Connection Science

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/ccos20

A feature-based intelligent deduplication compression system with extreme resemblance detection

Xiaotong Wu , Jiaquan Gao , Genlin Ji , Taotao Wu , Yuan Tian & Najla Al-Nabhan

To cite this article: Xiaotong Wu , Jiaquan Gao , Genlin Ji , Taotao Wu , Yuan Tian & Najla Al-Nabhan (2020): A feature-based intelligent deduplication compression system with extreme resemblance detection, Connection Science, DOI: 10.1080/09540091.2020.1862058

To link to this article: https://doi.org/10.1080/09540091.2020.1862058

Published online: 21 Dec 2020.

CONNECTION SCIENCE
https://doi.org/10.1080/09540091.2020.1862058

A feature-based intelligent deduplication compression system with extreme resemblance detection

Xiaotong Wu[a], Jiaquan Gao[a], Genlin Ji[a], Taotao Wu[b], Yuan Tian[c] and Najla Al-Nabhan[d]

[a]School of Computer and Electronic Information, Nanjing Normal University, Nanjing, People's Republic of China; [b]Alibaba Group, Hangzhou, People's Republic of China; [c]Nanjing Institute of Technology, Nanjing, People's Republic of China; [d]Department of Computer Science, King Saud University, Riyadh, Kingdom of Saudi Arabia

ABSTRACT
With the fast development of various computing paradigms, the amount of data is rapidly increasing that brings the huge storage overhead. However, the existing data deduplication techniques do not make full use of similarity detection to improve the storage efficiency and data transmission rate. In this paper, we study the problem of utilising the duplicate and resemblance detection techniques to further compress data. We first present a framework of FIDCS-ERD, a feature-based intelligent deduplication compression system with extreme resemblance detection. We also introduce the main components and the detailed workflow of our compression system. We propose a content-defined chunking algorithm for duplicate detection and a Bloom filter-based resemblance detection algorithm. FIDCS-ERD implements the intelligent file chunking and the fast duplicate and resemblance detection. By extensive experiments over the real datasets, we demonstrate that FIDCS-ERD has better compression effect and more accurate resemblance detection compared to the existing approaches.

ARTICLE HISTORY
Received 28 July 2020
Accepted 3 December 2020

KEYWORDS
Data storage; resemblance detection; delta compression; deduplication compression

1. Introduction

With the fast development of multiple novel computing paradigms (e.g. edge computing Xu, Liu et al., 2020, big data Wang, Yang, Wang et al., 2021 and cloud computing Xu, Mo et al., 2020), the amount of data is rapidly increasing from various services and applications that brings the huge storage overhead. For example, International Data Corporation (IDC) has reported that it needs more than 19 ZB of different storage media to meet data storage demands from 2017 to 2025 (Cao et al., 2019). Faced with so many storage demands, it is necessary to adopt some useful measurements to reduce the size of data to be stored in cloud storage or other media (Hosam & Ahmad, 2019; Zeng, 2020). In order to improve the storage efficiency, compression is one of the most important techniques to reduce the size of the data, which contributes to reducing storage demands and increasing transmission ratio (Wen et al., 2016). In general, there are two typical types of compression techniques,

CONTACT Taotao Wu wuxiaotong@njnu.edu.cn, wutaotaoxpy@gmail.com

i.e. data reduction and data deduplication, which are applied to different storage scenes. Data reduction uses a dictionary model to identify redundancy for short strings, such as LZ77/LZ88 (Ziv & Lempel, 1977, 1978). Data deduplication utilises the duplication and similarity between two data chunks to only store part of a chunk (Shilane et al., 2012). Relatively speaking, data deduplication is more scalable and efficient than data reduction.

In recent years, there have been a series of research works to improve storage efficiency by various data deduplication techniques (Aronovich et al., 2009, 2016; Cao et al., 2019; Muthitacharoen et al., 2001; Shilane et al., 2012; Xia et al., 2016, 2014; Zhang et al., 2019). Data deduplication is a lossless compression technique and makes full use of the correlation between any two data streams, such as resemblance and duplication. The advanced technique could greatly decrease the size of data that needs to be stored and transmitted. Naturally, it is attracting a lot of attention from academia and industry. For example, researchers apply data deduplication to various applications, including the network file system (Cao et al., 2019; Muthitacharoen et al., 2001) and cloud backup (Fu et al., 2019; Mao et al., 2012; Tan et al., 2011). Meanwhile, data deduplication is widely applied to multiple storage media in industry, such as memory and solid state disk (SSD) (Liu et al., 2018; Wang et al., 2018). The existing research has demonstrated that data deduplication is an extremely potential technique for storage systems to improve the storage efficiency and reduce the cost of hardware.

Although it provides the good compression performance, data deduplication is computationally intensive due to *duplicate detection* and *resemblance detection*. In general, a data stream is firstly divided to a set of small chunks to improve the possibility of duplication and resemblance. Duplicate detection distinguishes the same chunks, while resemblance detection finds the most similar chunk. The core problem is *how to fast distinguish the duplicated or similar chunk from the existing chunks*. Since the number of existing chunks is extremely huge, it is unrealistic and impossible to directly compare any two chunks byte by byte. In contrast, each chunk should be labelled by a fingerprint (i.e. feature) and a sketch. The feature of a chunk is just like someone's identification number, while the sketch is like his/her gene sequence. The feature is used to check out whether or not the chunk is duplicate, while the sketch is used to find which chunk is similar to the specified one. It is noted that for the same chunks, they should have the same features.

Unfortunately, there are two main challenges for duplicate and resemblance detection. For duplicate detection, it is important to divide data to multiple chunks. Insertion and deletion of a small number of bytes should not influence duplicate detection of the other chunks, which leads to *the boundary-shift problem* (BSP). For resemblance detection, the key is to find the similar chunk as much as possible, which is defined as *the resemblance detection problem* (RDP). Both of them have an important influence to determine the compression performance and the corresponding overhead (e.g. computation and index). In recent years, there have been a series of research works in both academia and industry to solve these two challenges (Muthitacharoen et al., 2001; Shilane et al., 2012; Zhang et al., 2019). Muthitacharoen et al. (2001) proposed a content-based breakpoint chunking algorithm in LBFS to properly chunk the data stream and solve the BSP. Shilane et al. (2012) utilised the sliding window and Rabin fingerprints to compute multiple super-features, which are used to detect the resemblance. Zhang et al. (2019) proposed Finesse, a fine-grained feature-locality-based fast resemblance detection approach. The above approaches provide a practical direction to solve the BSP and RDP.

However, there are still several disadvantages of the previous approaches that do not maximise the compression efficiency. For the boundary-shift problem, most of the previous works utilise a sliding window to compute a Rabin fingerprint (Rabin, 1981). If the Rabin fingerprint is equal to some predefined value, the position is set as a breakpoint. However, in some extreme cases, it is possible to find no breakpoints, since no Rabin fingerprint is equal to the value. This causes the minimal and maximal boundary problem (i.e. Min-BP and Max-BP). For the resemblance detection problem, they usually divide the chunk into multiple small sub-chunks, which each has a feature. Then, multiple features (e.g. 4) are combined to compute a super-feature (i.e. sketch). In general, a super-feature only represents a small number of sub-chunks. It has a high probability to find no similar chunk. These disadvantages greatly decrease compression efficiency of data streams.

In order to solve the above disadvantages of the previous methods, we propose FIDCS-ERD, a feature-based intelligent deduplication compression system with extreme resemblance detection. The system presents a new prefix matching method with multiple thresholds. It greatly decreases the probability to cause Max-BP. Meanwhile, the system constructs a special data structure based on Bloom filter (Luo et al., 2019) to record features of more sub-chunks, which is called sketch. The sketch consists of more information about some chunk and has higher probability to find the similar chunk. To the best of our knowledge, there are few works to maximise the compression efficiency for data streams. In summary, the contributions of the paper can be summarised as follows:

- We present a novel framework FIDCS-ERD to compress data streams with duplicate and resemblance detection. The framework consists of multiple core components and the related workflow for data compression.
- For duplicate and resemblance detection, we propose an efficient prefix matching chunking algorithm to solve BSP, Min-BP and Max-BP and a new feature-based algorithm to implement the extreme resemblance detection. The algorithms improve the compression efficiency of data.
- We evaluate our algorithms by extensive experiments on the real datasets. Experimental results show that our system has the high compression performance and the low computational overhead.

The remaining part of the paper is listed as follows. Section 2 presents the related work about data deduplication. Section 3 introduces an overview of our compression system. Section 4 implements the key components for the compression system. Section 5 evaluates the performance of the system by multiple metrics. Section 6 concludes the paper.

2. Related work

How to improve transmission and storage efficiency is always an important problem, especially in cloud storage and SSD. For example, Mao et al. (2019) proposed the IOFollow scheme to increase the migration performance of virtual machines in cloud. Wu, Li et al. (2019) proposed request redirection by exploiting the asymmetric read-write performance characteristics of SSDs and the hot-spare SSD in SSD-based RAIDs to alleviate the GC-induced performance variability. In special, data deduplication is an effective technique to compress data, which is introduced by the following subsections.

2.1. Data reduction and deduplication

Data reduction and deduplication are the most common techniques of data compression to improve storage efficiency. Data reduction utilises a dictionary model to identify redundancy for short strings, such as LZ77/LZ88 (Ziv & Lempel, 1977, 1978). In general, the technique first computes a weak hash of strings and then compares the hash-matched strings byte by byte. Although they have a certain amount of compression efficiency to reduce redundant data, the dictionary-model-based techniques have relatively high computational overhead and space complexity. As a result, the techniques are always applied to compress data in a small region, which weakens the compression power to improve processing speed. The other typical data reduction approaches consist of LZO (Kane & Yang, 2012), LZW (Gawrychowski, 2013) and DEFLATE (Harnik et al., 2014).

Data deduplication uses the duplication and similarity between two data chunks to only store part of a chunk (Shilane et al., 2012). For example, Mao et al. (2019) proposed a deduplication-assisted cloud storage to improve the storage efficiency and network bandwidth. Compared to dictionary-model-based data reduction, data deduplication has two remarkable features. The first one is that it eliminates redundant data at the chunk-level (e.g. 8 KB) or file-level, while data reduction implements the similar objective at the string or byte level. The second one is that it identifies duplicate or similar data (files or chunks) by calculating its cryptographically secure hash-based fingerprints while data reduction achieves them with byte-by-byte comparison. The features implies that data deduplication is more scalable and efficient than data reduction. More importantly, data deduplication is fully suited to global data compression in large-scale storage systems.

2.2. Duplicate detection

Chunking is the first component in a data deduplication system that has an important influence of the following duplicate and resemblance detection on compression performance. For duplicate detection, the key operation is to divide the data stream to multiple chunks, which each is labelled as a fingerprint. The same chunks have the same fingerprints. There are two classes of chunking algorithms, i.e. fixed-size chunking (FSC) and content-defined chunking (CDC). The former has the boundary-shift problem, which greatly reduces the possibility of duplicate detection. The latter utilises the sliding window technique to get the breakpoints. Muthitacharoen et al. (2001) utilised the sliding window and the Rabin fingerprints to propose the first CDC algorithm to solve the boundary-shift problem. On this basis, there are a series of works to modify the algorithm to improve the performance. The objective of the works (Agarwal et al., 2010; Xia et al., 2014; Yu et al., 2015; Zhang et al., 2015) is to improve the speed of the CDC algorithm due to its computationally intensive. Some works (Bobbarjung et al., 2006; El-Shimi et al., 2012) paid attention on the restrictions on the max/min chunk size to solve the Max-BP and Min-BP and improve the effect of duplicate detection. The works (Drago et al., 2012; Lu et al., 2010; Zhou & Wen, 2013) investigated to re-chunk the non-duplicate chunks to find more redundancy.

There are still two challenging problems in duplicate detection that needs to be improved. In order to overcome Max-BP and Min-BP and improve compression effect, the general operation of the existing approaches is the restrictions on the max/min chunk size. However, it has a limited improvement for the compression ratio and is necessary to further improve the compression ratio. Besides, there are few works to consider the special case, in which fingerprints are stored in the disk since the memory cannot support so much fingerprints. It is challenging to fast find the duplicate chunk by the fingerprint index.

2.3. Resemblance detection

The resemblance detection approaches can be divided into two classes, i.e. the file- and chunk-level. For the file-level, the general operation is to compare all the fingerprints of two files and get their similarity (Douglis & Iyengar, 2003). In the paper, we are focused on resemblance detection at the chunk level. The main idea of the previous works is to compute multiple super-features for each chunk, which each is computed by multiple fingerprints. Shilane et al. (2012) utilised Rabin fingerprints to compute multiple super-features, which are used to detect the resemblance. By rigorous theoretical analysis and extensive experiments, Xia et al. (2016) demonstrated that using fewer features per super-feature not only reduces the computational overhead but also gets more resemblance. Since resemblance detection needs a certain amount of computational overhead, Zhang et al. (2017) proposed a scheme that electively performs delta compression after identifying the non-base-fragmented chunks. Zhang et al. (2019) proposed Finesse, a fine-grained feature-locality-based fast resemblance detection approach to ensure the effect of resemblance detection and decrease the computational complexity. In order to decrease the computation, Aronovich et al. (2009) directly selected multiple features from Rabin fingerprints of some chunk as its super-features.

In the existing approaches, the nature of resemblance detection is to compare a certain number of Rabin fingerprints of two chunks. When the number of same fingerprints exceeds some threshold, two chunks are similar. However, compared to the size of a chunk, there are few sub-chunks that are chosen to represent the whole chunk. In general, there are less than 20 fingerprints compared for resemblance detection when the number of super features is 4, which each has 5 fingerprints. As a result, these approaches use a small number of fingerprints and do not take full advantage of the fingerprints to get the optimal compression efficiency.

2.4. Deep and delta compression

For other types of data, there are two classes of compression methods, i.e. delta and deep compression. Deep compression (e.g. LZW Nelson, 1989, GZIP Gailly & Adler, n.d.) utilises the compression approaches based on data reduction to compress a chunk. There are two classes of coding models for lossless data compression, including statistical model-based coding (byte-level) and dictionary model-based coding (string-level). The typical methods in the former are Huffman and Arithmetic coding, while the latter has LZW (Nelson, 1989) and LZ77. Delta compression utilises an existing chunk to compress a new chunk and only stores the difference of the latter. For example, Xdelta (MacDonald, 2000) utilises a byte-wise sliding window to identify the matched strings between the base and the input chunks for the delta calculation. Zdelta (Trendafilov et al., 2002) utilises the Huffman coding to further improve the delta compression by eliminating redundancy. Ddelta (Xia et al., 2014) proposed a deduplication-inspired delta-encoding approach that put similar chunks into independent and nonoverlapped strings by a fast Gear-based content-defined chunking algorithm.

Data deduplication has been widely applied to various scenes, including backup storage systems and media (Fu et al., 2019; Liu et al., 2018; Mao et al., 2012; Tan et al., 2011; Wang et al., 2018). For example, some works (Fu et al., 2019; Mao et al., 2012; Tan et al., 2011) attempt to construct deduplication-based storage systems for cloud backup systems. The other works investigate the practical applications to apply data deduplication algorithms to SSD and main memory (Liu et al., 2018; Wang et al., 2018). These storage systems and media do not consider data format. Therefore, they are suited to multiple applications, such as data trade (Bloembergen et al., 2015), network traffic analysis (Fang et al., 2018; Liu et al., 2019), feature selection (Nayak et al., 2018; Wang, Yang, Song et al., 2020; Xu, Shen et al., 2020), service computing (Wang et al., 2019; Xu, Zhang et al., 2020).

3. FIDCS-ERD

3.1. System framework

Here, we introduce an overview of FIDCS-ERD, which consists of two important functions, i.e. **duplicate detection** (DD) and **resemblance detection** (RD). The former is to detect duplicate chunks, while the latter is to detect similar ones. That is, for duplicate chunks, FIDCS-ERD just stores one physical chunk, which is also indexed by any other same chunks. In addition, for highly similar chunks, the system only stores their differences so as to remove the redundant data. By DD and RD, it greatly reduces data redundancy and increases storage efficiency. On the other hand, it causes a certain amount of extra computational overhead.

Figure 1 presents a framework of FIDCS-ERD. For a file (or a data stream) to be stored, it is firstly divided to multiple small chunks of equal or variable sizes (i.e. chunking). The following operations are to detect whether or not these chunks are duplicate (i.e. DD) or similar (i.e. RD) to the existing chunks that have been stored in the storage media. In general, it is unrealistic to directly compare two chunks byte by byte, because it needs a large amount of time and computational overhead. Instead, the system uses a *fingerprint* to represent a unique chunk and a *sketch* to indicate the similarity. The fingerprint is just like someone's identification number, while the sketch is like his/her gene sequence. It is challenging to design the proper fingerprint and sketch for any chunk which satisfy some specified requirements.

In general, there are two metrics to evaluate whether or not fingerprints are proper. The first one is that for any two same chunks, their fingerprints need to be same. The second one is that for any two different chunks whatever difference there might be, their fingerprints need to be different. It is worthwhile noting that the length of a fingerprint is short (e.g. 64 bits Muthitacharoen et al., 2001). The system just evaluates the duplicate between any pair of chunks by the comparison of their fingerprints instead of the content. Therefore, it is key to choose a proper mathematical model to solve the above problems. To this end, the most of the existing works utilise cryptographic hash-based fingerprints (e.g. SHA-1 or SHA-256) to compute a fingerprint of a chunk (Wen et al., 2016). That is, the fingerprint of a chunk is computed as follows:

$$\text{fingerprint} = \textbf{hash}(\text{chunk}),$$

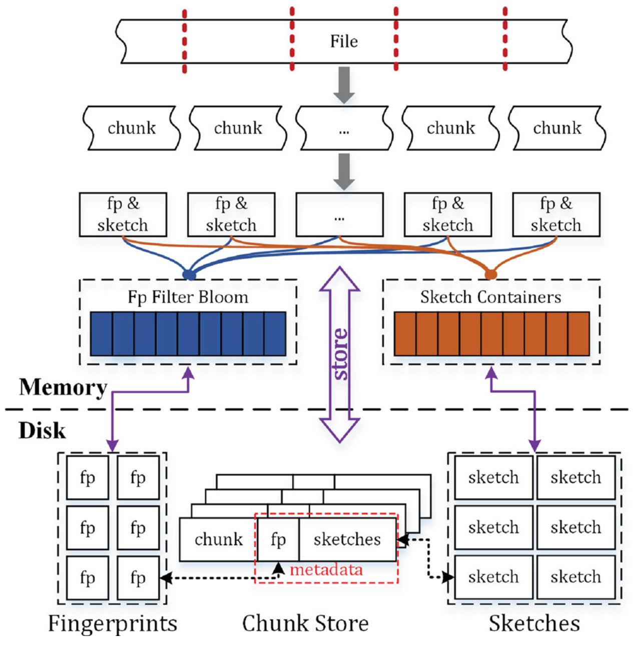

Figure 1. An overview of FIDCS-ERD. fp means the fingerprint representing an identifier of a chunk, which is used to identify the duplicate chunk. Sketch is a Bloom filter-based array, which is compared to find the similar chunk.

in which **hash**$(\cdot)$ may be SHA-1, SHA-256 or SHA-512.

However, there is a very small probability to cause hash collision, meaning that different chunks have the same fingerprints. For various hash algorithms, Wen et al. (2016) analysed hash collision probability with different sizes of data under an average chunk size of 8 KB. In the worst case, the probability is 10^{-8} with SHA-1 under size of 1 YB. Meanwhile, the probability decreases to 10^{-37} with SHA-256 for the same size. On the other hand, SHA-256 needs more computational workload. Therefore, the deduplication system selects the most proper hash algorithm according to its storage and computation capability. For hash algorithm SHA-1, SHA-256, SHA-512, the output length is 160, 256 and 512 bits, respectively. In general, the system extracts the first ξ bits of the output as the fingerprint of a chunk. For example, Muthitacharoen et al. (2001) indexed each chunk by the first 64 bits of SHA-1 hash value in the network file system.

Here, we use the following example to illustrate the size of fingerprints in a storage system.

Example 3.1: Assume that there is a storage system with space size of 1 TB. The average size of each chunk is 4 KB. The length of each fingerprint is 64 bits. In the worst case in which there is no same chunk, the total size of fingerprints is $1\,\text{TB} \cdot 64\,\text{b}/4\,\text{KB} = 2\,\text{GB}$ with the number of 2^{28}. Furthermore, if storage space is 1 PB, the total size of fingerprints is up to 2 TB.

By the above example, we find that it is difficult for the memory of a storage system to process so many fingerprints when data streams are very huge. The part or all of fingerprints need to be stored in the storage media. If so, the time to traverse the existing fingerprints in the storage media is too long. In order to reduce the time, we utilise *Bloom filter* (Luo et al., 2019) to check whether or not there is a specified fingerprint. The concrete implementation for fingerprint index is shown in Section 4.2.

The fingerprints are used to fast compare whether or not two chunks are duplicated. For two different chunks, there may be further compressed if these two chunks are similar. To this end, *delta compression* is a practical and efficient technique to compress a similar chunk. Here, the difficulty is to check the similarity between any two chunks. Similar to duplicate detection, resemblance detection does not directly compare the content of any two chunks. In contrast, it uses sketch to show the degree of the similarity among chunks. In the previous works (Aronovich et al., 2009; Shilane et al., 2012; Zhang et al., 2019), the sketch is represented as a list of features or super-features based on a group of hash values (e.g. Rabin fingerprints Rabin, 1981). Different from them, we compute an array containing only 0 and 1 with a fixed length based on Bloom filter. The similarity between any two chunks is evaluated by their sketches. Similar to fingerprints, sketches also need a certain amount of storage space and index overhead. The concrete implementation is shown in Section 4.3. It is worthwhile noting that the fingerprint and the sketch of a chunk are combined as its *metadata* with other chunk information.

3.2. Processing flow of **FIDCS-ERD**

When a file is stored to the disk, there are a lot of operations that should be executed, which is shown in Figure 2. The concrete process is divided into four parts, including chunking, duplicate detection, resemblance detection and compression. Here, we briefly introduce the processing flow of FIDCS-ERD.

The function of chunking is to divide a file to multiple chunks with same or variable sizes, which decides the effect of duplicate and resemblance detection. In addition, the average size of a chunk has the important influence on computation and storage workload and compression effect. In other words, the smaller the average size, the higher the workload of computation and storage is and the better the compression effect is. On the one hand, the more chunks imply more fingerprints and sketches, which need more computation and storage. On the other hand, there is a higher probability to find the duplicate or the resemblance. In Section 4.1, we will investigate the influence of the chunk size on DD and RD and present the concrete chunking algorithm. Once the system finishes the division of a file, the corresponding fingerprints and sketches of the chunks are computed.

When a storage system stores vast amounts of data, there are a large number of fingerprints for chunks. It is difficult for the memory to support so many fingerprints. Therefore, in our designed deduplication system, we store all fingerprints in the disk. However, it has greatly poor efficiency to search a fingerprint in the disk. The idea of our system is to utilise FP-BF (i.e. a Bloom filter-based data structure) to fast check whether or not the fingerprint is in the disk. If the result is "No", it implies that there is not a duplicate chunk in the existing chunks. The following step is to check whether or not the chunk is similar to the existing chunks. If the result is "Yes", since FP-BF is not accurate, it needs to check whether or not the chunk is indeed duplicated. In order to reduce query time, the system will put a certain number of recently used fingerprints in the cache in advance. If the fingerprint is in the cache, it implies that it is a duplicate chunk. Otherwise, it seeks whether or not the fingerprint is

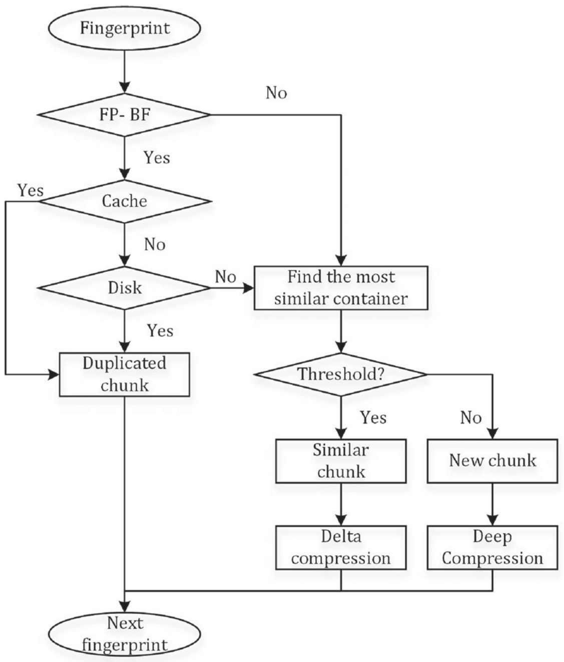

Figure 2. The processing flow in FIDCS-ERD.

in the disk. By the above operations, the system is able to check whether the fingerprint is duplicate. For different chunks, the next step is to check its similarity with the existing chunks. For duplicate chunks, it needs no extra operations.

For a non-duplicate chunk, it is possible to save storage space by the delta compression technique. That is, if there are two similar chunks, the second chunk just stores the difference between two chunks. In this way, it improves the storage efficiency. Here, the challenge is to find the most similar chunk for a specified one. In order to solve the problem, the system utilises the Bloom filter technique to compute an array containing only 0 and 1 named Sketch. The resemblance detection is changed to compare the sketches of chunks. Meanwhile, all the sketches are stored in the disk. In order to reduce detection time, the system first clusters the similar chunks as a container and then chooses a chunk as a representative. All the representatives are put into the cache. For a coming chunk, the system just compares it with the representative and gets the most similar chunk. Next, the system checks whether or not the degree of the similar chunk exceeds some threshold. If exceeding, the chunk will be stored by delta compression. Otherwise, it is a new chunk.

By duplicate and resemblance detection, there are three possible cases, including duplicate, similar or new chunk. Different cases have different compression methods listed as follows.

- For a duplicate chunk, it is no need to compress it. It just remembers the fingerprint of the chunk and storage location in the storage media.

- For a similar chunk, it needs to execute delta compression and store the difference, which is named by random string with a fixed length. In addition, it indexes the location of both source chunk and the difference.
- For a new chunk, the system compresses it by some deep compression algorithm and then stores the compressed data. The compressed file is named by the unique fingerprint.

Therefore, the system improves the storage efficiency by various types of compression. Of course, it always increases computational overhead so as to cause a certain amount of delay. It is worthwhile to deal with the balance between compression performance and computational efficiency.

4. Implementations

In the section, we introduce the details of the key components in FIDCS-ERD, including chunking, duplicate and resemblance detection, and compression. In particular, we present an extreme resemblance detection algorithm to make full use of the similarity between chunks and maximise the compression effect. Assume that there is a coming data stream or file $\mathbf{B} = \{B_0, B_1, \ldots, B_{n-1}\}$ with the size of n, in which $B_i \in [0, 255]$ is an integer. We summarise the key notations in Table 1. In the following paper, we might omit the symbols for simplicity if thus it would not lead to the ambiguity.

4.1. Chunking

Chunking of the data stream is the first important step in a deduplication system. Its main function is to divide a file or a data stream into multiple small data chunks with same or variable sizes. Here, we present our chunking algorithm based on content-defined chunking.

4.1.1. Content-defined chunking

In general, the existing chunking algorithms are divided to two classes, i.e. fixed-size chunking (FSC) and content-defined chunking (CDC). In FCS, it usually divides data streams or files to multiple small chunks of a fixed size. Although this method is simple and convenient, deletion or insertion of a byte of the source data leads to the modification of all chunks,

Table 1. Description of notations.

Notation	Description
$\mathbf{B}$	A data stream which is a set of integers $\{B_0, \ldots, B_{n-1}\}, B_i \in [0, 255]$
w	The size of the sliding window
α	The average size of the chunk
p	The primer to compute Rabin function
β	The minimal size of the chunk
γ	The maximal size of the chunk
$\mathcal{H}$	A set of independent hash functions $\{h_1, \ldots, h_k\}$
ℓ	The number of bits of each hash function h_i
m	The number of bits of sketch
τ	The threshold of common bits over two sketches
z	The number of chosen Rabin fingerprints for sketch

which is defined as *the boundary-shift problem*. That is, deletion or insertion of a byte for a file generates fully different chunks, which greatly decreases the ratio of the same chunks. In order to solve the problem, CDC uses a sliding-window technique on the content of files and then computes a hash value of the window (Muthitacharoen et al., 2001; Yu et al., 2015; Zhou & Wen, 2013). A breakpoint is determined if the hash value of this sliding window is equal to some predefined condition. The CDC-based algorithms are capable to improve the ratio of duplicated chunks. Therefore, our system selects the content-based chunking algorithm to implement chunking of data streams or files.

4.1.2. Chunking algorithm

In CDC-based algorithms, there is an important parameter, i.e. the average size of a chunk defined as α. The system utilises Rabin fingerprints to compute a hash value for a plurality of consecutive bytes (Rabin, 1981). That is, starting from the first byte, the system moves the window byte by byte. In each movement, it computes the Rabin fingerprint of the bytes in the sliding window. If its value is equal to some predefined value, the position is chosen as a breakpoint and thus the chunk is divided. The Rabin fingerprint in a sliding window of the size of w is computed as follows:

$$\mathrm{Rabin}(B_i, B_{i+1}, \ldots, B_{i+w-1}) = \left\{ \sum_{k=1}^{w} B_{i+k-1} \cdot p^{w-k} \right\} \bmod \alpha, \tag{1}$$

in which p is a primer. For simplicity, $\mathrm{Rabin}(B_i, B_{i+1}, \ldots, B_{i+w+1})$ is also written as $\mathrm{Rabin}(\mathbf{B}[i : i + w - 1])$.

In general, if the sliding window moves to the next byte, a simple and direct approach is to compute $\mathrm{Rabin}(\mathbf{B}[i + 1 : i + w])$ according to Equation (1). If so, it is inefficient to cause the huge computational overhead. In order to improve the efficiency, an alternative approach is to utilise the calculated value $\mathrm{Rabin}(\mathbf{B}[i : i + w - 1])$. Compared to $\mathrm{Rabin}(\mathbf{B}[i : i + w - 1])$, $\mathrm{Rabin}(\mathbf{B}[i + 1 : i + w])$ adds a new element B_{i+w} and removes an element B_i. Therefore, the concrete value of $\mathrm{Rabin}(\mathbf{B}[i + 1 : i + w])$ is derived as follows:

$$\begin{aligned}
&\mathrm{Rabin}(B_{i+1}, B_{i+2}, \ldots, B_{i+w}) \\
&\quad = \left\{ \sum_{k=1}^{w} B_{i+k} \cdot p^{w-k} \right\} \bmod \alpha \\
&\quad = \left\{ B_{i+w} - B_i \cdot p^{w} + p \cdot \sum_{k=1}^{w} B_{i+k-1} \cdot p^{w-k} \right\} \bmod \alpha \\
&\quad = \{ p \cdot \mathrm{Rabin}(B_i, B_{i+1}, \ldots, B_{i+w-1}) + B_{i+w} \\
&\qquad - B_i \cdot p^{w} \} \bmod \alpha \\
&\quad = \left\{ \begin{aligned} &\left[p \cdot \mathrm{Rabin}(B_i, B_{i+1}, \ldots, B_{i+w-1}) \right] \bmod \alpha \\ &+ \underbrace{B_{i+w} \bmod \alpha}_{\triangle} - \underbrace{\left[B_i \cdot p^{w} \right] \bmod \alpha}_{\triangledown} \end{aligned} \right\} \bmod \alpha
\end{aligned} \tag{2}$$

Algorithm 1 Computation of Breakpoints CBP

```
Input: data stream B, the size of sliding window $w$, the average size $\alpha$, the minimum size $\beta$, the maximum size $\gamma$, the number of features f_n, cryptographic hash function $h$, threshold $\sigma$
Output: a set of breakpoints bps
1: set bps, fps and c_bps as $\emptyset$, off and len as 0
2: $n \leftarrow$ Size(B)
3: if $w > n$ then
4:     fps $\leftarrow$ Rabin(B, off, $n$, $\alpha$)
5:     return [($h$(B), off, $n$, SER(fps, 1))]
6: end if
7: fp $\leftarrow$ Rabin(B, offset, $w$, $\alpha$)
8: len $\leftarrow$ len + $w$
9: for $i \in [w, n-1]$ do
10:     fp $\leftarrow$ Rabin_next(fp, B, off, len, $w$, $\alpha$)
11:     fps $\leftarrow$ fp
12:     len $\leftarrow$ len + 1
13:     if $\beta \le$ len $< \gamma$ then
14:         if fp == Mask then
15:             id $\leftarrow$ $h$(B[off : off + len])
16:             bps $\leftarrow$ (id, off, len, SER(fps, f_n))
17:             offset $\leftarrow$ offset + len
18:             set len as 0, fps and c_bps as $\emptyset$
19:         else if Prefix(fp, Mask) $\ge \sigma$ then
20:             $p$ $\leftarrow$ Prefix(fp, Mask)
21:             c_bps $\leftarrow$ (p, off, len, SER(fps, f_n))
22:         end if
23:     else if len $\ge \gamma$ then
24:         if c_bps $\ne \emptyset$ then
25:             b_l $\leftarrow$ SelectBreakpoint(c_bps)
26:             id $\leftarrow$ $h$(B[off : off + b_l])
27:             bps $\leftarrow$ (id, off, b_len, SER(fps[: b_l], f_n))
28:             off $\leftarrow$ b_l + 1
29:             len $\leftarrow$ len − b_l
30:             fps $\leftarrow$ fps[b_l + 1 :]
31:             c_bps $\leftarrow$ candidate breakpoints after b_l
32:         else
33:             bps $\leftarrow$ (off, len, SER(fps, f_num))
34:             set off and len as 0, fps and c_bps as $\emptyset$
35:         end if
36:     end if
37: end for
38: if fps then
39:     bps $\leftarrow$ (off, len, SER(fps, f_num))
40: end if
41: return bps
```

By the above derivation, the computational overhead of Rabin fingerprint greatly decreases. In order to further improve the computation efficiency, we modify the calculation method of Equation $\triangle$ and $\triangledown$. In order to reduce the computation, we use pre-computed tables $T_{\triangle}$ and $T_{\triangledown}$ to record each value of Equation $\triangle$ and $\triangledown$, respectively. By a series of improvements, we reduce the computation workload of Rabin fingerprint.

However, there may be two extreme cases that influence the effect of chunking listed as follows:

- **Minimal Boundary Problem (Min-BP).** It is possible to generate a lot of chunks, each of which has an extreme small size. For these small chunks, it is not worth executing further operations in terms of compression effect and cost. Therefore, the system needs to set a minimal size of each chunk.
- **Maximal Boundary Problem (Max-BP).** It is also possible to generate a lot of chunks, each of which has an extreme big size. If the size of a chunk is big, the probability to find a duplicated and similar chunk is low. Finally, it greatly reduces compression effect of the system.

In order to solve the above problems, it is a need to set a forced lower and upper boundary (El-Shimi et al., 2012; Muthitacharoen et al., 2001). As a result, there are two corresponding parameters, i.e. β and γ. The former sets a minimal value of the size of a chunk, while the latter limits the maximum of the size.

For the minimal boundary problem, a simple method is to skip the judgment of a breakpoint when the length is less than β. However, for the maximal boundary problem, it is inefficient to adopt the same method with the minimal boundary. Here, we propose a prefix matching method to get the breakpoints when the length of a chunk exceeds the value γ. The idea is to compare the prefix of a Rabin fingerprint and a given value $\mathcal{V}$. For example, if $\mathcal{V} = 11111111$ and the length of the prefix is 6, the system records all the breakpoints, whose first 6 integers are equal to 111111, i.e. $\mathcal{V}[:6] = \text{Rabin}(\cdot)[:6]$. In particular, if there are multiple breakpoints, we prefer to select the breakpoint with the longest prefix.

4.1.3. Chunking size

In the CDC-based algorithm, the average size of the chunk is an important parameter that has a great influence on the following duplicate and resemblance detection. Here, we take Figure 3 as an example to illustrate the influence. We compare two cases of different average sizes of chunks, i.e. 4 and 8 KB. For simplicity, assume that a chunk with size of 8 KB is combined by two adjacent chunks with size of 4 KB, which are source chunks in Figure 3. Compared to source chunks, there are 6 possible cases for the coming chunk, i.e. (a)–(f) in Figure 3. It is obvious to find that when the average size of a chunk changes from 4 to 8 KB, there are three different cases, including (b), (d) and (e), listed as follows.

- In case (b), the combination of a duplicated chunk and a similar chunk is changed to a similar chunk, which implies delta compression for a duplicate. It reduces the compression efficiency and increases the delay due to compression and decompression.
- In case (d), a chunk is composed by a duplicate and a different sub-chunk. Compared to source chunks, there are two possible detection results, i.e. a similar or different chunk. Obviously, the latter reduces the compression efficiency.
- In case (e), similar to case (d), a chunk is composed by a similar chunk and a different chunk. The resemblance detection result “No” implies the decrease of compression efficiency.

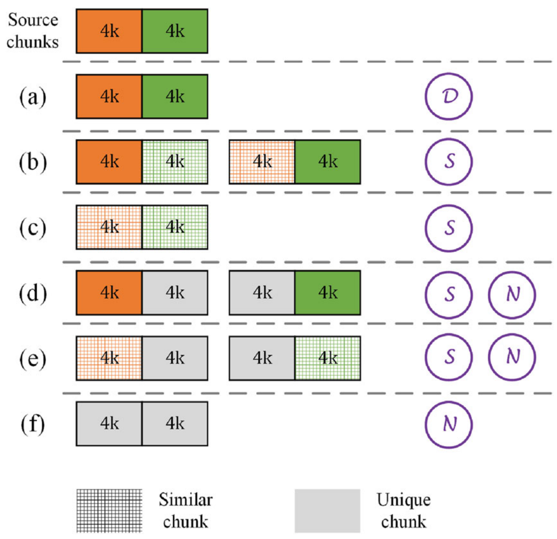

Figure 3. The influence of the average size of a chunk on duplicate and resemblance detection. D, S and N represent the duplicate, the similar, and the unique chunk, respectively.

Although the smaller chunks imply better compression effect, it also increases the number of fingerprints and sketches so as to add the overhead of computation and index. Therefore, it is necessary to consider the balance between data compression efficiency and index overhead. Algorithm 1 shows the computation of breakpoints of a data stream or file.

4.2. Fingerprint detection

As illustrated in Example 3.1, when a storage system supports data with size of 1 PB, the total size of fingerprints is up to 2 TB in the worst case. Therefore, it is impossible and unrealistic for a memory to support so much data. A feasible approach is that all the fingerprints are stored in the disk and frequently used parts of them are in the memory. Besides, we utilise Bloom filter to fast screen out a fingerprint, which is different from all the fingerprints.

4.2.1. Bloom filter

A Bloom filter is a special data structure to represent a group of elements and support their queries (Luo et al., 2019). It implements efficient space utilisation and has a drawback of false positives. That is, an element in a Bloom filter does not imply its existence. On the other hand, it does not exist if an element is not in the Bloom filter. Therefore, according to the property of Bloom filter, it is applied to checking the nonexistence of a specified fingerprint.

Assume that there are s unique fingerprints and a set of independent hash functions $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$. A Bloom filter has ℓ bits, each of which is 0 or 1. Figure 4 shows three basic operations of Bloom filter, including *initialization*, *insertion* and *lookup*. At first, all the entries of a Bloom filter is initialised to 0. When a fingerprint is coming, an entry is set as 1 at a specified position, which is computed by hash function h_i. When checking whether or not an element is in the set, it just compares whether or not all the entries indexed by hash functions are 1. In Figure 4, y_1 is not in the set since its value is $(0, 1, 0)$, while y_2 is

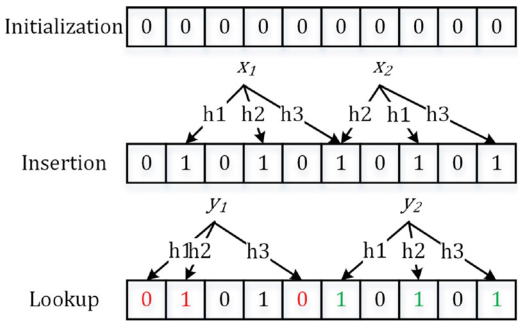

Figure 4. Basic operations of Bloom filter.

in the set. Here, the objective is to set the proper values of parameters s, ℓ, k to minimise the false-positive rate. Broder and Mitzenmacher (2003) demonstrated that in order to get the minimum of the false positive rate, $k = \ln 2 \cdot (s/\ell)$ and then the minimum is $(1/2)^k \approx (0.6185)^{s/\ell}$. For example, the false-positive rate is about 2.14% with $s/\ell = 8$ and 0.82% with $s/\ell = 10$. In our experiments, we choose $s/\ell = 8$ and $k = 5$.

4.2.2. Fingerprint index

As described in Section 3, there are three main steps for duplicate detection, including (i) constructing a Bloom filter-based data structure (i.e. FP-BF) to fast check the non-existence of a specified fingerprint; (ii) inquiring whether or not the fingerprint is in the memory; and (iii) checking whether or not the fingerprint is in the disk.

Here, we illustrate how to choose proper fingerprints in the memory, which has an important influence on the efficiency of duplicate detection. In the memory, we allocate a space $\mathcal{S}$ of a specified size to store all of or a part of fingerprints. When a fingerprint is indeed in the disk, it is efficient to detect it by step (ii), but time consuming by step (iii). Therefore, the focus is to avoid searching the disk as much as possible. When there is still free space in $\mathcal{S}$, a new fingerprint is directly added to the memory. However, when $\mathcal{S}$ is full, it needs to update the fingerprints in the memory. Our strategy is that when the size of $\mathcal{S}$ exceeds some threshold, the system will clear the fingerprints in $\mathcal{S}$, which are not matched in a long time.

For a unique fingerprint, it needs to be stored the disk. In order to reduce the time of insertion, the system does not immediately insert it to the disk when a new fingerprint is coming. In contrast, the new fingerprint is first stored in the memory. When the number of unique fingerprints is up to some given threshold, all of them will be stored in the disk. In this way, the system reduces the number of new fingerprints added to the disk so as to decrease the insertion time and improve the efficiency.

4.3. Resemblance detection

In many scenes, there are a large number of different but similar files or data streams. For example, in an archive platform for an open source software (e.g. Linux Kernel Organization, 2020), it stores all the codes and related documents of various versions. For two adjacent versions, it is possible to just update a small part of each file. For these files, it is

Algorithm 2 Sketch with Extreme Resemblance SER

```
Input: a set of Rabin fingerprints bps, the length of Bloom filter $m$, hash functions $\mathcal{H} = \{h_1, h_2, \cdots, h_k\}$, the number of chosen Rabin fingerprints $z$
Output: sketch
1:  bps_n ← Size(bps)
2:  sketch ← boolean array with length of $m$, initially 0
3:  unit_n ← bps_n/$z$
4:  for $i \in [z]$ do
5:      pos ← arg max {bps[$i$ ∗ unit_n : ($i$ + 1) ∗ unit_n]}
6:      pos ← (pos + 4)/unit_n + $i$ ∗ unit_n
7:      for $j \in [k]$ do
8:          sketch[$h_j$(bps[pos])/$m$] ← 1
9:      end for
10: end for
11: return sketch
```

efficient and reasonable to store the difference between two similar files, instead of storing them again. In this way, it greatly improves the storage efficiency. To this end, resemblance detection is proposed to find the most similar chunk for some unique one and then store the difference by delta compression.

4.3.1. Sketch computation

In recent years, there have been a lot of works to research resemblance detection of chunks (Aronovich et al., 2009, 2016; Shilane et al., 2012; Xia et al., 2016; Zhang et al., 2019). In general, most of them have three main steps, including (i) dividing the chunk to multiple sub-chunks; (ii) computing a feature of each sub-chunk based on Rabin fingerprints; (iii) combining features to a set of super-features. Meanwhile, the algorithms proposed by them can be divided into two classes according to the computation of features. The first one (i.e. feature-based resemblance detection, FRD) is to directly choose a Rabin fingerprint as a feature (Aronovich et al., 2009). The second one (i.e. super-feature-based resemblance detection, SRD) is to combine a certain number of fingerprints to a feature (Shilane et al., 2012; Zhang et al., 2019). *FRD has the lower computation, while SRD has the higher resemblance rate but more complex computation.* Therefore, we take advantage of these two classes for resemblance detection and propose our own detection algorithm.

In the existing algorithms (Aronovich et al., 2009; Shilane et al., 2012; Xia et al., 2016; Zhang et al., 2019), the sketch is a set of features or super-features with size of 4, each of which is computed by the combination of Rabin fingerprints in a sub-chunk or a whole chunk. However, they do not make full use of the similarity between two chunks since their nature is to choose a very small proportion of Rabin fingerprints to compare the resemblance. Here, we illustrate the disadvantages by the following example.

Example 4.1: Assume that the size of a chunk and a sliding window is 8 and 64 KB. Then, the number of Rabin fingerprints of the chunk is $8 * 1024 - 64 + 1 = 8129$. In FRD, it usually chooses 4 Rabin fingerprints as a sketch. In SRD, it computes 4 super-features, each of which is combined by 3 Rabin fingerprints. It is obvious to see that compared to all of Rabin fingerprints, the number of the chosen ones is very small.

The above example shows that there is still a space to improve the resemblance rate. To this end, we propose an extreme resemblance detection approach, which introduces the idea of the existing resemblance detection algorithms (Aronovich et al., 2009; Shilane et al., 2012; Xia et al., 2016; Zhang et al., 2019). In Algorithm 2, it first initials a Bloom filter with length m (line 2). The number of chosen Rabin fingerprints z is far greater than that used in the previous algorithms. We divide all the Rabin fingerprints into z blocks. For each block, we utilise the method in (Aronovich et al., 2009) to choose a specified Rabin fingerprint as a feature (line 5–6). That is, it first finds a position, in which the Rabin fingerprint has a maximal value in a block. Then, it chooses the Rabin fingerprint, whose position is offset of 4 relative to the maximum. Finally, the chosen fingerprint is added to the Bloom filter (line 7–9). Here, parameters m, k and z have the important influence on the accuracy of resemblance detection. Just as duplicate detection, we also set $k = \ln 2 \cdot (m/z)$, in which $m/z = 8$.

4.3.2. Sketch index

In FRD and SRD, two chunks are similar if they have at least one same super-feature. However, it is not suited for SER. Instead, for any pair of two super-features sp_1 and sp_2, we need to check whether or not the number of bits of 1 in $sp_1 \& sp_2$ exceeds some threshold. If we compare them one by one, the time complexity is very high. To this end, we propose a similar cluster approach to reduce the time of resemblance detection. That is, we first cluster extremely similar chunks in a container, which have similar super-features. Then, we select a chunk as a representative for each container. Next, for a coming chunk, it is just compared with these representatives to check whether to exceed some threshold τ. Once it exceeds, it records the container. If there is no container satisfying the requirement, it shows that the chunk is unique. Otherwise, we compare the coming chunk with all the chunks in the similar containers to find the most similar chunk. If the similar chunk satisfies the threshold σ, the coming chunk has a similar one. Meanwhile, we add the chunk to the container. Otherwise, it has no similar chunk. For those unique chunks, each of them is a container.

The index time of sketch depends on the number of containers, which are controlled by the threshold τ. If the value of τ is small, it decreases the number of containers but increases the number of comparisons between similar chunks. If τ is high, it increases the number of containers, but decreases the number of comparisons. In addition, σ has an influence on the effect of resemblance detection.

4.4. Chunk compression

By the duplicate and resemblance detection, a coming chunk is evaluated as a unique, duplicate, or similar chunk. As illustrated in Section 3.2, different cases have different compression, including delta and deep compression. Delta compression (e.g. Xdelta (MacDonald, 2000), Zdelta (Xia et al., 2014)) utilises an existing chunk to compress a new chunk and only stores the different part of the latter. Deep compression (e.g. LZW (Nelson, 1989), GZIP (Gailly & Adler, n.d.)) utilises the traditional compression approaches to compress a

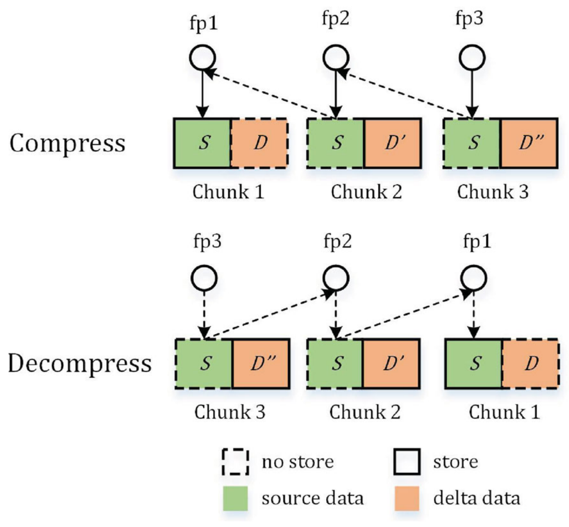

Figure 5. The process of compression and decompression of a similar chunk.

chunk. It is noted that both compression and decompression are lossless, which implies that it does not decrease the quality of data.

4.4.1. Compression and decompression

In FIDCS-ERD, the system adopts a full compression. Before illustrating the compression, we first define two classes of chunks as follows:

- **Source Chunk (SC).** It is a chunk that has been stored in the storage system by delta or deep compression. For a new chunk, source chunk is a duplicate or a similar chunk.
- **Target Chunk (TC).** It is a new chunk to be stored by delta or deep compression. It needs to find its source chunk by duplicate or resemblance detection. When a TC is unique, it has no SC.

We use a ternary table $\mathcal{T} = \{(TC, SC, D)\}$ to record the relationship between TC and its SC and the file name of the difference of TC in the disk. If SC is null, it implies that it is a chunk directly stored in the disk. If SC is not null, it means that TC has a similar chunk and is stored by delta compression.

For the compression process of a TC, it is divided to three different cases. When it is duplicate, it needs no compression. When it is a unique chunk, it is deeply compressed. When it has a similar chunk, it is relatively complex to be compressed. Figure 5 shows a simple example, in which Chunk 2 is a SC and Chunk 3 is a TC. For a TC, it first needs to find its SC, whose difference may be stored. If so, Chunk 2 needs to first find its SC, i.e. Chunk 1 and then get data of SC by decompression. After that, Chunk 3 is executed by delta compression. Meanwhile, $(fp3, fp2, D'')$ is inserted to table $\mathcal{T}$.

For the decompression process of a TC, it is executed by the relationship in table $\mathcal{T}$. Here, we take the decompression process in Figure 5 as a simple example, in which it wants to get data of Chunk 3. At first, it finds record $(fp3, fp2, D'')$ and knows that Chunk 2 is its SC. Then, it continues to get $(fp2, fp1, D')$ and knows that Chunk 1 is a SC of Chunk 2. Next, Chunk 1 is a unique chunk accord to record $(fp1, Null, Null)$. Finally, it gets data of Chunk 3 by a reverse process.

4.4.2. Delta and deep compression

Delta compression is used to compress two similar files or chunks. For a target chunk, its source chunk is a base, which is used as a reference of delta compression. The data after compression is the difference between chunks. In FIDCS-ERD, it implements delta encoding based on Xdelta (MacDonald, 2000), which is one of the optimal approaches to compress highly similar data streams. That is, Xdelta introduces a `Copy/Insert` algorithm, which utilises a string matching technique to first find matching offsets in SC and TC and then generate a set of `Copy` instructions for every matching range and `Insert` instructions to cover the unmatched regions. Xdelta is an approximate implementation of the greedy algorithm based on the hash techniques. It satisfies the linear time and space complexity at the cost of sub-optimal compression.

We use the traditional compression methods to compress the unique chunks. There are a large number of data compression approaches, which use the statistical-model-based coding or the dictionary-model-based coding. They identify the redundancy at the byte level and at the string level, respectively. Wen et al. (2016) illustrated that both the statistical- and dictionary- model-based coding limit the compression window to consider the balance between the time and compression ratio. Since the statistical-model-based coding has the poor scalability, the system selects the dictionary-model-based coding. In the dictionary-model-based coding, there are two classes of the most common approaches. The first one is to improve compression ratio (e.g.DEFLATE and LZMA (Deutsch, 1996; Source, 2020)), while the second one is to reduce the compression time (e.g. LZO Oberhumer, 1997 and LZW Nelson, 1989). In order to get the better compression ratio, the system selects LZMA to compress unique chunks. Meanwhile, for those special files (e.g. video and images Srivastava & Biswas, 2020; Wu, Gao et al., 2019), since they have been executed by the encoding and compression algorithms, using LZMA to compress them has little compression effect. Therefore, our deduplication system will not compress them again.

4.5. Algorithm analysis

In our system, there are two important algorithms, i.e. CBP and SER. The former decides the breakpoints for a data stream, while the latter computes a sketch for some chunk. The time and space complexity to compute a sketch are $O(zk)$ and $O(m)$, in which z is the number of Rabin fingerprints. The key problem is how to chunk the data stream. For different cases, we discuss the time complexity of chunking algorithm CBP as follows:

- $w \geq n$. That is, the size of a data stream is less than that of the sliding window. The data stream is viewed as a chunk and then its fingerprint is directly computed by Equation (1). Meanwhile, there is only one sketch computed, whose time complexity is $O(k)$. Therefore, the time complexity of this case is $O(n + k)$.
- $w < n \leq \beta$. That is, the size of a data stream is greater than that of the sliding window and less than the minimal size of a chunk. Therefore, the data stream is viewed as a chunk

and has only one sketch. For the fingerprint of the first sliding window, its time complexity is $O(w)$ by Equation (1). For the rest fingerprints, each time complexity is $O(1)$ by Equation (2). As a result, the time complexity of all the fingerprints is $O(n)$. For the sketch, the time complexity is $O(zk)$. Therefore, the time complexity is $O(n + zk)$.

- $n > \beta$. The size of a data stream is greater than the maximal size. For fingerprints of the data stream, the time complexity is $O(n)$. For the time complexity of sketches, it is dependent on the number to call Algorithm SER. The maximal number is n/β and the minimal number is n/γ. Therefore, the time complexity is in between $O(n + nzk/\gamma)$ and $O(n + nzk/\beta)$.

From the above analysis, we can find that parameter β and γ have a certain amount of influence on computational complexity and compression ratio when $n \geq \beta$. The important influence is on the computation of sketch. That is, when β is smaller, more sketches may be computed and the size of the chunk may be smaller. Both the time complexity and the compression efficiency may increase. When β is greater, less sketches may be computed and the size of the chunk may be greater. Both the time complexity and the compression efficiency is lower.

5. Experimental evaluation

5.1. Experimental setting

The evaluation is performed on a desktop computer, which has 16GB of RAM, 1TB of SSD and AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx 2.10 GHZ running Windows 10 operating system. All of all algorithms are implemented by `Python` and `C`.

Parameter Setting. There are a lot of parameters that should be considered. Here, we illustrate the values of several important parameters. The average size of a chunk is selected from [4, 8, 16, 32, 64] KB. The value of parameter β and γ is $\alpha - 2$ and $\alpha + 2$, respectively. The length of the sliding window is 48. The number of hash functions for duplicate and resemblance detection is 5. The number of sub-chunks for resemblance detection is 20 and then the length of Bloom filter is 160. The hash algorithm used in our experiments is SHA-256.

Datasets. In our experiments, we select multiple different datasets, including one open source dataset and two private datasets. The datasets are listed as follows:

- *Glibc.* There[1] are 67 files from glibc and glibc-ports, which each is a tar file. Glibc contains the sources of multiple versions of the GNU C library published by a source code version control platform. The total size of the dataset is 10 GB.
- *Email.* The dataset contains private email files of 10 users. The total size of the dataset is 25 GB.
- *Image.* The dataset is about life photos of 6 users. The total size of the dataset is 20 GB.

The detailed description of the datasets is shown in Table 2. These datasets represent three scenes having different levels of duplicate and resemblance. Since each version may have few changes to the previous version, Glibc has a large number of duplicate or similar files. Email contains email files from work of users. There are a certain number of similar files.

Table 2. An overview of datasets.

Dataset	Size	Description
Glibc	10 GB	67 versions of glibc and glibc-ports. Each version is packaged as a tar file.
Email	25 GB	Email files from 10 users.
Image	20 GB	Life photos from 6 users. The format consists of jpg and png.

Image contains a group of image files including JPG, PNG and EPS and has few duplicate and similar files. By these different datasets, we can get the detailed performance of our deduplication system.

Evaluation Metrics. We use multiple metrics to evaluate our proposed system in terms of compression effect and computational efficiency. The concrete metrics are listed as follows:

- *Compression Ratio (*CR*).* The metric is used to evaluate the whole compression effect of the deduplication system and is computed as:

$$\mathrm{CR} = \frac{\text{the size of datasets before compression}}{\text{the size of datasets after compression}}.$$

  CR is the most important metric for any compression algorithm. It is easy to find that CR is the ratio of the size of datasets before compression and after compression. The greater the value of CR, the better the compression effect is.
- **Delta Compression Ratio per Chunk (DCRpC).** The metric is used to evaluate compression effect of resemblance detection algorithms and is computed as

$$\mathrm{DCRpC} = \frac{\text{the removed size per chunk after delta compression}}{\text{the size per chunk before delta compression}}.$$

  It is noted that the metric only considers those chunks that are executed by delta compression. Thus, it does not consist of the duplicated and unique chunks.
- *Throughput (*TP*).* The metric is used to evaluate the computational efficiency of the whole system. It is the volume of processed data streams per second. That is,

$$\mathrm{TP} = \frac{\text{the volume of data streams}}{\text{the processing time}}.$$

Compared Algorithm for Resemblance Detection. Here, we compare our system with the resemblance detection algorithm Finesse in terms of compression performance and computational overhead. Finesse is a representative resemblance detection approach for data deduplication, which was proposed in 2019 (Zhang et al., 2019). Compared to the previous works, it not only guarantees the high resemblance detection effect but also has the lower computational overhead. Therefore, we compare our work with Finesse in multiple metrics.

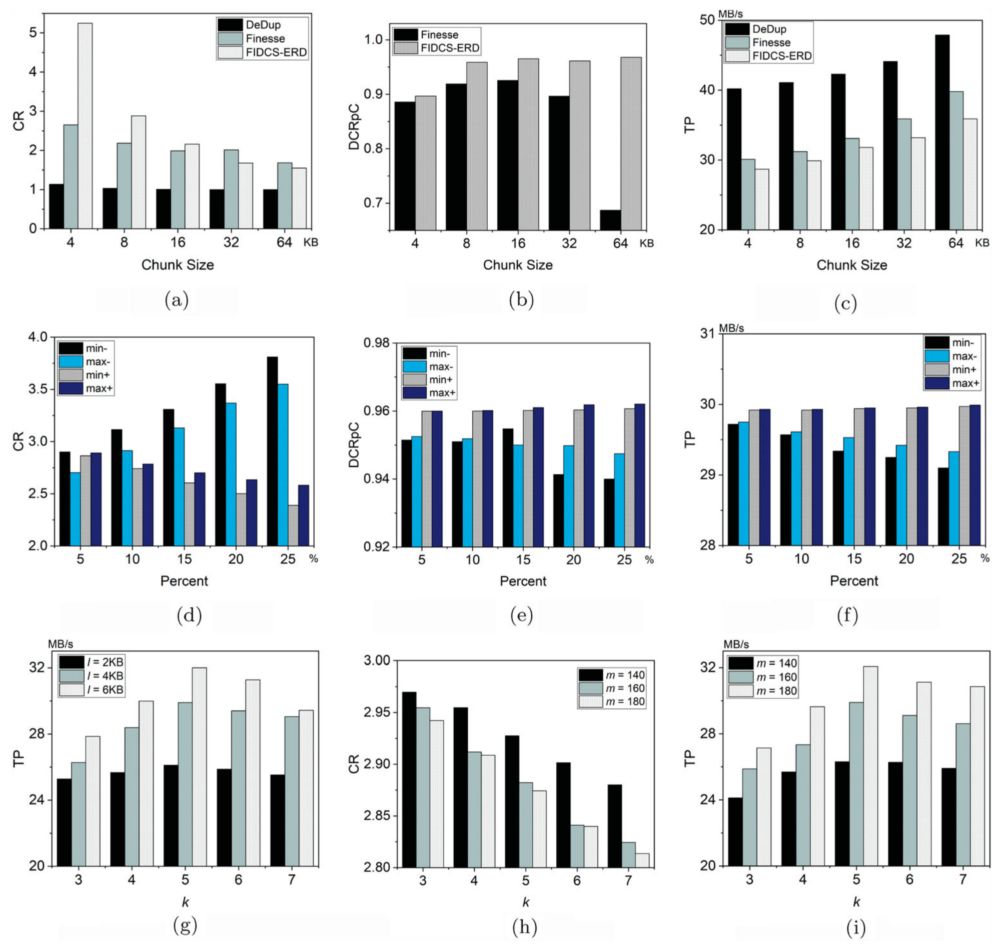

Figure 6. Results of dataset Glibc. (a) CR with α. (b) DCRpC with α. (c) TP with α. (d) CR with β and γ. (e) DCRpC with β and γ. (f) TP with β and γ. (g) CR with *k* for DD. (h) CR with *k* for RD. (i) TP with *k* for RD.

5.2. The impact of parameters

5.2.1. The size of the chunk

There are three important parameters that have an important influence on the size of the chunks, including the average size α, the minimal size β and the maximal size γ. These parameters influence both compression effect and computational efficiency due to different sizes of chunks.

Figure 6(a–c), Figure 7(a–c) and Figure 8(a–c) show the concrete results under different datasets. The average size is set as [4, 8, 16, 32, 64]. Figure 6(a) shows that as α is increasing for 4 KB to 64 KB, the compression ratio of FIDCS-ERD is fast decreasing. The similar results are shown in Figures 7(a) and 8(a) for Email and Image. They demonstrate that the average size has an important influence on compression effect. Meanwhile, Figure 6(b), Figure 7(b) and Figure 8(b) show that the delta compression ratio per chunk does not obviously change as the average size increases. It illustrates that FIDCS-ERD has a good capability to accurately distinguish the similar chunks for different chunks whatever parameter α is. However, the

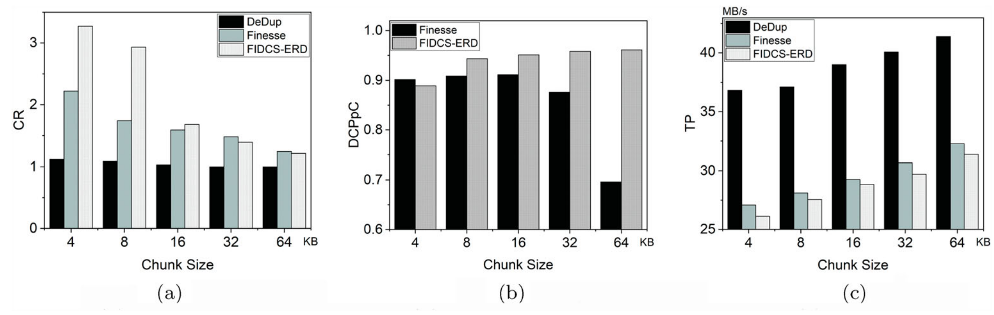

Figure 7. Results of dataset email. (a) CR with α. (b) DCRpC with α. (c) TP with α.

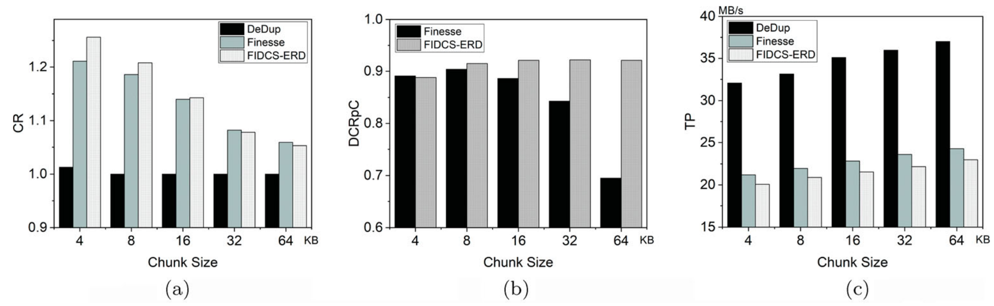

Figure 8. Results of image. (a) CR with α. (b) DCRpC with α. (c) TP with α.

decrease of the average size brings the poor performance since the throughput is reducing. Therefore, the average size also has an important impact on computational efficiency. It is necessary to consider the balance between compression ratio and computation efficiency by setting the reasonable value of parameter α.

It is obvious to find that as the average size of the chunk increases, the compression ratio of duplicate detection is decreasing. When the average size is greater than 32 KB, there are extremely few duplicated chunks. It implies that the increase of the average chunk greatly decreases the compression effect. Combined with the throughput, we suggest that $\alpha = 8$ KB is a relatively optimal selection to balance the computational overhead and compression performance.

Figure 6(d–f) shows the result as parameter γ and β change. In general, the value of β and γ is $\alpha - 2$ and $\alpha + 2$, respectively. In our experiments of Figure 6(d–f), the average size is 8 KB. Thus, the initial value of β and γ is 6 and 10 KB, respectively. In order to analyse the impact of β and γ, we dynamically increase or decrease some percentage on the basis of the initial value of β and γ. For example, "min-, Percent = 5%" means that the value of β is $6 * (1 - 5\%)$ KB, while "max+, Percent = 5%" means that the value of γ is $10 * (1 + 5\%)$ KB. With the decrease of percentage, the compression ratio of "min-" and "max-" is increasing, while that of "min+" and "max+" is increasing. The reason is that the size of chunk in "min-" and "max-" increases, while that in "min+" and "max+" increases. Meanwhile, the delta compression ratio per chunk of "min+" and "max+" is increasing, while that of "min-" and "max-" is decreasing. It illustrates that although the decrease of the size of chunk adds the compression ratio, but reduces the delta compression ratio per chunk. The reason is that the smaller chunk implies the higher probability to find the similar chunks. Meanwhile, the throughput is decreasing when β and γ are decreasing.

5.2.2. Parameters of bloom filter

In our algorithms, Bloom filter is used to find the duplicate and similar chunk. The key parameters are the length of Bloom filter and the number of hash functions. We analyse them by different experiments.

For duplicate detection, the objective is to fast find the same chunk. The parameters of Bloom filter do not influence the compression ratio. Here, we discuss the influence of Bloom filter on the throughput. Figure 6(g) shows the influence of the length of Bloom filter and the number of hash functions, i.e. ℓ and k. When ℓ is constant, $k = 5$ has the optimal throughput. On one hand, the smaller value of k implies the higher error rate, which increases the number of detecting duplicate chunks in the disk. Otherwise, it increases the number of operations by hash functions. Therefore, $k = 5$ is relatively optimal to maximise the computational performance for duplicate detection. On the other hand, the length of Bloom filter also has an important impact on compression ratio and computational efficiency. The greater value of ℓ implies the lower error ratio and higher computation by hash functions. As a result, it reduces the detection in the disk and increases the computation of hash functions. In general, $\ell = 6$ KB is a good selection for duplicate detection.

For resemblance detection, Bloom filter is used to compare the similarity between two chunks. The key parameters (e.g. the length of Bloom filter and the number of hash functions) have an important influence on compression ratio and computational efficiency. Figure 6(h) shows that the greater value of k and m implies the higher requirement of similar detection so as to reduce the compression ratio. On the other hand, the greater value of k implies the more computation overhead, while the greater value of k implies the more resemblance detection. Figure 6(i) shows that $k = 5$ and $m = 180$ are the proper parameters to balance the compression ratio and computational overhead.

5.3. Finesse *vs.* FIDCS-ERD

Figure 6(a–c), Figure 7(a–c) and Figure 8(a–c) show the concrete results of DeDup, Finesse and FIDCS-ERD under different values of the average size. Here, we compare them in terms of compression performance and computational efficiency.

Figures 6(a), 7(a) and 8(a) show the compression ratio for different algorithms. They show that when the average size of the chunk is equal to or less than 16 KB, CR of FIDCS-ERD is greater than that of Finesse. However, when the average size is greater than 16 KB, CR of FIDCS-ERD is less than that of Finesse. The important reason is the influence of the number of sub-chunks for resemblance detection. For all of the average sizes, the parameter is constant. When the average size is small, the length of each sub-chunk is small, which implies that there are a high proportion of bytes to represent the chunk. A solution is to adjust the number of sub-chunks for different average sizes. Meanwhile, it needs to adjust the length of the Bloom filter for each chunk. For image data, Figure 8(a) shows that the optimal CR is greater than 1.25.

Figures 6(b), Figure 7(b) and Figure 8(b) show the result of delta compression of Finesse and FIDCS-ERD for different datasets. It shows that compared to Finesse, FIDCS-ERD removes more redundant data for any chunk by delta compression. When the average size is greater than 16 KB, DCRpC of Finesse is fast decreasing, while that of FIDCS-ERD still has a high value. It implies that resemblance detection in FIDCS-ERD is more accurate than that in Finesse.

We also evaluate computational efficiency of different algorithms under different datasets. Figures 6(c), 7(c) and 8(c) show that DeDup has the best computational efficiency since it needs no resemblance detection. Meanwhile, FIDCS-ERD is just slightly worse than Finesse in terms of throughput. The reason is that FIDCS-ERD fully compares the fingerprints of each chunk to get the optimal similarity.

In summary, compared to Finesse, FIDCS-ERD has two important advantages, including better compression effect and more accurate resemblance detection. Thus, FIDCS-ERD is a good compression technique to provide the strong compression performance.

5.4. Applicable scenarios

The objective of our deduplication system is to find the duplicate and similar chunks and then remove redundant data. In essence, the upper bound of the final compression effect fully depends on the data itself. If there are few duplicate or similar chunks after data streams are chunked, it implies the extremely low compression ratio. It is obvious that data deduplication is not suited to these scenes. In our experiments (e.g. Figures 6–8), we can find that dataset Glibc has the optimal compression effect, while dataset Image has the worse effect. That is, the maximal compression ratio for dataset Glibc is greater than 5, while dataset Image has only 1.25. Meanwhile, since the computational complexity of our deduplication system is relatively high, it is not necessary to execute deduplication operations for data that is neither duplicate nor similar. As a result, our deduplication system can be widely applied to many applications, such as the backup/archive storage system, source code version control, remote synchronisation, storage hardware (Wen et al., 2016). In these applications, the system can get high compression ratio to improve storage efficiency.

6. Conclusion

In this paper, we propose a feature-based intelligent deduplication compression system with extreme resemblance detection. The system consists of two important functions, including duplicate and resemblance detection. The first function removes duplicate chunks, while the second one compresses similar chunks. We propose a content-defined chunking algorithm to solve the boundary-shift problem, Min-BP and Max-BP. We also design a Bloom filter-based resemblance detection algorithm to fast find more similar chunks. Experimental results show good compression effect and low computational overhead for various formats of data. We plan to solve the index problem for resemblance detection to further improve the detection speed in the future. Meanwhile, we also consider some parallel techniques (e.g. multithreading technology) to improve the processing speed and one control node and multiple storage nodes to increase the space.

Note

1. http://ftp.gnu.org/gnu/glibc/.

Acknowledgments

The authors extend their appreciation to the Deanship of Scientific Research at King Saud University for funding this work through research group no. RG-1441-331. This work was supported in part by the National Natural Science Foundation of China under Grant No. 41971343, No. 61872422, No. 41771411.

Disclosure statement

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

The authors extend their appreciation to the Deanship of Scientific Research at King Saud University for funding this work through research group no. RG-1441-331. This work was supported in part by the National Natural Science Foundation of China [grant numbers 41971343, 61872422, 41771411].

ORCID

Xiaotong Wu http://orcid.org/0000-0003-3262-9420

References

Agarwal, B., Akella, A., Anand, A., Balachandran, A., Chitnis, P., Muthukrishnan, C., Ramjee, R., & Varghese, G. (2010). EndRE: An end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX symposium on networked systems design and implementation, NSDI* (pp. 419–432). USENIX Association.

Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., & Klein, S. T. (2009). The design of a similarity based deduplication system. In *Proceedings of SYSTOR* (p. 6).

Aronovich, L., Asher, R., Harnik, D., Hirsch, M., Klein, S. T., & Toaff, Y. (2016). Similarity based deduplication with small data chunks. *Discrete Applied Mathematics*, *212*, 10–22. https://doi.org/10.1016/j.dam.2015.09.018

Bloembergen, D., Hennes, D., McBurney, P., & Tuyls, K. (2015). Trading in markets with noisy information: An evolutionary analysis. *Connection Science*, *27*(3), 253–268. https://doi.org/10.1080/09540091.2015.1039492

Bobbarjung, D. R., Jagannathan, S., & Dubnicki, C. (2006). Improving duplicate elimination in storage systems. *ACM Transactions on Storage*, *2*(4), 424–448. https://doi.org/10.1145/1210596.1210599

Broder, A. Z., & Mitzenmacher, M. (2003). Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, *1*(4), 485–509. https://doi.org/10.1080/15427951.2004.10129096

Cao, Z., Wen, H., Ge, X., Ma, J., Diehl, J., & Du, D. H. C. (2019). TDDFS: A tier-Aware data deduplication-Based file system. *ACM Transactions on Storage*, *15*(1), 4:1–4:26. https://doi.org/10.1145/3295461

Deutsch, L. P. (1996). Deflate compressed data format specification version 1.3. http://tools.ietf.org/html/rfc1951.

Douglis, F., & Iyengar, A. (2003). Application-specific delta-encoding via resemblance detection. In *Proceedings of USENIX annual technical conference* (pp. 113–126).

Drago, I., Mellia, M., Munafò, M. M., Sperotto, A., Sadre, R., & Pras, A. (2012). Inside dropbox: Understanding personal cloud storage services. In *Proceedings of the 12th ACM SIGCOMM internet measurement conference, IMC* (pp. 481–494). ACM.

El-Shimi, A., Kalach, R., Kumar, A., Ottean, A., Li, J., & Sengupta, S. (2012). Primary data deduplication – large scale study and system design. In *2012 USENIX annual technical conference* (pp. 285–296).

Fang, S., Smith, G., & Tabor, W. (2018). The importance of situation-specific encodings: Analysis of a simple connectionist model of letter transposition effects. *Connection Science*, *30*(2), 135–159. https://doi.org/10.1080/09540091.2016.1272097

Fu, Y., Xiao, N., Jiang, H., Hu, G., & Chen, W. (2019). Application-Aware big data deduplication in cloud environment. *IEEE Transactions on Cloud Computing*, *7*(4), 921–934. https://doi.org/10.1109/TCC.6245519

Gailly, J., & Adler, M. (n.d.). The gzip compressor. http://www.gzip.org/.

Gawrychowski, P. (2013). Optimal pattern matching in LZW compressed strings. *ACM Transactions on Algorithms*, *9*(3), 25:1–25:17. https://doi.org/10.1145/2483699.2483705

Harnik, D., Khaitzin, E., Sotnikov, D., & Taharlev, S. (2014). A fast implementation of deflate. In *Data compression conference, DCC* (pp. 223–232). IEEE.

Hosam, O., & Ahmad, M. H. (2019). Hybrid design for cloud data security using combination of AES, ECC and LSB steganography. *International Journal of Computational Science and Engineering*, *19*(2), 153–161. https://doi.org/10.1504/IJCSE.2019.100236

Kane, J., & Yang, Q. (2012). Compression speed enhancements to LZO for multi-core systems. In *IEEE 24th international symposium on computer architecture and high performance computing, SBAC-PAD* (pp. 108–115).

Liu, J., Chai, Y., Qin, X., & Liu, Y. (2018). Endurable SSD-based read cache for improving the performance of selective restore from deduplication systems. *Journal of Computer Science and Technology*, *33*(1), 58–78. https://doi.org/10.1007/s11390-018-1808-5

Liu, Z., Japkowicz, N., Wang, R., & Tang, D. (2019). Adaptive learning on mobile network traffic data. *Connection Science*, *31*(2), 185–214. https://doi.org/10.1080/09540091.2018.1512557

Lu, G., Jin, Y., & Du, D. H. C. (2010). Frequency based chunking for data de-duplication. In *Proceedings of 18th annual IEEE/ACM international symposium on modeling, analysis and simulation of computer and telecommunication systems, MASCOTS* (pp. 287–296). IEEE Computer Society.

Luo, L., Guo, D., Ma, R. T. B., Rottenstreich, O., & Luo, X. (2019). Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, *21*(2), 1912–1949. https://doi.org/10.1109/COMST.9739

MacDonald, J. (2000). File system support for delta compression [Master' s thesis]. Department of Electrical Engineering and Computer Science, University of California at Berkeley.

Mao, B., Jiang, H., Wu, S., Fu, Y., & Tian, L. (2012). SAR: SSD assisted restore optimization for deduplication-based storage systems in the cloud. In *Seventh IEEE international conference on networking, architecture, and storage, NAS* (pp. 328–337). IEEE Computer Society.

Mao, B., Yang, Y., Wu, S., Jiang, H., & Li, K. (2019). IOFollow: Improving the performance of VM live storage migration with IO following in the cloud. *Future Generation Computer Systems*, *91*, 167–176. https://doi.org/10.1016/j.future.2018.08.036

Muthitacharoen, A., Chen, B., & Mazières, D. (2001). A low-bandwidth network file system. In *Proceedings of the 18th ACM symposium on operating system principles, SOSP* (pp. 174–187).

Nayak, S. K., Rout, P. K., Jagadev, A. K., & Swarnkar, T. (2018). Elitism-based multi-objective differential evolution with extreme learning machine for feature selection: A novel searching technique. *Connection Science*, *30*(4), 362–387. https://doi.org/10.1080/09540091.2018.1487384

Nelson, M. R. (1989). LZW data compression. *Dr. Dobb's Journal*, *14*(10), 29–36.

Oberhumer, M. (1997). Lzo real-time data compression library. http://www.infosys.tuwien.ac.at/Staff/lux/marco/lzo.html.

Organization, L. K. (2020). The linux kernel archives. https://www.kernel.org/.

Rabin, M. O. (1981). Fingerprinting by random polynomials. Center for Research in Computing Techn., Aiken Computation Laboratory.

Shilane, P., Huang, M., Wallace, G., & Hsu, W. (2012). WAN optimized replication of backup datasets using stream-informed delta compression. In *Proceedings of the 10th USENIX conference on file and storage technologies, FAST* (p. 5).

Source, O. (2020). 7zip. http://www.7-zip.org/.

Srivastava, V., & Biswas, B. (2020). CNN-based salient features in HSI image semantic target prediction. *Connection Science*, *32*(2), 113–131. https://doi.org/10.1080/09540091.2019.1650330

Tan, Y., Jiang, H., Feng, D., Tian, L., & Yan, Z. (2011). CABdedupe: A causality-based deduplication performance booster for cloud backup services. In *25th IEEE international symposium on parallel and distributed processing, IPDPS* (pp. 1266–1277). IEEE.

Trendafilov, D., Memon, N., & Suel, T. (2002). Zdelta: *An efficient delta compression tool* [Tech. Rep.]. Dept. Comput. Inf. Sci., Polytechnic Univ.

Wang, C., Wei, Q., Yang, J., Chen, C., Yang, Y., & Xue, M. (2018). NV-Dedup: High-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers*, *67*(5), 658–671. https://doi.org/10.1109/TC.2017.2774270

Wang, X., Yang, L. T., Song, L., Wang, H., Ren, L., & Deen, J. (2020). A tensor-based multi-attributes visual feature recognition method for industrial intelligence. *IEEE Transactions on Industrial Informatics*, 17(3): 2231-2241. https://doi.org/10.1109/TII.2020.2999901

Wang, X., Yang, L. T., Wang, Y., Liu, X., Zhang, Q., & Deen, M. J. (2019). A distributed tensor-Train decomposition method for cyber-Physical-Social services. *TCPS*, *3*(4), 35:1–35:15. https://doi.org/10.1145/3323926

Wang, X., Yang, L. T., Wang, Y., Ren, L., & Deen, M. J. (2021). ADTT: A highly-efficient distributed tensor-train decomposition method for IIoT big data. *IEEE Transactions on Industrial Informatics*, 17(3): 1573-1582. https://doi.org/10.1109/TII.2020.2967768

Wen, X., Hong, J., Dan, F., Douglis, F., Shilane, P., Yu, H., & Zhou, Y. (2016). A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, *104*(9), 1681–1710. https://doi.org/10.1109/JPROC.2016.2571298

Wu, Z., Gao, Y., Li, L., Xue, J., & Li, Y. (2019). Semantic segmentation of high-resolution remote sensing images using fully convolutional network with adaptive threshold. *Connection Science*, *31*(2), 169–184. https://doi.org/10.1080/09540091.2018.1510902

Wu, S., Li, H., Mao, B., Chen, X., & Li, K. (2019). Overcome the GC-Induced performance variability in SSD-based RAIDs with request redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *38*(5), 822–833. https://doi.org/10.1109/TCAD.43

Xia, W., Jiang, H., Feng, D., & Tian, L. (2016). DARE: A deduplication-Aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Transactions on Computers*, *65*(6), 1692–1705. https://doi.org/10.1109/TC.2015.2456015

Xia, W., Jiang, H., Feng, D., Tian, L., Fu, M., & Zhou, Y. (2014). Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, *79*, 258–272. https://doi.org/10.1016/j.peva.2014.07.016

Xu, X., Liu, X., Xu, Z., Dai, F., Zhang, X., & Qi, L. (2020). Trust-Oriented IoT service placement for smart cities in edge computing. *IEEE Internet of Things Journal*, *7*(5), 4084–4091. https://doi.org/10.1109/JIoT.6488907

Xu, X., Mo, R., Dai, F., Lin, W., Wan, S., & Dou, W. (2020). Dynamic resource provisioning with fault tolerance for data-Intensive meteorological workflows in cloud. *IEEE Transactions on Industrial Informatics*, *16*(9), 6172–6181. https://doi.org/10.1109/TII.9424

Xu, X., Shen, B., Yin, X., Khosravi, M., Wu, H., Qi, L., & Wan, S. (2020). Edge server quantification and placement for offloading social media services in industrial cognitive IoV. *IEEE Transactions on Industrial Informatics*, 1. https://doi.org/10.1109/TII.2020.2987994

Xu, X., Zhang, X., Liu, X., Jiang, J., Qi, L., & Bhuiyan, M. Z. A. (2020). Adaptive computation offloading with edge for 5G-Envisioned internet of connected vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 1–10. https://doi.org/10.1109/TITS.2020.2982186

Yu, C., Zhang, C., Mao, Y., & Li, F. (2015). Leap-based Content Defined Chunking – Theory and Implementation. In *IEEE 31st symposium on mass storage systems and technologies, MSST* (pp. 1–12). IEEE Computer Society.

Zeng, S. (2020). VFS_CS: A light-weight and extensible virtual file system middleware for cloud storage system. *International Journal of Computational Science and Engineering*, *21*(4), 513–521. https://doi.org/10.1504/IJCSE.2020.106865

Zhang, Y., Feng, D., Hua, Y., Hu, Y., Xia, W., Fu, M., & Zhou, Y. (2017). Reducing chunk fragmentation for in-line delta compressed and deduplicated backup systems. In *International conference on networking, architecture, and storage, NAS* (pp. 1–10).

Zhang, Y., Jiang, H., Feng, D., Xia, W., Fu, M., Huang, F., & Zhou, Y. (2015). AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE conference on computer communications, INFOCOM* (pp. 1337–1345). IEEE.

Zhang, Y., Xia, W., Feng, D., Jiang, H., Hua, Y., & Wang, Q. (2019). Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *17th USENIX conference on file and storage technologies, FAST* (pp. 121–128).

Zhou, B., & Wen, J. (2013). Hysteresis re-chunking based metadata harnessing deduplication of disk images. In *Proceedings of the 42nd international conference on parallel processing, ICPP* (pp. 389–398). IEEE Computer Society.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory/Professional Technical Group on Information Theory*, *23*(3), 337–343. https://doi.org/10.1109/TIT.1977.1055714

Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory/Professional Technical Group on Information Theory*, *24*(5), 530–536. https://doi.org/10.1109/TIT.1978.1055934