s	98m	213.42 GB	1256951	10.44 GB (4.89%)	2.9.0
✓hadoopb-data2.ece.uprm.edu:50010 (136.145.59.104:50010)	http://hadoopb-data2.ece.uprm.edu:50075	2s	3m	196.36 GB	1261208	10.47 GB (5.33%)	2.9.0
✓hadoopb-data3.ece.uprm.edu:50010 (136.145.59.105:50010)	http://hadoopb-data3.ece.uprm.edu:50075	1s	263m	48.96 GB	1259884	10.46 GB (21.36%)	2.9.0
✓hadoopb-data4.ece.uprm.edu:50010 (136.145.59.106:50010)	http://hadoopb-data4.ece.uprm.edu:50075	2s	350m	48.96 GB	1255712	10.43 GB (21.3%)	2.9.0
✓hadoopb-data5.ece.uprm.edu:50010 (136.145.59.107:50010)	http://hadoopb-data5.ece.uprm.edu:50075	2s	12m	48.96 GB	1256227	10.43 GB (21.3%)	2.9.0
✓hadoopb-data6.ece.uprm.edu:50010 (136.145.59.108:50010)	http://hadoopb-data6.ece.uprm.edu:50075	0s	1m	48.96 GB	1253986	10.41 GB (21.27%)	2.9.0

Figure 3–20: HD2FS cluster data nodes.

3.6 Summary

In this chapter, a preliminary study and a Database Management System comparison in order to design and implement our new deduplicated and distributed file system HD2FS was presented. Deduplication was integrated to Hadoop Distributed File System (HDFS) and MariaDB Galera cluster was used to store and replicate the data. In the next chapters, a study of performance and the advantages of using HD2FS to store data in cloud and to improve storage management of genomic data will be presented.

Chapter 4
USING HD2FS TO EFFICIENTLY STORE DATA IN CLOUD

This chapter presents the use of the deduplication storage system HD2FS, to improve data storage capacity and efficiency in distributed file systems. Instead of storing multiple copies of the same data, our deduplication system will store only the data that is different along with a map to reconstruct all data files. In order to demonstrate it's feasibility and performance, we present a study of the relation between the number of different consecutive versions (distributions) of popular source codes like GCC, Linux kernels, database log files, versus the deduplication ratio, and the percentage of duplicate chunks, respectively.

4.1 Experiments

The experiments were designed to study the performance of HD2FS using real world data sets.

4.1.1 Experimental Setup

This experiments used consecutive versions (distributions) of the GCC source codes, Linux Kernel source codes, and InnoDB database log files. The source codes were written to HD2FS cluster. As mentioned in Chapter 3, Table 3–4 shows the HD2FS cluster configuration used in our experiments. The name node is a DELL PowerEdge R220 server equipped with an Intel(R) Xeon(R) CPU E3-1286 v3 @ 3.70 GHz 8 core CPU, 32 GB of RAM, 400 GB SSD, and 8 TB of extra storage using regular hard drives. The experiments were conducted on the SSD as primary storage. The secondary name node and one data node are AMD A10-7850K Radeon R7, 12

Compute Cores 4C+8G @ 3 GHz 4 core CPU, 16 GB of RAM, 200 GB SSD. All the remaining data nodes are virtual servers. One virtual server has 4 cores, 16 GB of RAM, 200 GB HDD, and the remaining ones have 2 cores, 4 GB of RAM, 50 GB HDD. The operating system on all servers in the HD2FS cluster was Ubuntu 17.10 LTS with kernel Linux 4.13.0-43..

4.1.2 Data Sets

Source codes from 84 versions of GCC were downloaded from the GNU Archives http://ftp.gnu.org/gnu/gcc/. Also, the sorce codes from 201 versions of Linux kernel 2.x, the source codes from 75 versions of Linux kernel 3.x, and the source codes from 75 versions of Linux kernel 4.x were downloaded from a mirror of the Linux Archives http://ftp.kernel.org/pub/linux/kernel/. During the writes of the previous source codes to HD2FS, 16 Innodb database log files were saved every 1.5 hours. Table 4–1 contains the data sets used for HD2FS performance evaluation.

Table 4–1: Data sets used for HDFS with deduplication performance evaluation.

Name	**Versions**	**Avg size (MB)**	**Total size (GB)**	**Comp ratio**	**Dedup ratio**	**DupCh (%)**
GCC	84	288.1	23.6	7.0	4.7	79
Linux kernels 2.x	201	173.2	34.0	5.1	10.2	90
Linux kernels 3.x	75	494.0	36.2	5.3	61.6	98
Linux kernels 4.x	50	731.1	35.7	5.6	21.6	95
InnoDB logs	16	2048.0	32.0		8.2	88

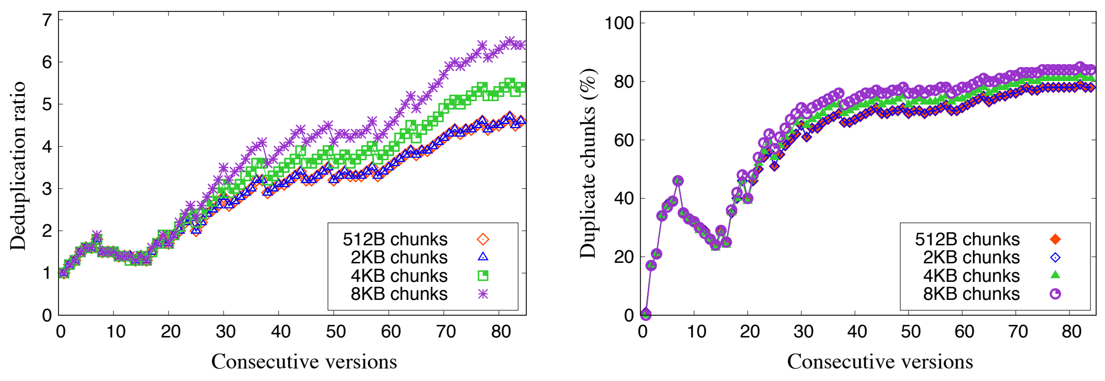
4.1.3 Results

The experiments were done using the following chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. For reference, the experiments were also run on the original unmodified HDFS cluster. The first experiment was with the GCC source codes, followed by the Linux kernel versions, considering the source codes for Linux kernel 2.x, 3.x, and 4.x separately. These experiments were performed to evaluate the deduplication ratio and the percentage of duplicate chunks obtained with HD2FS.

The next experiments were performed to evaluate the time performances of HD2FS and the scalability of our new system.

GCC source codes

The first set of experiments used 84 consecutive versions of the GCC source codes downloaded from their website. The source codes were written to HD2FS cluster. The experiment was repeated for different chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. As Figure 4–1a shows, the deduplication ratio depends on the chunk size, and is ranging between 4.5 and 6.5 as more source codes are added. Also, as Figure 4–1b shows, the percentage of duplicate chunks also depends on the deduplication chunk size, and is ranging between 78 % and 84 %.



(a) Deduplication ratio analysis for GCC versions.

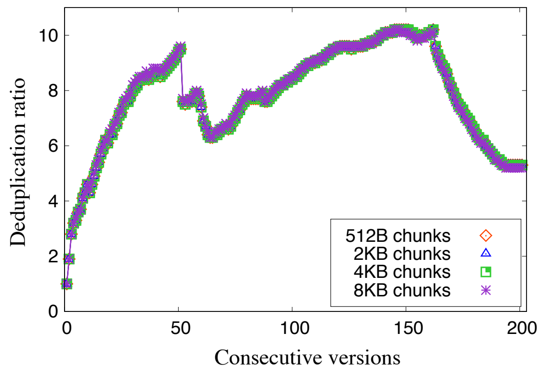
(b) Percentage of duplicate chunks analysis for GCC versions.

Figure 4–1: Deduplication ratio and percentage of duplicate chunks analysis for GCC versions.

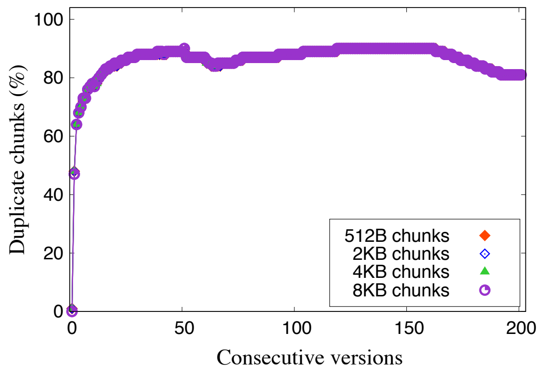
Linux kernel 2.x source codes

The second set of experiments used 201 consecutive versions of the Linux kernel 2.x source codes downloaded from their website. The source codes were written to HD2FS cluster. The experiment was repeated for different chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. As Figure 4–2a shows, the deduplication ratio does not depend on the chunk size, and is ranging between 4.5 and 10.2 as more source codes are added. Also, as Figure 4–2b shows, the percentage of duplicate chunks does not depend on the chunk size, neither. After adding all of the 201 Linux kernel

2.x source codes, the percentage of duplicate chunks is ranging between 80 % and 90 %. In the case when the current version has lots of changes compared to the previous versions, the deduplication ration and the percentage of duplicate chunks are decreasing, followed by future increases when more and more source code versions are added.



(a) Deduplication ratio analysis for Linux Kernels 2.x.



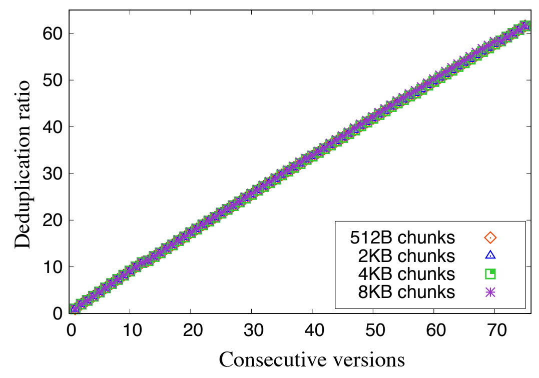
(b) Percentage of duplicate chunks analysis for Linux Kernels 2.x.

Figure 4–2: Deduplication ratio and percentage of duplicate chunks analysis for Linux Kernels 2.x distributions.

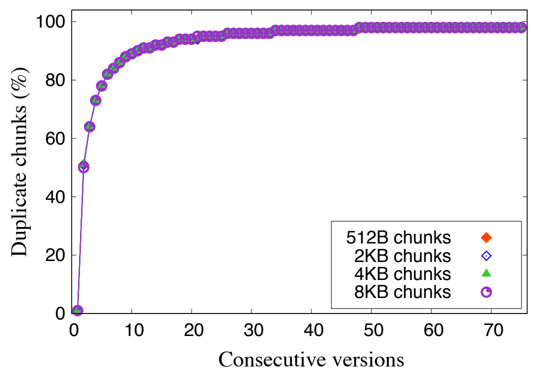
Linux kernel 3.x source codes

The third set of experiments used 75 consecutive versions of the Linux kernel 3.x source codes downloaded from their website. The source codes were also written to HD2FS cluster. The experiment was repeated for different chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB.

As Figure 4–3a shows, the deduplication ratio does not depend on the chunk size and it shows a linear increase. By adding the source code of 75 consecutive kernel versions, the deduplication ratio was about 61.6, by far superior to 5.3 compression ratio obtained by regular compression algorithms such as tar or gzip. As Figure 4–3b shows, the percentage of duplicate chunks does not depend on the chunk size neither. After adding the source codes, the percentage of duplicate chunks was 98 %. This is the case when there are less changes between the source codes in the kernel versions. This shows a clear benefit of using HD2FS for this commonly used data.



(a) Deduplication ratio analysis for Linux Kernels 3.x.

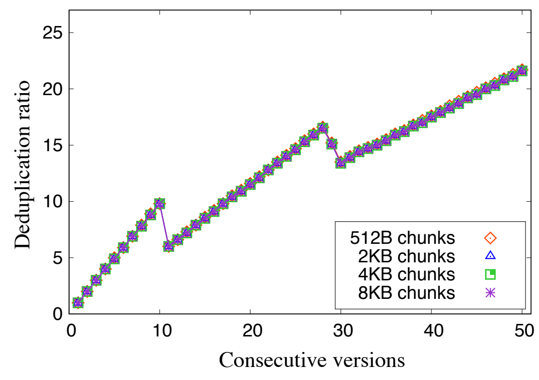


(b) Percentage of duplicate chunks analysis for Linux Kernels 3.x.

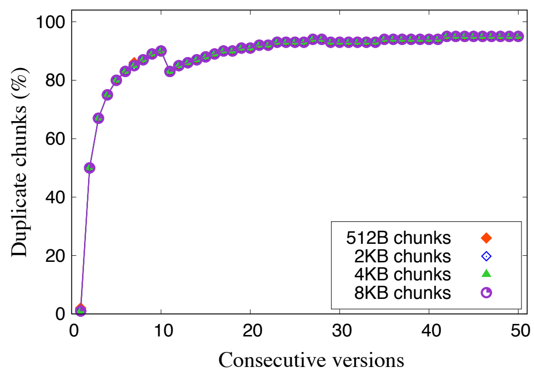
Figure 4–3: Deduplication ratio and percentage of duplicate chunks analysis for Linux Kernels 3.x distributions.

Linux kernel 4.x source codes

The fourth set of experiments used 50 consecutive versions of the Linux kernel 4.x source codes downloaded from their website. The source codes also were also written to HD2FS cluster. The experiment was repeated for different chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB.



(a) Deduplication ratio analysis for Linux Kernels 4.x.



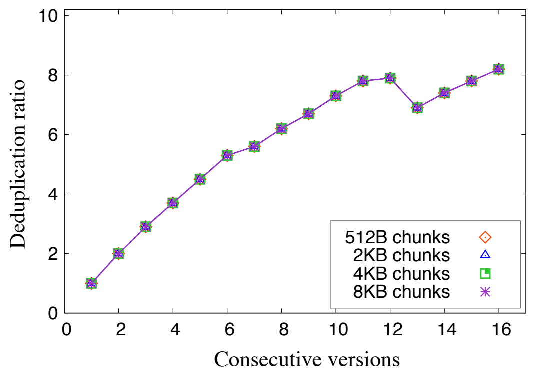
(b) Percentage of duplicate chunks analysis for Linux Kernels 4.x.

Figure 4–4: Deduplication ratio and percentage of duplicate chunks analysis for Linux Kernels 4.x distributions.

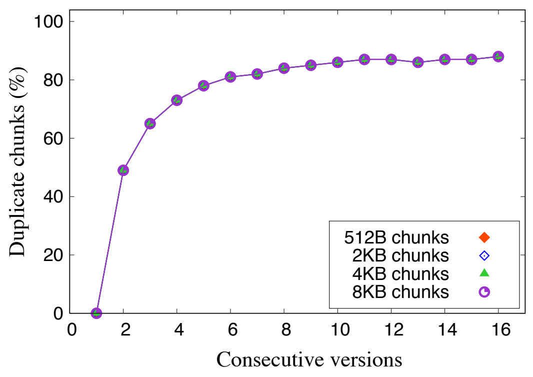
As Figure 4–4a shows, the deduplication ratio does not depend on the chunk size and it shows a linear increase. By adding the source code of 50 consecutive kernel versions, the deduplication ratio was about 21.6, greater than the 5.6 compression ratio obtained by regular compression algorithms such as tar or gzip. As Figure 4–4b shows, the percentage of duplicate chunks does not depend on the chunk size neither. After adding the source codes, the percentage of duplicate chunks was 95 %. We can observe some downs in the figure. It is the case when the current written source code has lots of changes compared to the previous already written source codes.

Database log files

The fifth set of experiments used the previously saved InnoDB database log files. This experiment was also repeated for different chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. As Figure 4–5a shows, the deduplication ratio does not depend on the chunk size and it shows a linear increase. By adding 16 consecutive database log files, the deduplication ratio was about 8.2. As Figure 4–5b shows, the percentage of duplicate chunks does not depend on the chunk size neither. After adding the database log files, the percentage of duplicate chunks was 88 %.



(a) Deduplication ratio analysis for Innodb database log files.



(b) Percentage of duplicate chunks analysis for Innodb database log files.

Figure 4–5: Deduplication ratio and percentage of duplicate chunks analysis for 16 consetutive Innodb database log files using HD2FS cluster.

4.1.4 Performance analysis

In order to test HD2FS performance, a set of user files of size 100 MB each was written and than read from HD2FS in order to determine how the deduplication chunk size influence the overall performance. As Figure 4–6 shows, for very small chunk sizes such as 512 B or 1 KB, the write times are relatively high compared to chunk sizes over 2 KB. The no-dedup experiment was done with the original unmodified HDFS cluster, therefore without deduplication. We can see that HD2FS performance is very similar to the performance of the original HDFS, but with the big advantage of improving storage space requirements.

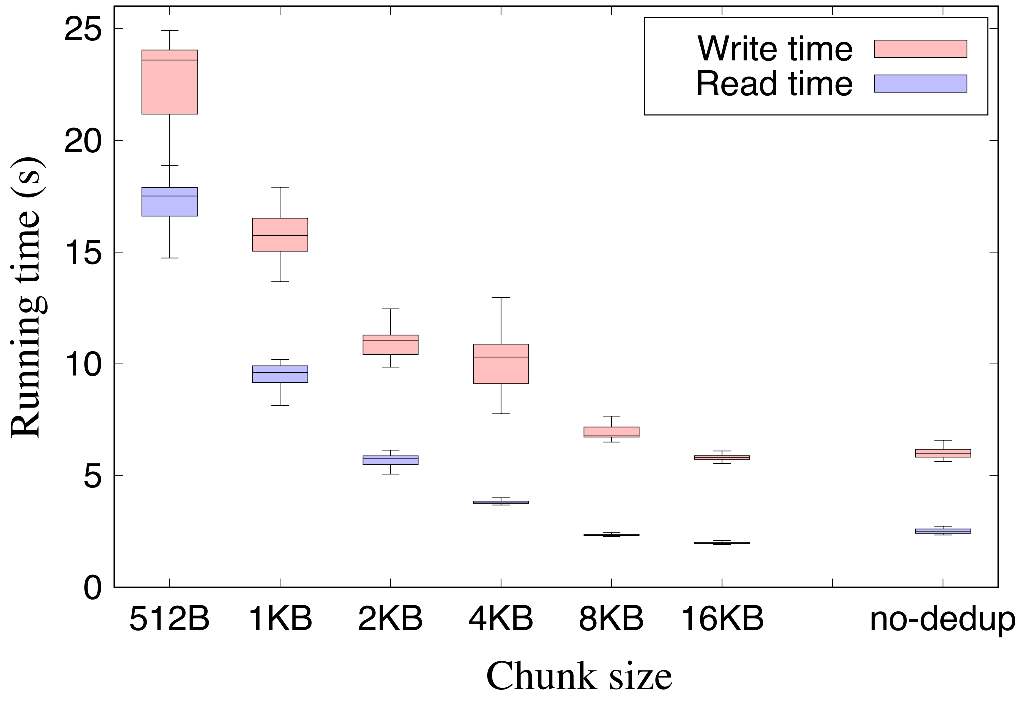


Figure 4–6: Comparative box plots for Write and Read times for 100 MB files using HD2FS cluster for different chunk sizes. Write times include the deduplication time and read times include the reconstruction time.

4.1.5 Scalability analysis

The scalability of HD2FS was also tested. The experiments were started with the number of data nodes equal the default replication (3) in HDFS. Starting the cluster with 3 data nodes, a set of user files of size 100 MB each was written and than read from HD2FS. As Figure 4–7 shows, HD2FS cluster performance is not altered by increasing the number of data nodes. Several configurations were tested. The experiment was than repeated for 4, 5, and 6, the maximum available data nodes in the cluster. The results show that as the number of data nodes is increasing, the overall performance is slightly improved, the improvement is more visible for write times.

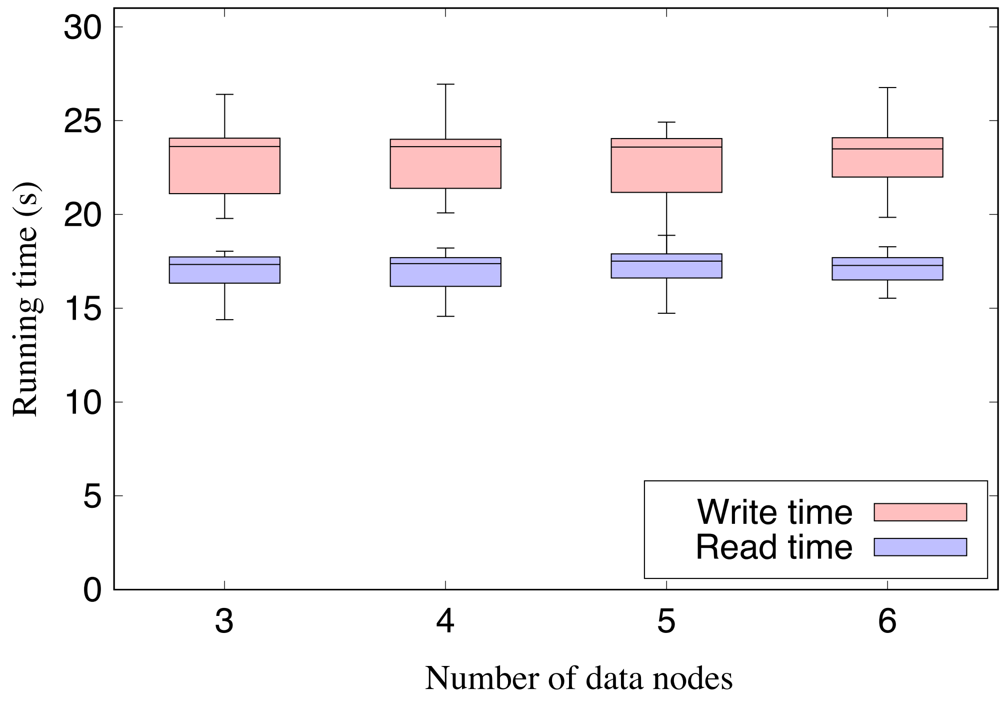


Figure 4–7: Scalability analysis for HD2FS. Comparative box plots for Write and Read times were measured for different HD2FS cluster configurations. The number of data nodes was increased from 3 up to 6, the maximum available data nodes in the cluster.

4.2 Summary

In this chapter, an application of the new deduplicated and distributed file system HD2FS to store user real data in cloud was presented. HD2FS was designed and implemented using Hadoop Distributed File System (HDFS) by adding deduplication. A study of performance was done using real world data sets. The advantages of using HD2FS have been presented. In particular, we have evaluated deduplication ratio, percentage of duplicate chunks, write and read times for the following chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. Our results over different types of commonly used data sets, show that the obtained deduplication ratio and percentage of duplicate chunks values are superior in the case of using HD2FS when compared to HDFS. The scalability analysis results show that as the number of data nodes is increasing, the overall performance is slightly improved, the improvement is more visible for write times. This means that there is potential for HD2FS to be effectively integrated to improve storage management of user data. Also, the obtained high deduplication ratio values and the write and read times comparable with HDFS, gives HD2FS a clear benefit.

Chapter 5
USING HD2FS TO MANAGE LARGE SCALE GENOMIC DATA

During the last years, the cost of sequencing has dropped, and the amount of generated genomic sequence data has skyrocketed. As a consequence, genomic sequence data have become more expensive to store than to generate. The storage needs for genomic sequence data are also following this trend. In order to solve these new storage needs, different compression algorithms have been used. Nevertheless, typical compression ratios for genomic data range between 3 and 10. In this chapter, we present the use of HD2FS, in order to improve overall storage capacity and efficiency in distributed file systems that handle genomic data

5.1 Introduction to Genomics

Genomics is the genetics area concerned with the sequencing and analysis of genomes. A genome is a complete set of deoxyribonucleic acid (DNA) of an organism, including all its genes. A gene is the basic physical and functional unit of heredity and is made up of DNA. DNA is the hereditary material in humans and almost all other organisms, stored as a code made up of four bases: adenine (A), guanine (G), cytosine (C), and thymine (T). DNA bases pair up with each other such that base A pairs with base T, and base C pairs with base G, to form units called base pairs (bp). One of the most important DNA properties is that it can replicate itself. A genome is usually stored in a single file. One of the most common file types currently used to store a genome is the FASTA format. The FASTA format is text-based, and originates from the FASTA software package [46, 47]. FASTA has become a standard in genomics. The simplicity of its format makes it easy to manipulate and parse sequences using languages like Python, Perl, Ruby, and other text-processing tools. A sequence in FASTA format is represented as a series of lines. Usually, the first line in a FASTA file starts with a ">" symbol. Following the initial line, is the actual sequence itself in standard one-letter code similar to $ATTATCATTTCTAAAGAGA$...

Human DNA consists of about 3 billion base pairs (bp), more than 99 percent of those bases are the same in all people, and it's representation requires about 100 GB of data [48]. Because the cost of sequencing a human genome has dropped from \$1,000,000 in 2007 to about \$1,000 in 2017, and is still dropping since then [49], the amount of generated genomics data has skyrocketed and the storage needs for genomics data are on pressure to keep up with the demand. Because a tumor's whole genome and a matching normal tissue needs about 1 TB of uncompressed data [48], one million genomes would require 1 million TB, which is 1 Exabyte.

Recently, the use of deduplication has shown potential to remove storage redundancy in similar files across file systems. The concept of a file can be adapted to refer to chunks (data blocks) and file recipes [3]. The file recipe for a file is a synopsis that contains a list of chunk identifiers (fingerprints) that comprise the file. Each chunk identifier can be created using a collision resistant hash over the contents of the block [5]. Once the chunk identifiers in a file recipe have been obtained, they can be combined as prescribed in the file recipe to reconstruct the file.

Typical deduplication solutions rely on fixed-size chunking methods [18]. Although it provides a simpler implementation, the benefits of deduplication in those scenarios limit the use of this technique mostly to backup data [2, 3, 50].

We present the use of HD2FS for genomics data in order to improve overall storage capacity and efficiency in distributed file systems that handle genomic data. Instead of storing multiple copies of the same genomics data, our deduplication system will store only the data that is different along with a map to reconstruct all genomics data files. In order to demonstrate it's feasibility and performance, we present a study of the relation between the amount of different types of mutations like point mutations, substitutions, inversions, and the deduplication ratio for vertebrate genomes data set stored in FASTA genomics data files.

Gene mutations play a part in both normal and abnormal biological processes including: evolution, cancer, and the development of the immune system, including junctional diversity. Gene mutation are permanent alterations in the DNA sequence, resulting from errors during DNA replication or any types of damage to the DNA sequence. It may also result from insertion or deletion of segments of DNA due to mobile genetic elements [51, 52]. Regarding the size of gene mutations, there are small scale mutations such as Point Mutation (Figure 5–1a), Substitution (Figure 5–1b), Inversion (Figure 5–1c), Insertion (Figure 5–1d), Deletion (Figure 5–2b), and large scale mutations such as Duplication of genes (Figure 5–2a), and Deletions (Figure 5–2b).

5.2 Deduplication for Genomic Data

The use of HD2FS to improve overall storage capacity and efficiency in distributed file systems that handle genomic data will be presented. HD2FS uses fixed-length deduplication, where each file is being divided into equally sized chunks. The deduplication engine (DE) is implemented at the name node level in a HDFS system. It consists of a typical HDFS name node plus a deduplication database which will hold information regarding chunks and the associated files they belong to, and a chunk identifier to generate unique ID per chunk. The DE will perform writes as follows: using the SHA-256 secure hash algorithm, a unique chunk identifier (fingerprint) is generated for every chunk and stored in the file recipe as well as being sent to the deduplication database. If the chunk identifier already exists

...	*T*	*A*	T	*C*	*T*	*G*	*C*	*A*	*G*	*A*	...
...	*T*	*A*	C	*C*	*T*	*G*	*C*	*A*	*G*	*A*	...

(a) **Point mutation**. Base **T** is replaced by the new base **C**.

...	*T*	*A*	T	C	T	G	C	*A*	*G*	*A*	...
...	*T*	*A*	C	A	G	T	T	*A*	*G*	*A*	...

(b) **Substitution**. The sequence **TCTGC** is replaced by **CAGTT**.

...	*T*	*A*	T	C	T	A	A	*A*	*G*	*A*	...
...	*T*	*A*	A	A	T	C	T	*A*	*G*	*A*	...

(c) **Inversion**. The sequence **TCTAA** being replaced by its inverse.

...	*T*	*A*	T	C	T	A	A	A	G	...	...
...	*T*	*A*	A	T	C	T	A	A	A	G	...

(d) **Insertion**. The new base **A** is inserted and the following sequence fter the deleted base is shifted one position to the right.

...	*T*	*A*	T	C	T	A	A	A	G	A	...
...	*T*	*A*	C	T	A	A	A	G	A	...	...

(e) **Deletion**. The base **T** is deleted and the following sequence after the deleted base is shifted one position to the left.

Figure 5–1: Small scale mutations: Point mutation, Substitution, Inversion, Insertion, and Deletion.

in the deduplication database, meaning that the chunk is already stored, the corresponding chunk counter will be incremented by 1. If the chunk identifier is not found in the deduplication database, it will be inserted and the chunk content will be stored in the deduplication database. In the case of reads, a file reconstruction algorithm will search for chunks stored in the file recipe and assemble the content by appropriately merging such chunks. Since a HDFS already creates file chunks, an extension to build the DE can leverage on this functionality and modify it to work at a different chunk scale.

In order to optimize HD2FS for genomics data, we are interested in finding the best chunk size (for read/write performance) for commonly used file types in genomics, such as FASTA genomics data files. We consider the following potential

...	*T*	*A*	*T*	**...**	*A*	**C**	**A**	**A**	**G**	**A**	**...**
...	*T*	*A*	*T*	**...**	*A*	**T**	**...**	**A**	**C**	**A**	**...**

(a) **Duplication of genes**. The gene **T...A** is duplicated, and the entire following sequence is shifted to the right.

...	*T*	*A*	*T*	**C**	**T**	**...**	**A**	**A**	**G**	**A**	**...**
...	*T*	*A*	*T*	**A**	**G**	**A**	**...**	**...**	**...**	**...**	**...**

(b) **Gene deletion**. The entire following sequence is shifted to the left.

Figure 5–2: Large scale mutations: Duplication of genes, and Gene deletion.

chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB. A performance evaluation using such sizes will be performed and the one that produces the best results is selected as the ideal chunk size for these data.

5.3 Experiments

The experiments were designed to provide an understanding of the relation between the amount of mutations, such as point mutations, substitutions, inversions and the deduplication ratio in order to select a chunk size that can be used for storing genomic data in HD2FS. Toward those ends, the focus was on the amount of mutations, and given this amount, the deduplication ratio for different chunk sizes was computed. The results of the experiments should provide us with sufficient information to identify the best chunk size for genomics data files.

5.3.1 Experimental Setup

The experiments were performed on a DELL PowerEdge R220 server equipped with an Intel(R) Xeon(R) CPU E3-1286 v3 @ 3.70 GHz 8 core CPU, 32 GB of RAM, 400 GB SSD, and 8 TB of extra storage using regular hard drives. The experiments were conducted on the SSD as primary storage and the results stored on the extra storage. Taking advantage of its low-latency access, the SSD was also used for storing metadata. Depending on the chunk size used for deduplication, the metadata ranges from 1 % of the total data for 4 KB chunks, to about 10 % of the total data for 512 B chunks. The operating system used was Ubuntu 16.04 LTS with Linux kernel 4.10.

Table 5–1: FASTA files data set.

Modify Type	**Sets**	**Files**	**Size (GB)**
Reference set	1	407	50
Point Mutations	11	4,477	550
Substitutions (700 to 1500 bp)	16	6,512	800
Substitutions (7000 bp average)	11	4,477	550
Inversions (700 to 1500 bp)	16	6,512	800
Inversions (7000 bp average)	11	4,477	550
Total:	**66**	**26,862**	**3,300**

5.3.2 Datasets

The reference set

The data set was generated from a sample of 407 FASTA genomic data files with a total size of 50 GB obtained from the collection of genome files downloaded from ftp.ensembl.org, a genome browser for vertebrate genomes. The compressed file collection has 282 GB, while uncompressed requires 683 GB of storage space. The experiments use the 50 GB sample of 407 FASTA uncompressed files as the reference set. The smallest file has 115 MB, the largest has 140 MB, and the average file size is 125.8 MB.

Inserting modifications to generate duplicate data

The reference set is used to generate new testing sets of modified FASTA files. The modified sets consist of mutations, substitutions and inversions. As showed in Figure 5–1a, we introduced point mutations at random positions. In this way we generated 11 new sets, with the following amounts of point mutations (in thousands): 5, 10, 20, 40, 60, 80, 100, 120, 140, 160, and 200, respectively. Every file in a given set has the same amount of point mutations. We also introduce substitutions (Figure 5–1b) and inversions (Figure 5–1c) at random positions. We considered two cases: substitutions / inversions of sequences between 700 bp and 1,500 bp long, and substitutions / inversions of sequences with an average length of 7,000 bp. In this way we generated 16 new sets with substitutions / inversions of sequences between 700 bp and 1,500 bp having 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, and 40 thousand substitutions per file. We also generated 11 new sets with substitutions / inversions of sequences with an average length of 7,000 bp having 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and 15 thousand substitutions per file. Every file in a given set has the same amount and type of substitutions / inversions. Table 5–1 shows the number of different sets of FASTA files, the total number of files, and the total size. The first line corresponds to the unmodified (reference) set of FASTA files. As shown, our testing dataset is of 3.3 TB of genomic data.

5.3.3 Deduplication for Genomic Data

With the obtained data set, the deduplication ratio and the percentage of duplicate chunks for every modified set of files and the reference set were computed. Let *chunkSize* denote the chunk size, *DedupRatio* denote the deduplication ratio, and *DupCh* denote the percentage of duplicate chunks. The deduplication ratio is defined as the ratio between the original data size and the data size after eliminating the duplicates. A higher *DedupRatio* shows a high redundancy in the file content while a lower ratio shows a high number of unique chunks in the file. Similarly, a high *DupCh* shows that the file content can be represented by a combination of a small subset of unique chunks, while a low value shows highly unique chunk content. We used the following *chunkSize* values: 512 B, 1 KB, 2 KB, and 4 KB. To compute the percentage of duplicate chunks and the deduplication ratio, the chunk identifiers from the original set of files were compared with the chunk identifiers from every modified set of files.

A	*A*	*B*	**C**	*D*	*E*	**F**	*G*	*H*	*I*
A	*A*	*B*	**X**	*D*	*E*	**Y**	*G*	*H*	*I*

Figure 5–3: Each file contains 10 chunks of the same size. Chunks C and F are replaced by chunks X and Y, respectively. Chunks A, B, D, E, G, H, and I are duplicates.

This is an example of these metrics using Figure 5–3, which shows the resulting chunks from two files after deduplication. In this theoretical example, it was assumed that after deduplication, each file was divided into 10 chunks of the same size.

The first one is the reference file, and the second one is the modified file. During deduplication, the chunk identifier together with a counter are recorded into the genomics deduplication database. The total number of chunks, the number of unique chunks (count=1), and the number of distinct chunks were computed by issuing queries to the database. The percentage of duplicate chunks is given by the formula in Equation (3.3). The deduplication ratio is given by the following formula in Equation (3.2). In this example, the total number of chunks is 20. There are 11 distinct chunks (A, B, C, D, E, F, G, H, I, X, Y), and 4 unique chunks (C, F, X, Y). The percentage of duplicate chunks and the deduplication ratio are:

$$DupCh = \frac{20-4}{20} = 80\,\%$$
$$DedupRatio = \frac{20}{11} = 1.82$$

5.3.4 Deduplication vs. Compression

On the other hand, the compression ratio is defined as the ratio between the original uncompressed data size and the compressed data size.

$$Compression\,Ratio = \frac{Original\,Data\,Size}{Compressed\,Data\,Size} \tag{5.1}$$

For genomics data, compression is widely used to save storage space. Different algorithms are used to compress data in an efficient way. The compression algorithm depends on the data we need to compress.

Table 5–2: Chunking and reconstruction times comparison for 100 MB files vs. GZIP compression.

Chunk size	Chunking time (s)	Reconstruction time (s)
		(GZIP decompress) 0.6
4 KB	1.1	0.9
2 KB	2.1	1.8
1 KB	4.1	3.7
512 B	8.2	7.3
	(GZIP compress) 9.1	

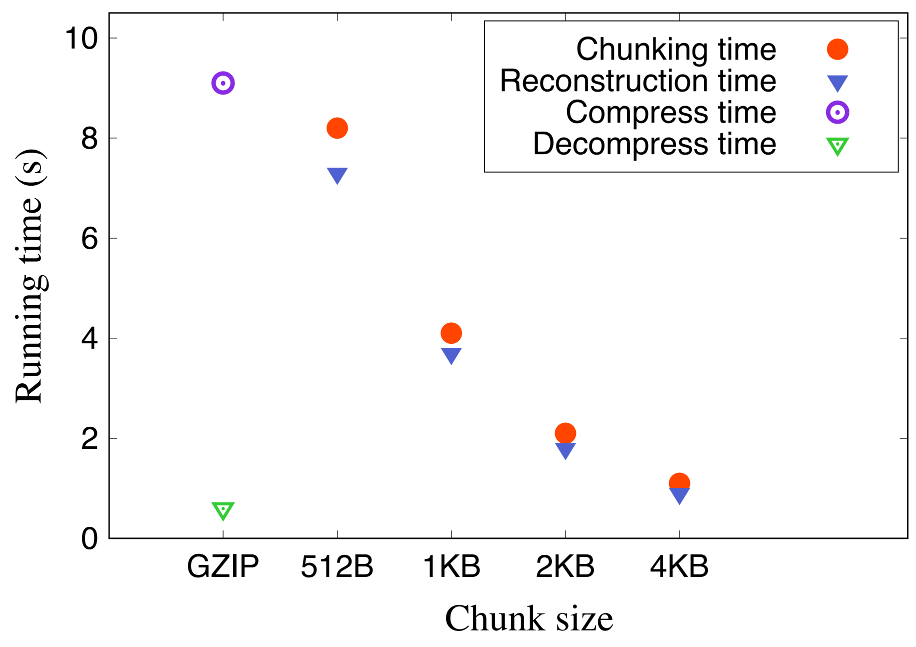


Figure 5–4: Compression vs. Deduplication of a 100 MB size file.

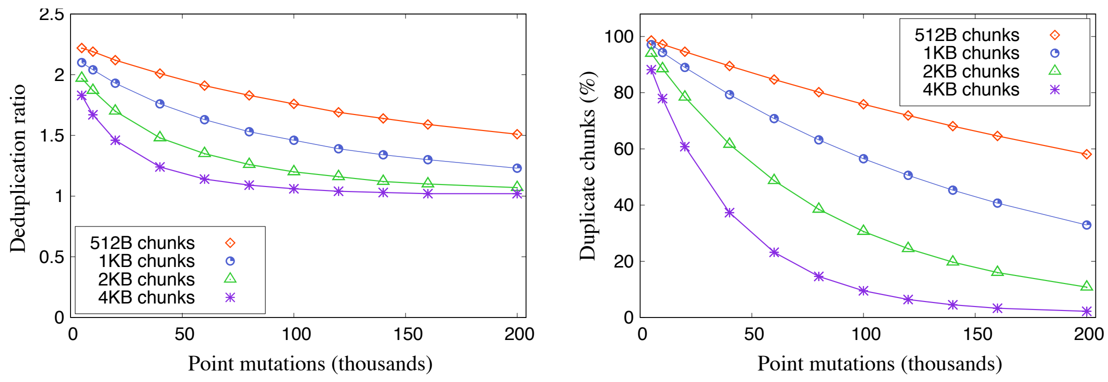
For the data set of FASTA genome files, the compression ratio is between three and four. Compression is identifying redundancy within a single file, only the data within that file is examined. The memory resource requirements are relatively small, only one file has to be processed. Deduplication needs more processing power. The comparisons are done across all the files from the file system, so the potential to identify redundancy is by far superior, especially when the file system contains many copies of modified files. This is the case of FASTA genome files.

5.3.5 Results

Point Mutations

As Figure 5–5a shows, the deduplication ratio ranges between 2.22 and 1.51 (512 B chunks), 2.10 and 1.23 (1 KB chunks), 1.97 and 1.07 (2 KB chunks), 1.83 and 1.02 (4 KB chunks), as the number of point mutations per file increases from 5 thousand to 200 thousand.

As Figure 5–5b shows, for 512 B chunks, the percentage of duplicate chunks is greater than 80 percent, even for 80 thousand point mutations per file. For extreme



(a) Deduplication ratio analysis. (b) Duplicate content analysis.

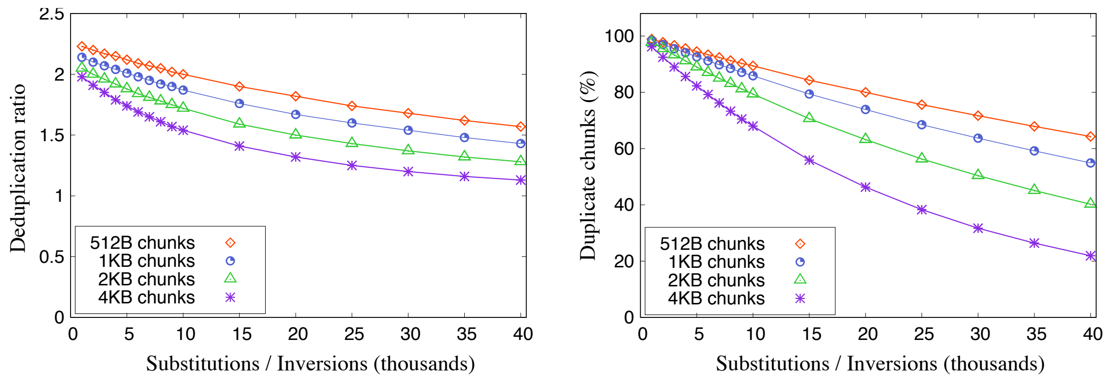
Figure 5–5: Deduplication ratio and duplicate chunks percentage for point mutations in FASTA genome files. The X axis represent the amount of point mutations in files of average size of 125 MB.

cases, where we considered 200 thousand point mutations per file, we still obtained 58 percent of duplicate chunks. For 1 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 40 thousand point mutations per file. For the case of 200 thousand point mutations, we still obtained 33 percent of duplicate chunks. For 2 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 20 thousand point mutations per file. For the case of 200 thousand point mutations, the percentage of duplicate chunks is 10 percent. For 4 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 10 thousand point mutations per file. For the case of 200 thousand point mutations, the percentage of duplicate chunks is only 2 percent.

From Figure 5–4 and Table 5–2, the chunking and reconstruction times when using 512 B chunks are 8.2 seconds, and 7.3 seconds, respectively. If 1 KB chunks are used, the chunking and reconstruction times are 4.1 seconds, and 3.7 seconds, respectively. If compression is used, 9.1 seconds are needed for compressing a 100 MB FASTA genome file, and 0.6 seconds to decompress it. From these experiments, we found that the best chunk size for deduplication of FASTA genome files with many point mutations is 512 B.

Substitutions and Inversions of sequences between 700 and 1,500 base pairs long

As Figure 5–6a shows, the deduplication ratio ranges between 2.23 and 1.57 (512 B chunks), 2.14 and 1.43 (1 KB chunks), 2.05 and 1.28 (2 KB chunks), 1.98 and 1.14 (4 KB chunks), as the number of point mutations per file increases from 5 thousand to 200 thousand.



(a) Deduplication ratio analysis.

(b) Duplicate content analysis.

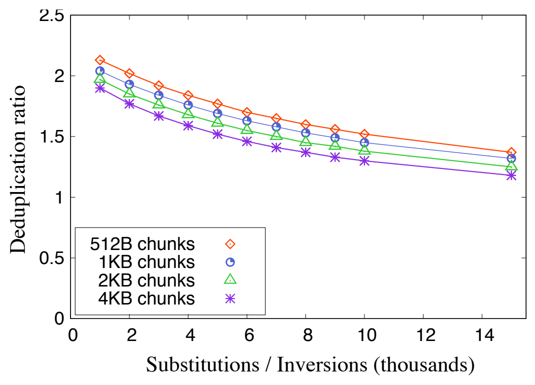
Figure 5–6: Deduplication ratio and duplicate chunks percentage for substitutions and inversions for sequences of length between 700 and 1,500 base pairs in FASTA genome files. The X axis represent the amount of substitutions or inversions in files of average size of 125 MB.

As Figure 5–6b shows, for 512 B chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 20 thousand substitutions per file. For 1 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 15 thousand substitutions per file. For 2 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 10 thousand substitutions per file. For the case of 15 thousand substitutions, the percentage of duplicate chunks is above 65 percent. For 4 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 6 thousand substitutions per file. For the case of 15 thousand substitutions, the percentage of duplicate chunks is still above 50 percent.

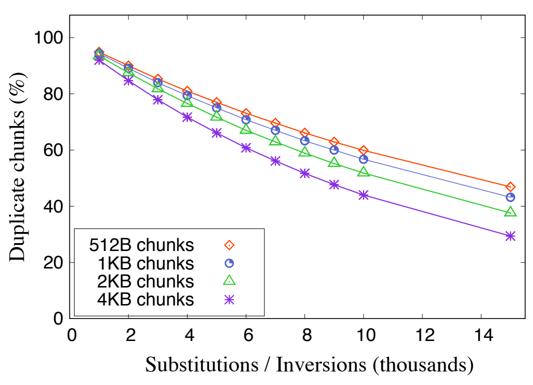
From these experiments, we found that 512 B and 1 KB chunk sizes have very similar performance, also using 2 KB chunks gave very good results. From Figure 5–4 and Table 5–2, the chunking and reconstruction times when using 2 KB chunks are 2.1 seconds, and 1.8 seconds, respectively. If 4 KB chunks are used, the chunking and reconstruction times are 1.1 seconds, and 0.9 seconds, respectively. If compression is used, 9.1 seconds are needed for compressing a 100 MB FASTA genome file, and 0.6 seconds to decompress it. From these experiments, we found that the best chunk size for deduplication of FASTA genome files with many substitutions of length between 700 bp and 1,500 bp is 2 KB.

Substitutions and Inversions of sequences with an average length of 7,000 base pairs

As Figure 5–7a shows, the deduplication ratio ranges between 2.14 and 1.37 (512 B chunks), 2.04 and 1.32 (1 KB chunks), 1.97 and 1.25 (2 KB chunks), 1.90 and 1.01 (4 KB chunks), as the number of point mutations per file increases from 5 thousand to 200 thousand.



(a) Deduplication ratio analysis.



(b) Duplicate content analysis.

Figure 5–7: Deduplication ratio and duplicate chunks percentage for substitutions and inversions for sequences of average length of 7,000 base pairs in FASTA genome files. The X axis represent the amount of substitutions or inversions in files of average size of 125 MB.

As Figure 5–7b shows, for 512 B chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 4 thousand substitutions per file. For extreme cases, where we considered 15 thousand substitutions, we still obtained 47 percent of duplicate chunks. For 1 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 4 thousand substitutions per file. For the case of 15 thousand substitutions, we still obtained 43 percent of duplicate chunks. For 2 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 3 thousand substitutions per file. For the case of 15 thousand substitutions, the percentage of duplicate chunks is above 33 percent. For 4 KB chunks, the percentage of duplicate chunks is greater than 80 percent, even there are 3 thousand substitutions per file. For the case of 15 thousand substitutions, the percentage of duplicate chunks is still above 29 percent.

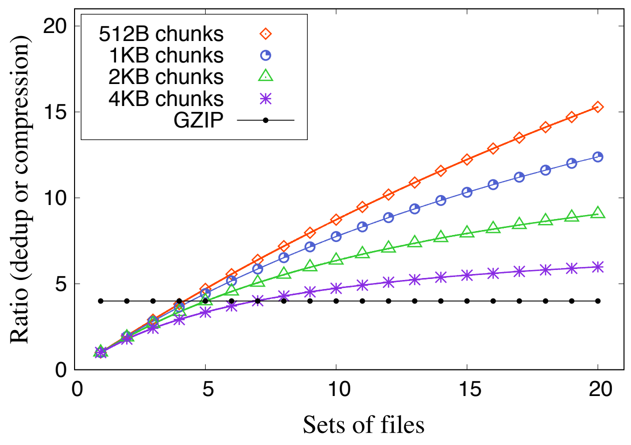
From these experiments, we found that all four chunks sizes behave very similar, so 2 KB, or even 4 KB chunks can be used for deduplication. From Figure 5–4 and Table 5–2, the chunking and reconstruction times when using 2 KB chunks are 2.1 seconds, and 1.8 seconds, respectively. If 4 KB chunks are used, the chunking and reconstruction times are 1.1 seconds, and 0.9 seconds, respectively. If compression is used, 9.1 seconds are needed for compressing a 100 MB FASTA genome file, and 0.6 seconds to decompress it. From these experiments, we found that the best chunk size for deduplication of FASTA genome files with many substitutions of average length of 7,000 bp is 2 KB.

Cummulative deduplication ratio

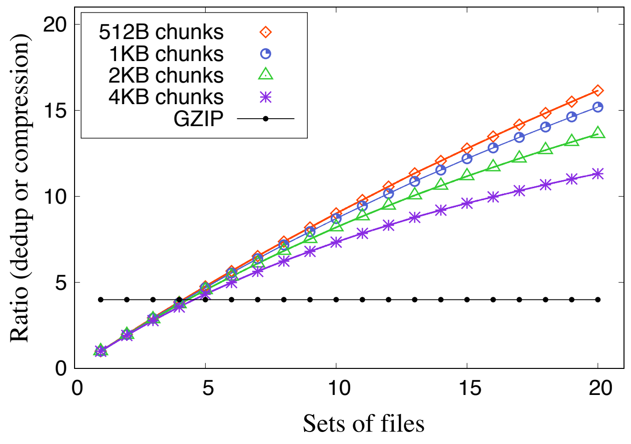
Instead of just deduplicating two sets of files, one reference set and one modified set, the experiment was run on 20 sets of files. After adding a new set of files, the deduplication ratio was computed for the following chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB. As Figure 5–8a shows, the deduplication ratio for point mutations is increasing from 1 to 15.3 for 512 B chunks, from 1 to 12.4 for 1 KB chunks, from 1 to 9.1 for 2 KB chunks, and from 1 to 6.0 for 4 KB chunks. As Figure 5–8b shows, the deduplication ratio for substitutions and inversions of sequences between 700 bp and 1,500 bp long is increasing from 1 to 16.1 for 512 B chunks, from 1 to 15.2 for 1 KB chunks, from 1 to 14 for 2 KB chunks, and from 1 to 11.3 for 4 KB chunks. As Figure 5–8c shows, the deduplication ratio for substitutions and inversions of sequences with an average length of 7,000 bp is increasing from 1 to 9.5 for 512 B chunks, from 1 to 9.2 for 1 KB chunks, from 1 to 8.6 for 2 KB chunks, and from 1 to 7.7 for 4 KB chunks.

For the data set of FASTA genome files, the compression ratio is between 3 and 4, represented by the horizontal line in Figure 5–8a, Figure 5–8b, and Figure 5–8c. As the figures show, the deduplication gives superior results then compression after adding more than 5 sets of files.

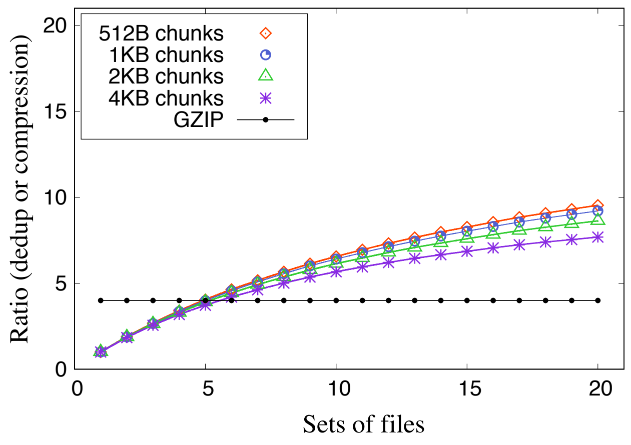
From these experiments we found that by adding more and more data, the deduplication ratio values are consistently greater than the compression ratio values. Therefore, deduplication clearly outperforms compression at any used chunk size, the results are even better for 512 B chunk size.



(a) Point Mutations.



(b) Substitutions and Inversions of sequences between 700 and 1,500 base pairs long.



(c) Substitutions and Inversions of sequences with an average length of 7,000 base pairs.

Figure 5–8: Cumulative Deduplication ratio.

5.4 Summary

A study on the relation between the amount of different types of mutations in genomic data such as point mutations, substitutions, inversions, and the effect of such in the deduplication ratio for a data set of vertebrate genomes in FASTA format to demonstrate the advantage of using HD2FS has been presented. In particular, the deduplication ratio and the percentage of duplicate chunks for the following potential chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB were evaluated. The obtained results over different types of mutations such as point mutations, substitutions, inversions, and the deduplication ratio for a vertebrate genome data set stored in FASTA format files show that the obtained ratio values are superior in the case of using deduplication when compared to the current approach relying on compression. Moreover, 512 B is the best chunk size for commonly used file types in genomics. This means that there is potential for HD2FS to be effectively integrated to improve storage management of genomic data.

Chapter 6
CONCLUSIONS

A new deduplicated and distributed file system HD2FS was designed and implemented by integrating deduplication into Hadoop Distributed File System (HDFS). A preliminary study and a Database Management System comparison in order to design and implement our new deduplicated and distributed file system HD2FS was presented. Deduplication was integrated to Hadoop Distributed File System (HDFS) and MariaDB Galera cluster was used to store and replicate the data.

A study of HD2FS performance was done using real world data sets. The advantages of using HD2FS have been presented. In particular, we have evaluated deduplication ratio, percentage of duplicate chunks, write and read times for the following potential chunk sizes: 512 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. Our results over different types of commonly used data sets, show that the obtained deduplication ratio and percentage of duplicate chunks values are superior in the case of using HD2FS when compared to HDFS without deduplication. This means that there is potential for HD2FS to be effectively integrated to improve storage management of user data. Also, the obtained high deduplication ratio values and the write and read times comparable with HDFS, gives HD2FS a clear benefit.

A study on the relation between the amount of different types of mutations in genomic data such as point mutations, substitutions, inversions, and the effect of such in the deduplication ratio for a data set of vertebrate genomes in FASTA format to demonstrate the advantage of using HD2FS has been presented. In particular, the deduplication ratio and the percentage of duplicate chunks for the following potential chunk sizes: 512 B, 1 KB, 2 KB, and 4 KB were evaluated. The obtained results over different types of mutations such as point mutations, substitutions, inversions, and the deduplication ratio for a vertebrate genome data set stored in FASTA format files show that the obtained ratio values are superior in the case of using deduplication when compared to the current approach relying on compression. Moreover, 512 B is the best chunk size for commonly used file types in genomics. This means that there is potential for HD2FS to be effectively integrated to improve storage management of genomic big data.

Overall, the obtained high deduplication ratio values and the write and read times comparable with HDFS, gives HD2FS a clear benefit.

Other contributions of this research include:

- The design and implementation of a distributed datastore using deduplication, which we called *Smartstorage: a deduplicated and distributed datastore* [8]. This was our first implementation of a deduplicated and distributed datastore.
- A characterization of file attributes that help determine the appropriate chunk size in primary deduplication systems, *The use of file attributes to determine the best chunk size in primary deduplication* [9]. This was done as a preliminary work for this thesis.
- The development of a file-aware deduplication storage system. *Using file-aware deduplication to improve capacity in storage systems* [10].
- The design and implementation of a deduplication storage system for genomics data, in order to improve data storage capacity and efficiency in distributed file systems without compromising I/O performance. *GDedup: Distributed file system level deduplication for genomic big data* [11],
- Integrating deduplication to a distributed file system environment such as the Hadoop Distributed File System (HDFS). *Distributed file system level deduplication* [12].

6.1 Future Works

Future research efforts include an analysis on the scalability limits of our new deduplicated and distributed file system HD2FS for big data storage including genomics.

APPENDICES

Appendix A
DATABASE MANAGEMENT SYSTEM SELECTION

A.1 MariaDB

A.1.1 Create tables

```
create table files(id CHAR(32) NOT NULL,name CHAR(200) NOT NULL,
size INT NOT NULL,chunksize INT NOT NULL, PRIMARY KEY (id,name));
create table chunks(id CHAR(64) PRIMARY KEY NOT NULL,count INT,
content BLOB);
```

A.1.2 Insert file information and chunks into the database

```
INSERT IGNORE INTO files(id,name,size,chunksize) VALUES
(fileID,fileName,fileSize, chunkSize);
INSERT IGNORE INTO chunks(id,count,content)
VALUES (fingerprint,1,chunk) ON DUPLICATE KEY UPDATE count=count+1;
```

A.1.3 Reconstruct the files

```
SELECT content FROM chunks WHERE id=fingerprint;
```

A.2 MongoDB

A.2.1 Create collections

```
db.createCollection("files")

db.files.createIndex({id:1},{unique:true});

db.createCollection("chunks")

db.chunks.createIndex({id:1},{unique:true});
```

A.2.2 Insert file information and chunks into the database

```
db.files.insert({id:fileID,name:fileName,size:fileSize,

chunksize:chunkSize})

db.chunks.update({id:fingerprint},{$inc:{count:1},

$setOnInsert:{content:chunk}}, {upsert:true});
```

A.2.3 Reconstruct the files

```
db.chunks.find({"$where": "this.id==this.fingerprint"},

{"content": 1});
```

A.3 Postgres

A.3.1 Create tables

```
CREATE TABLE files(id CHAR(32) NOT NULL,name CHAR(200) NOT NULL,
size INT NOT NULL,chunksize INT NOT NULL,PRIMARY KEY (id,name));
CREATE TABLE chunks(id CHAR(64) PRIMARY KEY NOT NULL,count INT,
content BYTEA);
```

A.3.2 Insert file information and chunks into the database

```
INSERT INTO files(id,name,size,chunksize) VALUES
(fileID,fileName,fileSize, chunkSize);
INSERT INTO chunks(id,count,content) VALUES (fingerprint,1,chunk)
ON CONFLICT (id) DO UPDATE SET count=chunks.count+1;
```

A.3.3 Reconstruct the files

```
SELECT content FROM chunks WHERE id=fingerprint;
```

A.4 SQLite

A.4.1 Create tables

```
CREATE TABLE files(id CHAR(32) NOT NULL, name CHAR(200) NOT NULL,
size INT NOT NULL, chunksize INT NOT NULL, PRIMARY KEY (id,name));
CREATE TABLE chunks(id CHAR(64) PRIMARY KEY NOT NULL,count INT,
content BLOB);
```

A.4.2 Insert file information and chunks into the database

```
INSERT OR IGNORE INTO files(id,name,size,chunksize) VALUES
(fileID,fileName,fileSize, chunkSize);
UPDATE OR IGNORE chunks SET count=count+1 WHERE id=fingerprint
INSERT OR IGNORE INTO chunks(id,count,content) VALUES
(fingerprint,1,chunk);
```

A.4.3 Reconstruct the files

```
SELECT content FROM chunks WHERE id=fingerprint;
```

Appendix B
JAVA SOURCE CODES FOR CHUNKING AND RECONSTRUCTION USED IN THE PRELIMINARY STUDY

B.1 connectionMariaDB.java

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class connectionMariaDB {

    private static final String JDBC_DRIVER =

     "org.mariadb.jdbc.Driver";

    private static final String DB_URL =

    "jdbc:mariadb://localhost:3306/dedup";

    private static final String DB_USER = "hadoop";

    private static final String DB_PASSWORD = "**********";

    public Connection getDBConnection() throws Exception {

        try {

            Class.forName(JDBC_DRIVER);

        } catch (ClassNotFoundException exception) {

            System.out.println(exception.getMessage());

        }

        try {

            Connection connectMariaDB =
            DriverManager.getConnection(DB_URL, DB_USER,
            DB_PASSWORD);
            return connectMariaDB;
        } catch (SQLException exception) {
            System.out.println(exception.getMessage());
        }
        return null;
    }
    public void closeDBConnection(Connection connectMariaDB)
     throws Exception {
        connectMariaDB.close();
    }
}
```

B.2 dedupChunk.java

```
import org.apache.commons.codec.digest.DigestUtils;
public class dedupChunk {
    private String chunkID;
    private byte[] chunkContent;
    private int chunkSize ;
    //Getters
    public int getChunkSize(){
        return chunkSize;
    }
    public byte[] getChunkContent(){
        return chunkContent;
    }
    public String getChunkID(){
        return chunkID;
    }
    //Setters
    public void setChunkID(){
        this.chunkID = DigestUtils.sha256Hex(chunkContent);
    }
    //Constructors
    public dedupChunk(int chunkSize) {
        this.chunkSize = chunkSize;
        this.chunkContent = new byte[chunkSize];
    }
}
```

B.3 hadoopChunk.java

```
import org.apache.commons.codec.digest.DigestUtils;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.sql.*;
public class hadoopChunk extends File{
    protected String inputFileName;
    dedupChunk dedupChunk = new dedupChunk(512);
    connectionMariaDB connectionMariaDB =
    new connectionMariaDB();
    //Getters
    public String getInputFileName(){
        return inputFileName;
    }
    public long getFileLength(){
        return length();
    }
    public String getFileParent(){
        return getParent();
    }
    public int getLastChunkSize(){
        int lastChunkSize = (int) (getFileLength() %
        dedupChunk.getChunkSize());
        return lastChunkSize;
    }
    public long getNumberOfChunks(){
        long numberOfChunks = (int) (getFileLength() /
        dedupChunk.getChunkSize());
        if (getLastChunkSize()>0){
            numberOfChunks += 1;
        }
        return numberOfChunks;
    }
    //Constructors
    public hadoopChunk(String inputFileName) {
        super(inputFileName);
        this.inputFileName = inputFileName;
    }
    public boolean checkIfExistsInDB() throws Exception {
        String sql_check_if_exists_file =
        "SELECT EXISTS (SELECT fileId FROM file WHERE "
                + "fileName = '"
                + inputFileName
                + "')";
        Connection connectMariaDB =
        connectionMariaDB.getDBConnection();
        Statement sqlStatement = connectMariaDB.createStatement();
        try{
            ResultSet fileExists =
            sqlStatement.executeQuery(sql_check_if_exists_file);
            fileExists.next();
            boolean exists = fileExists.getBoolean(1);
            return exists;
        } finally {
            connectionMariaDB.closeDBConnection(connectMariaDB);
            sqlStatement.close();
        }
    }
    public String generateFileID() throws Exception {
        String fileID =
        DigestUtils.md5Hex(Files.readAllBytes(
        Paths.get(inputFileName)));
        return fileID;
    }
    public long computeFileLength() throws Exception {
            Connection connectMariaDB =
            connectionMariaDB.getDBConnection();
            Statement sqlStatement =
            connectMariaDB.createStatement();
            InputStream in =
            new FileInputStream(new File(inputFileName + ".fr"));
            try {
                long fileLength = 0;
                int numBytes;
                byte buf[] = new byte[64];
                int bytesRead = in.read(buf);
                String chunkId = new String(buf);
                while (bytesRead>0) {
                    String sql_read_chunk_numBytes =
                    "SELECT numBytes from chunk where chunkId =
                    '"  + chunkId + "' LIMIT 1;";
                    ResultSet storedChunkSize =
                    sqlStatement.executeQuery(
                    sql_read_chunk_numBytes);
                    storedChunkSize.next();
                    numBytes =
                    storedChunkSize.getInt("numBytes");
                    fileLength += numBytes;
                    bytesRead = in.read(buf);
                    chunkId = new String(buf);
                }
                in.close();
                return fileLength;
            }
             finally{
                    connectionMariaDB.closeDBConnection(
                    connectMariaDB);
                    sqlStatement.close();
                    }
    }
    public void insertIntoDB() throws Exception {
        String sql_insert_file =
        "INSERT IGNORE INTO file(fileId, fileName, fileSize)
        VALUES ('" + generateFileID() + "', '"
        + inputFileName + "', " + getFileLength() + ");";
        Connection connectMariaDB =
        connectionMariaDB.getDBConnection();
        Statement sqlStatement =
         connectMariaDB.createStatement();
        try{
            sqlStatement.executeQuery(sql_insert_file);
        } finally {
            connectionMariaDB.closeDBConnection(
            connectMariaDB);
            sqlStatement.close();
        }
    }
    public void dedupHadoopChunk() throws Exception {
        Connection connectMariaDB = null;
        PreparedStatement sql_insert_blob = null;
        try {
            connectMariaDB = connectionMariaDB.
            getDBConnection();
            connectMariaDB.setAutoCommit(false);
            String sql_insert_chunk_content =
            "INSERT IGNORE INTO chunk(chunkId, numBytes,
            count, content) VALUES( ?, ?, 1, ?)"
             + " ON DUPLICATE KEY UPDATE count=count+1;";
            if (!checkIfExistsInDB()) {
                System.out.print(
                "The file does not exists in the database");
                File fileDirectoryDedup =
                new File(getFileParent());
                if (!fileDirectoryDedup.exists()) {
                    fileDirectoryDedup.mkdirs();
                }
                BufferedWriter fileRecipe =
                new BufferedWriter(new FileWriter(
                inputFileName + ".fr" ));
                InputStream in =
                new FileInputStream(inputFileName);
                sql_insert_blob =
                connectMariaDB.prepareStatement(
                sql_insert_chunk_content);
                int chunkSize = dedupChunk.getChunkSize();
                byte chunk[] = new byte[chunkSize];
                int bytesToChunk = in.read(chunk);
                while (bytesToChunk>0) {
                    int bytesToChunkNext =
                    (chunkSize<bytesToChunk) ? chunkSize :
                     (int) bytesToChunk;
                    String chunkIdString =
                    DigestUtils.sha256Hex(chunk);
                    // sha256 for the current chunk content
                    sql_insert_blob =
                    connectMariaDB.prepareStatement(
                    sql_insert_chunk_content);
                    sql_insert_blob.setString(1,chunkIdString);
                    sql_insert_blob.setInt(2,bytesToChunkNext);
                    sql_insert_blob.setBinaryStream(3,
                    new ByteArrayInputStream(chunk),
                    bytesToChunkNext);
                    sql_insert_blob.executeUpdate();
                    fileRecipe.write(chunkIdString);
                    bytesToChunk = in.read(chunk);
                }
                connectMariaDB.commit();
                fileRecipe.close();
                insertIntoDB();
                System.out.println("Successfully added!");
            } else {
                connectMariaDB.rollback();
                System.out.println(
                "File already in the database!");
            }
        } catch (SQLException sqlException1) {
            try {
                if(connectMariaDB != null)
                    connectMariaDB.rollback();
            } catch (SQLException sqlException2) {
              System.out.println(sqlException2.getMessage());
            }
            System.out.println(sqlException1.getMessage());
        } finally {
            try {
                if (sql_insert_blob != null) {
                    sql_insert_blob.close();
                }
                if (connectMariaDB != null) {
                    connectionMariaDB.closeDBConnection(
                    connectMariaDB);
                }
            } catch (SQLException sqlException3) {
                System.out.println(sqlException3.getMessage());
            }
        }
    }
    public hadoopChunk reconstructHadoopChunk()
    throws Exception {
        hadoopChunk reconstructedHadoopChunk =
        new hadoopChunk(inputFileName);
        String sql_file_dedup_properties =
        "SELECT fileId FROM file WHERE fileName='"
                + inputFileName + "';";
        Connection connectMariaDB =
        connectionMariaDB.getDBConnection();
        Statement sqlStatement =
        connectMariaDB.createStatement();
        try{
            if(! checkIfExistsInDB()) {
                System.out.println(
                "The requested file does not exists.");
            } else {
                System.out.print("Reconstructing ... ");
                File fileDirectoryReconstruct =
                new File(getFileParent());
                if (!fileDirectoryReconstruct.exists()) {
                    fileDirectoryReconstruct.mkdirs();
                }
                FileOutputStream out =
                new FileOutputStream(reconstructedHadoopChunk);
                ResultSet chunkProperties =
                sqlStatement.executeQuery(
                sql_file_dedup_properties);
                chunkProperties.next();
                InputStream in =
                new FileInputStream(
                new File(inputFileName + ".fr"));
                String originalFileID =
                chunkProperties.getNString("fileId");
                byte buf[] = new byte[64];
```

```
int bytesRead = in.read(buf);
String chunkId = new String(buf);
while (bytesRead>0){
    String sql_read_chunk_content =
    "SELECT content, numBytes from chunk where
    chunkId = '"
            + chunkId
            + "' LIMIT 1;";
    PreparedStatement statement =
    connectMariaDB.prepareStatement(
    sql_read_chunk_content);
    ResultSet chunkContentAndSize =
    statement.executeQuery();
    chunkContentAndSize.next();
    InputStream inp =
    chunkContentAndSize.getBinaryStream(
    "content");
    int chunkSize =
    chunkContentAndSize.getInt("numBytes");
    byte[] chunkByte = new byte[chunkSize];
    while (inp.read(chunkByte)>=0) {
        out.write(chunkByte);
    }
    bytesRead = in.read(buf);
    chunkId = new String(buf);
}
out.close();
in.close();
if ( originalFileID.compareTo(
                reconstructedHadoopChunk.generateFileID())==0)
                {
                    System.out.println(
                    "File reconstructed successfully!");
                } else {
                    System.out.println(
                    "Error reconstructing the file!");
                }
            }
        }
        catch (Exception exception) {
            System.out.println(exception.getMessage());
        } finally {
            try {
                if (sqlStatement != null) {
                    sqlStatement.close();
                }
                if (connectMariaDB != null) {
                    connectionMariaDB.closeDBConnection(
                    connectMariaDB);
                }
            } catch (SQLException sqlException) {
              System.out.println(sqlException.getMessage());
            }
        }
    return reconstructedHadoopChunk;
    }
}
```

Appendix C
CHUNK CONTENTS STORED IN THE DATABASE

C.1 Deduplication method in HD2FS

```
public void dedupFromByteArray(Connection connectMariaDB,
OutputStream out, byte b[], long dedupChunkSize, int numBytes)
throws Exception {
          String sql_insert_chunk_content =
          "INSERT IGNORE INTO chunk(chunkId, numBytes, count,
           content) VALUES( ?, ?, 1, ?)"
           + " ON DUPLICATE KEY UPDATE count=count+1;";
          int chunkSize = (int)dedupChunkSize;
          int bytesToChunk = numBytes;
          while (bytesToChunk>0) {
              int bytesToChunkNext = (chunkSize < bytesToChunk)
              ? chunkSize : bytesToChunk;
              byte [] chunk = Arrays.copyOfRange(b,numBytes -
              bytesToChunk,numBytes - bytesToChunk
              + bytesToChunkNext);
              String chunkId = DigestUtils.sha256Hex(chunk);
              // compute sha256 for the current chunk content
              sql_insert_blob =
              connectMariaDB.prepareStatement(
              sql_insert_chunk_content);
```

```
                sql_insert_blob.setString(1, chunkId);
                sql_insert_blob.setInt(2, bytesToChunkNext);
                sql_insert_blob.setBinaryStream(3,
                new ByteArrayInputStream(chunk),
                bytesToChunkNext);
                out.write(chunkId.getBytes(), 0,
                chunkId.getBytes().length);
                try {
                    sql_insert_blob.executeUpdate();
                } finally {
                    sql_insert_blob.close();
                }
                bytesToChunk -= bytesToChunkNext;
            }
}
```

C.2 Reconstruction method in HD2FS

```
public void reconstructFromStreamToFile(Connection
        connectMariaDB, InputStream in, PathData target)
        throws Exception {
           Statement sqlStatement =
           connectMariaDB.createStatement();
           PathData newFile = target;
           FileOutputStream out =
           new FileOutputStream(newFile.toFile());
           try{
               byte buf[] = new byte[64];
               int bytesRead = in.read(buf);
               String chunkId = new String(buf);
               while (bytesRead > 0){
                   String sql_read_chunk_content =
                   "SELECT content, numBytes from chunk where
                   chunkId = '" + chunkId + "' LIMIT 1;";
                   PreparedStatement statement =
                   connectMariaDB.prepareStatement
                   (sql_read_chunk_content);
                   ResultSet chunkContentAndSize =
                   statement.executeQuery();
                   chunkContentAndSize.next();
                   InputStream inContent =
                   chunkContentAndSize.getBinaryStream("content");
                   int chunkSize =
                   chunkContentAndSize.getInt("numBytes");
                   byte[] chunkByte = new byte[chunkSize];
                   while (inContent.read(chunkByte) >= 0) {
```

```
                out.write(chunkByte);
            }
            bytesRead = in.read(buf);
            chunkId = new String(buf);
        }
        out.close();
        in.close();
    }
    catch (Exception exception) {
    System.out.println(exception.getMessage());
    } finally {
        try {
            if (sqlStatement != null) {
                sqlStatement.close();
            }
        } catch (SQLException sqlException) {
          System.out.println(sqlException.getMessage());
        }
    }
}
```

Appendix D
CHUNK CONTENTS STORED AS FILES

D.1 Deduplication method in HD2FS

```
public void dedupFromByteArray(OutputStream out, byte b[],
          long dedupChunkSize, int numBytes) throws Exception{
          int chunkSize = (int)dedupChunkSize;
          int bytesToChunk = numBytes;
          while (bytesToChunk>0) {
              int bytesToChunkNext =
              (chunkSize<bytesToChunk) ? chunkSize : bytesToChunk;
              byte [] chunk = Arrays.copyOfRange(b,numBytes -
              bytesToChunk,numBytes - bytesToChunk
              + bytesToChunkNext);
              String chunkId = DigestUtils.sha256Hex(chunk);
              // compute sha256 for the current chunk content
              FileOutputStream outputStream =
              new FileOutputStream(dedupChunkFolder + chunkId);
              outputStream.write(chunk);
              outputStream.close();
              out.write(chunkId.getBytes(), 0,
              chunkId.getBytes().length);
              bytesToChunk -= bytesToChunkNext;
          }
}
```

D.2 Reconstruction method in HD2FS

```
public void reconstructFromStreamToFile(String dedupChunkFolder,
      long dedupChunkSize, InputStream in, PathData target)
      throws Exception {
      PathData newFile = target;
      FileOutputStream out =
      new FileOutputStream(newFile.toFile());
      try{
        byte buf[] = new byte[64];
        int bytesRead = in.read(buf);
        String chunkId = new String(buf);
        while (bytesRead > 0){
          InputStream chunkStream =
          new FileInputStream(dedupChunkFolder + chunkId);
          byte[] chunkByte = new byte[chunkStream.available()];
          while (chunkStream.read(chunkByte) >= 0) {
            out.write(chunkByte);
          }
          bytesRead = in.read(buf);
          chunkId = new String(buf);
        }
        out.close();
        in.close();
      }
      catch (Exception exception) {
        System.out.println(exception.getMessage());
      }
}
```

Appendix E
HDFS-SITE.XML

E.1 hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
 <!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at
    http://www.apache.org/licenses/LICENSE-2.0
  Unless required by applicable law or agreed to in writing,
  software distributed under the License is distributed on an
  AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
  OF ANY KIND, either express or implied.
  See the License for the specific language governing
  permissions and limitations under the License.
  See accompanying LICENSE file.
-->
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
 <name>dfs.replication</name>
  <value>3
  </value>
```

```
</property>
<property>
  <name>dfs.namenode.name.dir</name>
    <value>/home/hadoop/hadoopdata/hdfs/namenode
    </value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
    <value>/home/hadoop/hadoopdata/hdfs/datanode
    </value>
</property>
<property>
  <name>dfs.dedupchunk.dir</name>
    <value>/home/hadoop/hadoopdata/dedupchunk/
    </value>
</property>
<property>
    <name>dfs.dedupchunk.location.db</name>
    <value>false
    </value>
    <description>Dedup chunk location. true for database
                          and false for file</description>
</property>
<property>
     <name>dfs.namenode.http-address</name>
     <value>hadoopb-name1.ece.uprm.edu:50070
     </value>
     <description>NameNode hostname for http access.
     </description>
```

```
</property>
<property>
   <name>dfs.namenode.secondary.http-address</name>
   <value>hadoopb-name2.ece.uprm.edu:50090
   </value>
   <description>Secondary NameNode Hostname for http access.
   </description>
</property>
<property>
    <name>dfs.namenode.checkpoint.dir</name>
 <value>/home/hadoop/hadoopdata/hdfs/namesecondary
 </value>
</property>
<property>
    <name>dfs.blocksize</name>
    <value>1048576
    </value>
    <description>Block size</description>
</property>
<property>
    <name>dfs.stream-buffer-size</name>
    <value>1048576
    </value>
    <description>Stream buffer size</description>
</property>
<property>
    <name>io.file.buffer.size</name>
    <value>1048576
    </value>
    <description>Buffer size</description>
</property>
<property>
    <name>dfs.client-write-packet-size</name>
         <value>1048576
         </value>
    <description>Write packet size</description>
</property>
<property>
    <name>dfs.dedupchunksize</name>
    <value>512
    </value>
    <description>Dedup chunk size</description>
</property>
</configuration>
```

REFERENCES

[1] Dirk Meister, André Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 175–182, Berkeley, CA, USA, 2013. USENIX Association.

[2] Nohhyun Park and David J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, February 2012.

[4] 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*, SYSTOR '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[5] J. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[6] Xing Lin, Fred Douglis, Jim Li, Xudong Li, Robert Ricci, Stephen Smaldone, and Grant Wallace. Metadata considered harmful ... to deduplication. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, pages 11–11, Berkeley, CA, USA, 2015. USENIX Association.

[7] Maohua Lu, David Chambliss, Joseph Glider, and Cornel Constantinescu. Insights for data reduction in primary storage: A practical analysis. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 17:1–17:7, New York, NY, USA, 2012. ACM.

[8] P. Bartus et al. Smartstorage: A deduplicated and distributed datastore. *Poster presented at the CAHSI Summit Conference* 2015*, Caribe Hilton, San Juan, PR, September* 11*,* 2015.

[9] P. Bartus and E. Arzuaga. On the use of file attributes to determine the best chunk size in primary deduplication. *Poster presented at the CAHSI Summit /HEENAC Conference* 2016*, Anaheim Convention Center, CA, October* 7*,* 2016.

[10] P. Bartus and E. Arzuaga. Using file-aware deduplication to improve capacity in storage systems. In *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, Aug 2017.

[11] P. Bartus and E. Arzuaga. Gdedup: Distributed file system level deduplication for genomic big data. *Accepted to the IEEE International Congress on Big Data, July 2-7, 2018, San Francisco, CA, USA.*

[12] P. Bartus and E. Arzuaga. Distributed file system level deduplication. *Submitted to the ACM Symposium on Cloud Computing (SoCC'18), May 17, 2018.*

[13] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[14] Joao Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Comput. Surv.*, 47(1):11:1–11:30, June 2014.

[15] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 111–123, Berkeley, CA, USA, 2009. USENIX Association.

[16] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[17] Chan-I Ku, Guo-Heng Luo, Che-Pin Chang, and Shyan-Ming Yuan. File deduplication with cloud storage file system. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 280–287, Dec 2013.

[18] Xun Zhao, Yang Zhang, Yongwei Wu, Kang Chen, Jinlei Jiang, and Keqin Li. Liquid: A scalable deduplication file system for virtual machine images. *Parallel and Distributed Systems, IEEE Transactions on*, 25(5):1257–1266, May 2014.

[19] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 331–344, Berkeley, CA, USA, 2015. USENIX Association.

[20] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Read-performance optimization for deduplication-based storage systems in the cloud. *Trans. Storage*, 10(2):6:1–6:22, March 2014.

[21] Dirk Meister and Andre Brinkmann. Dedupv1: Improving deduplication throughput using solid state drives (ssd). In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST

'10, pages 1–6, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[23] Keonwoo Kim, Jeehong Kim, Changwoo Min, and YoungIk Eom. Content-based chunk placement scheme for decentralized deduplication on distributed file systems. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, CarmeloM. Torre, Hong-Quang Nguyen, David Taniar, BernadyO. Apduhan, and Osvaldo Gervasi, editors, *Computational Science and Its Applications, ICCSA 2013*, volume 7971 of *Lecture Notes in Computer Science*, pages 173–183. Springer Berlin Heidelberg, 2013.

[24] Fei Xie, Michael Condict, and Sandip Shete. Estimating duplication by content-based sampling. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 181–186, Berkeley, CA, USA, 2013. USENIX Association.

[25] Matsumiya Ryo, Sasaki Shin, Takahashi Kazushi, and Oyama Yoshihiro. ifarm: Implementing inline deduplication to a distributed file system. Technical report, University of Electro-Communications, Japan, 2014.

[26] Stephanie Jones. Online de-duplication in a log-structured file system for primary storage. Technical Report UCSC-SSRC-11-03, University of California, Santa Cruz, May 2011.

[27] S.N. Jones, A. Amer, E.L. Miller, D.D.E. Long, R. Pitchumani, and C.R. Strong. Classifying data to reduce long term data movement in shingled write disks. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–9, May 2015.

[28] A. Wildani, E.L. Miller, and O. Rodeh. Hands: A heuristically arranged non-backup in-line deduplication system. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 446–457, April 2013.

[29] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[30] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving i/o performance using virtual disk introspection. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 11–11, Berkeley, CA, USA, 2013. USENIX Association.

[31] Vasily Tarasov, Sonam Mandal, Philip Shilane, Deepak Jain, Geoff Kuenning, Karthikeyani Palanisami, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication appears in the proceedings of the 2014 ottawa linux symposium (ols'14).

[32] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, Santa Clara, CA, February 2016. USENIX Association.

[33] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[34] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*,

SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009. ACM.

[35] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 291–299, Berkeley, CA, USA, 2016. USENIX Association.

[36] Yannis Klonatos, Thanos Makatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Transparent online storage compression at the block-level. *Trans. Storage*, 8(2):5:1–5:33, May 2012.

[37] 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*, SYSTOR '09, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[38] N. Kumar, S. Antwal, G. Samarthyam, and S. C. Jain. Genetic optimized data deduplication for distributed big data storage systems. In *2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)*, pages 7–15, Sept 2017.

[39] H. Kamboj and B. Sinha. Dedup: Deduplication system for encrypted data in cloud. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 795–800, May 2017.

[40] Zhe Sun, Jun Shen, and Jianming Yong. Dedu: Building a deduplication storage system over cloud computing. In *Computer Supported Cooperative Work in Design (CSCWD), 2011 15th International Conference on*, pages 348–355, June 2011.

[41] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

[42] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 1–10, New York, NY, USA, 2008. ACM.

[43] B. Zhang, C. Wang, B. B. Zhou, and A. Y. Zomaya. Inline data deduplication for ssd-based distributed storage. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 593–600, Dec 2015.

[44] D. Zhang, C. Liao, W. Yan, R. Tao, and W. Zheng. Data deduplication based on hadoop. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 147–152, Aug 2017.

[45] Q. Liu, Y. Fu, G. Ni, and R. Hou. Hadoop based scalable cluster deduplication for big data. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 98–105, June 2016.

[46] DJ Lipman and WR Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[47] W R Pearson and D J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[48] Aisling ODriscoll, Jurate Daugelaite, and Roy D. Sleator. big data, hadoop and cloud computing in genomics. *Journal of Biomedical Informatics*, 46(5):774 – 781, 2013.

[49] National Human Genome Research Institute. DNA sequencing costs. https://www.genome.gov/sequencingcostsdata/.

[50] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in hpc storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 7:1–7:11, Los Alamitos, CA,

USA, 2012. IEEE Computer Society Press.

[51] John S Bertram. The molecular biology of cancer. *Molecular Aspects of Medicine*, 21(6):167 – 223, 2000.

[52] Vincent Burrus and Matthew K Waldor. Shaping bacterial genomes with integrative and conjugative elements. *Research in Microbiology*, 155(5):376 – 386, 2004. Genome plasticity and the evolution of microbial genomes.