Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Analysis and Comparison of Deduplication Strategies in IPFS

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Marcel Gregoriadis

Gutachter/innen: Prof. Dr. Florian Tschorsch
Prof. Dr. Matthias Weidlich

eingereicht am: 3.11.2022 verteidigt am: 25.1.2023

Acknowledgements

I would like to express my gratitude to my primary supervisor, Leonhard Balduf, who guided and supported me throughout this project, as well as to Dr. Martin Florian, who likewise inspired this work with his ideas.

Furthermore, I want to thank Prof. Dr. Florian Tschorsch and Prof. Dr. Matthias Weidlich for agreeing to examine this thesis.

I also wish to acknowledge my very good friends, Hai Dang Nguyen and Anna Tverdovska, who supported this thesis by providing authentic data for its analyses.

Finally, I want to thank my parents who always supported me and furthermore paved the way for me to get to this point. *Danke für alles! Ich liebe euch.*

Contents

Abstract

IPFS has recently risen in popularity, as it represents the backbone for file sharing in a decentralized web. As the amount of files exchanged on IPFS grows, and both storage and network bandwidth are expensive, the discussion around deduplication strategies becomes pressing. This discussion is largely founded on the execution of chunking algorithms.
To this end, we analyzed and compared FastCDC and AE, as two state-of-the-art chunking algorithms, with Rabin, Buzhash, and fixed-size chunking, which are prevalent in implementations of IPFS. Our analysis includes the evaluation of computational efficiency, deduplication ratio on various datasets, average chunk size and chunk-size distribution. We reproduce results from previous literature, and further believe to be the first to present comprehensive results for Buzhash, as well as for certain metrics of the other chunking algorithms. Furthermore, we conducted an analysis on an empirical dataset based on traces from IPFS, which reveals further insights about the data landscape and user behavior in IPFS. Finally, we propose an improved chunking strategy for IPFS.

1. Introduction

With the current rise of decentralized technologies such as the cryptocurrencies, NFTs, and the Metaverse, the InterPlanetary File System (IPFS) presents itself as the backbone of a new generation of the internet, famous by the names of Web3 and Web 3.0. The solution it provides is a globally distributed and decentralized network which enables peers to store and exchange files. This enables better data resiliency and overall democratizes data by protecting it against censorships. As such, IPFS has widely been used to store the asset files behind NFTs, cross-referenced in the blockchains of various cryptocurrencies [1, 2]. Furthermore, data on IPFS is addressed by its content rather than by its location (as is the case with HTTP). IPFS then serves the content from a node that holds that data.

Protocol Labs (the company behind IPFS) currently claims over 2 million weekly unique users and around 125 TB of traffic each week on IPFS [3]. In a vision of an internet that aims to be more efficient than traditional HTTP, detecting redundancies is at IPFS' core. This goes for both, deduplication on the large scale to use available storage more effectively, and on user-individual operations with the network. Once requested, files can remain cached on the user's device and will not have to be retrieved again when accessing contents which *include* the file. This has a positive effect on the user experience since the interactions with the network become more efficient, but it also reduces the overall load on the network. To this end, the data does not even have to be an actual, whole file. In fact, much better results are achieved by splitting files into blocks of data, i.e., chunks [4, 5, 6, 7]. A fine granularity increases the chance and the amount of duplicated blocks of data that can be detected between files.

Finding the right granularity and determining the right chunk boundaries is a balance act between time and space complexity of the operation, and deduplication performance. We explain the measure of deduplication performance more technically in our analysis in Section 4. In a nutshell, it refers to how little data can be used effectively to represent all the data (e.g., of a file). Another aspect to consider in a distributed setting like IPFS are the network round-trip times that of course are expected to be at greater expense with smaller chunk sizes. Many algorithms exist that attempt to achieve the best results in consideration of these parameters, and finding those still remains an active topic of research [8, 9, 10].

IPFS by default splits files into chunks of 256 KiB. This approach is called fixed-size chunking (FSC, cf. Section 2.1.2) and is very efficient but lacks the advantages in deduplication performance that could have been achieved with variable-size chunking. The category of variable-size chunking algorithms sets the chunking boundaries on the basis of various parameters that can shift when the underlying content is altered. Hence, it is also called content-defined chunking (CDC). The rationale behind this procedure is for a modified version of a file to only affect as little chunks as possible. This objective is failed by fixed-size chunking where an extra byte or a removed byte somewhere in the file shifts all the following, still redundant content. With fixed chunking boundaries, the previous boundaries will likely not be in alignment with the content in the way they were before.

However, IPFS does also support CDC through a CLI parameter. In their current implementation, Rabin-based chunking (cf. Section 2.1.3) and Buzhash (cf. Section 2.1.4) are the supported CDC algorithms. Furthermore, it allows setting the target chunk size and other Rabin-specific parameters. As to be expected however, since it is the default option, most objects in IPFS are chunked using FSC.

In this thesis, we explore state-of-the-art chunking algorithms and their effect on data deduplication, specifically in the case of IPFS. To this end, we analyze the features of FSC, Rabin-based chunking, Buzhash, and also two chunking algorithms that came about more recently, FastCDC and Asymmetric Extremum (AE), and evaluate their performance in specific benchmarks. Those are in particular computational efficiency and the deduplication ratio achieved with every algorithm. While the computational efficiency does not depend on the object of data, deduplication performance does. Especially with CDC, where the output is inherently content-dependant, the deduplication ratio is content-dependant as well. For example, two randomly generated files are likely not to share any redundant content. Furthermore, the chance decreases with increasing chunk size. In turn, two consecutive versions of the same textual document are very likely to share a lot of redundant content. Therefore, it is relevant on which datasets the analyses are performed. In order to get a higher resolution understanding of those results, we further conduct an analysis of the produced chunk sizes, i.e., their distribution and average. This is important because, as a result of the content-dependance of CDC, the actual *average* chunk size can vary from the specified *target* chunk size in different degrees.

In Section 4, we evaluate and compare the deduplication performance of the five introduced algorithms based on different prepared datasets and file types. Afterwards, we complete the same analysis but on empirical data that we collected from running nodes in the IPFS network and recording the traces of a large population of nodes (based on [11]). Our empirical analysis includes two factors that the previous analysis does not: (1) It is applied on a dataset that is representative for the kind of data that is actually exchanged on IPFS. (2) It also considers the dataset individual users are interested in. This is interesting because deduplication in file retrieval (and a reduction in traffic) can only then be observed if the file requested by the user shares redundant content with the data already downloaded by the user. We describe our methodology and its results in greater detail in Section 5.

Ultimately, deriving from the results of both analysis, which we discuss in Section 6, we determine a deduplication strategy with regards to chunking that can achieve better results than what is currently implemented in IPFS.

Lastly, we sum up the key results of our analyses, as well as conclusions regarding an improvement to IPFS in Section 7. In addition, we give an outlook on possible next steps that could benefit deduplication in IPFS in Section 8.

2. Preliminaries

This section provides the required preliminaries to follow along the remainder of this thesis. Here, we give an overview of the mechanics of chunking for deduplication, which metrics exist and how they relate to each other. This in particular lays out the background to understand our analyses and how to interpret their results. We further give a detailed explanation on the algorithmic details of each analyzed algorithm. Lastly, we provide a background on IPFS and deduplication in IPFS.

2.1. Chunking Algorithms

Traditionally, data reduction has been the result of compression algorithms such as LZ77 [12] or DEFLATE [13]. Many of these algorithms have a dictionary model based approach. They look for recurring byte sequences and replace them with pointers to the actual value. Despite their effectiveness on individual files, compression algorithms are not suitable for large-scale storage systems for two reasons:

1. Their time and space complexity and byte-level approach can quickly lead to scalability issues with bigger files or bigger collections of files.

2. Due to the way those algorithms work, a slight change in an original file can already create a byte schema in the compressed output that looks very different from the previous version. This makes it difficult (or impossible) to detect redundancies between files that are similar.

The latter becomes a concern when it is common to have multiple versions of a file, as with hard disk backups or as it is the case for data in IPFS.

Therefore, the much more common approach with deduplication in such systems is to rely on chunking algorithms to split files into small blocks of data (e.g., 8 kB), called *chunks*. The size of those blocks determine the granularity on which redundancies could be detected and ultimately tradeoff computing efficiency for deduplication performance. Subsequently, each chunk gets indexed by the computed hash value of their content, also called their *fingerprint*. Typically, SHA-1 is used for this [7, 14, 15].

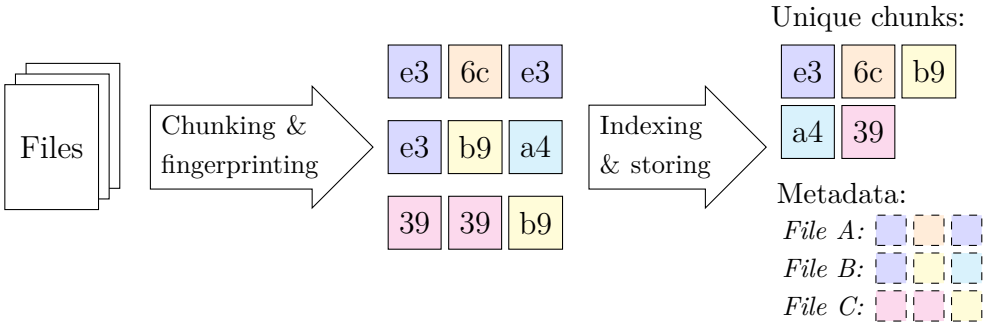


Figure 1: Overview of deduplication processing. Adapted from [16].

On the lowest level, chunking algorithms can be subdivided into FSC and CDC. CDC deals with the challenge of determining the right chunk boundaries inside a file to obtain maximum deduplication. A selection of CDC algorithms is presented in the remainder of this chapter.

2.1.1. Evaluation Criteria

Before presenting and discussing different chunking algorithms, we first want to determine what criteria makes a good chunking algorithm and what goals a chunking algorithm should pursue. To this end, we will have an abstract view on a chunking algorithm and introduce some notation along the way.

Let A be a chunking algorithm and let S be the sequence of bytes that is object to chunking. Furthermore, let $H^A(S)$ be the function that splits S into its chunks $c_1, \ldots, c_n$.

Obviously, we want $H^A(S)$ to perform quick and with no greater than linear complexity in relation to $|S|$, i.e., we can define **computational efficiency** as our first evaluation criterion.

The other goal of a chunking algorithm is to achieve high **deduplication ratios**. Deduplication can be achieved when two or more files share similarities to one another, or even when a single file contains redundant data. To this end, let S' be a slightly locally modified version of S, such that S and S' have shared sequences before and after the modification. For this purpose, let $premax(S, S')$ define the longest shared prefix, and let $postmax(S, S')$ define the longest shared suffix. For better comprehension, Equation 1 shows an example for two strings, the local modification being underlined. It also introduces $shared(S, S')$, which can be described as the size of duplicated content between S and S', and $new(S, S')$, the size of the introduced difference.

$$\begin{aligned} S &= \texttt{this is an example text} \\ S' &= \texttt{this is a sample text} \\ premax(S, S') &= \texttt{this is a} \\ postmax(S, S') &= \texttt{ample text} \\ shared(S, S') &= |premax(S, S')| + |postmax(S, S')| && = 19 \\ new(S, S') &= |S'| - shared(S, S') && = 2 \end{aligned} \tag{1}$$

Ideally, a chunking algorithm is able to achieve a deduplication ratio of $|shared(S, S')|/|S|$, which in the case of the example would result in a ratio of about 78%. This is rarely possible due to chunks and not bytes being the atomic units that determine the size of the shared sub-sequences. Equation 2 shows the same example, but with the sequences divided into chunks of 5 bytes (marked by the surrounding brackets and using fixed-size chunking, cf. Section 2.1.2).

$$\begin{aligned} S &= \texttt{[this][is an][exam][ple t][ext]} \\ S' &= \texttt{[this][is a][sampl][e tex][t]} \end{aligned} \tag{2}$$

As can be seen, this leads to a much larger sum of differing bytes, where only the first 5 bytes could be deduplicated. This comes naturally with the reduced granularity with increased chunk sizes. Moreover, this example demonstrates the *boundary-shift* problem, which causes all consecutive chunks in S' after the local modification to differ from the equivalent chunks in S. This has to do with where the chunking boundary is set and is a challenge for which other, *content-defined* chunking algorithms attempt to find solutions. The following sections will address this topic in greater detail.

While the maximum deduplication ratio could always be achieved by setting a chunk's size to exactly 1 byte, this has other negative implications. For one, the computational efficiency is affected a lot by the chosen chunk size. This is even more the case with algorithms more complex than fixed-size chunking, and considering files of many mega- or even gigabytes, this can quickly become an unacceptable compromise. On the other hand, one also has to consider the cost of the metadata that is necessary to represent a file as a sequence of its chunks. That is to say, deciding on the chunk size is always about finding a balance between computational efficiency and data overhead on the one side, and deduplication ratio on the other side.

After all, the deduplication ratio of an algorithm is not a fixed value. It depends on the applied dataset and only holds true for this exact dataset. However, we can formalize another evaluation criterion which can generally predict the deduplication ratio, and this is the **chunk-size variance**, as disclosed in [17]. In CDC, chunk sizes vary. This is natural, since the chunking algorithm attempts to determine the "right spot" to set the next chunking boundary. Nevertheless, how much chunk sizes vary has a direct implication on the deduplication ratio of any given algorithm.

To prove the relationship between chunk-size variance and deduplication ratio formally, some further notations need to be introduced:

- $\mu^A(S)$ is the average chunk size in $H^A(S)$.
- μ^A is $\mu^A(S)$ over random large S.
- $\sigma^A(S)$ is the standard deviation of chunk sizes in $H^A(S)$
- σ^A is $\sigma^A(S)$ over random large S.
- $\Delta^A(S, S')$ is the total size of all chunks that occur in $H^A(S')$ and not in $H^A(S)$.
- $V^A(S)$ is the *expected* value of $\Delta^A(S, S') - new(S, S')$.

In FSC, $V^A(S) \approx |S|/2$, because a modification (which on average will occur in the center of S) usually affects every consecutive chunk until the end of S, as could be observed in Equation 2. Therefore, here $V^A(S)$ depends on the length of S. This is not the case for CDC algorithms, such that we can denote V^A as the average expected value. The goal of any CDC algorithm A of course is to get $V^A(S)$ to be as low as possible. Let us contemplate again the case of a S that has undergone a slight modification. We define a "slight modification" such that $new(S, S') < \mu^A(S)$. Let $\Phi^A(S)$ be the expected size of the first affected chunk. Statistically, and also intuitively, this is likely

to be a large chunk. We can also derive this value from the previously introduced variables, see Equation 3.

$$\Phi^A(S) = \mu^A(S) + \frac{(\sigma^A(S))^2}{\mu^A(S)} \tag{3}$$

It is very unlikely, that this will be the only affected chunk, since the chunk boundary is very likely to shift as well and this can even trigger a chain reaction. Generally, CDC algorithms attempt to control this effect and keep the total number of chunks affected by a small modification low. Let us denote this number as K^A. Finally, Equation 4 derives the expected total size of chunks affected by a slight modification. The authors of [17] refer to it as the **modification overhead** of A.

$$V^A \approx \Phi^A + (K^A - 1) \cdot \mu^A \tag{4}$$

This goes to show us that in fact a higher chunk-size variance, or more specifically the expected size of the first affected chunk Φ^A, generally have a negative effect on the deduplication ratio.

However, algorithms with very low chunk-size variances σ^A effectively turn into FSC, where $\sigma^A = 0$. Recall how also just decreasing the denominator of the fraction in Equation 3, i.e., the average chunk size $\mu^A(S)$, will add to the cost of metadata storage, but also the required update on the metadata after any slight modification. We refer to the expected number of bytes affected by a change in the file as the **index overhead**, see Equation 5.

$$\alpha^A = \frac{V^A}{\mu^A} \tag{5}$$

Finally, the goal is to optimize the benefits of reducing the chunk-size variance and at the same time keeping the number of chunks affected by a local modification low. CDC algorithms, as presented in the subsequent sections, have different approaches to achieve that.

Besides the chunk-size variance, there is also a need for a discussion about the optimal (average) chunk size μ. It is obvious how splitting the file into too large chunks has a negative effect in deduplication, because redundancies can only be spotted on a coarse granularity. In other words, any small change will affect at least μ bytes and more likely a multiple of μ. I.e., it will generally result in a larger *modification overhead*.

Going to the other extreme, smaller chunks will produce more duplicated chunks. Furthermore, a small modification will also affect fewer bytes the smaller the size of the chunks. So, within limits, smaller chunk sizes will return better deduplication ratios. However, it comes with cost in the time and space complexity. Smaller chunks will obviously result in more chunks, which means more indexes to store and maintain, i.e., more index overhead. Moreover, the performance of the process of splitting the file into its chunks will suffer greatly. With files in the range of hundreds of megabytes, this can have a very significant effect on the user experience.

As concluded in [7], a well-designed system should choose the smallest (average) chunk size given the throughput and capacity requirements for the product. In all chunking algorithms that we are aware of, the average chunk size is a parameter that can be set arbitrarily (disregarding some technical limitations at the extremes).

2.1.2. Fixed-Size Chunking

Fixed-Size Chunking (FSC) splits a file f into chunks of a fixed/predefined size s. For the generated chunks $C = (c_1, \ldots, c_n)$, it can be said that $\forall i < n : |c_i| = s$ with the last chunk $|c_n| \leq s$.

This is the most primitive approach of file chunking and at the same time the most efficient in computation. However, if only one byte is added or removed in a chunk c_i, the byte shift will also affect all consecutive chunks. This can easily lead to a small intersection (i.e., poor deduplication) between C and the updated result C'. Unless there exists a correctly aligned sequence of homogenous bytes of length $s + 1$, it can be observed that $|C \cap C'| = i - 1$.

Finally, the chunking of two files that differ only in the first few bytes can lead to result sets with no or very little similar chunks. In the same way, a large sequence of bytes entirely contained in another file, e.g., an MP4 file that is also part of a TAR archive, will likely not receive any benefits of deduplication. This is known as the *boundary-shift* problem [18, 8].

This undesirable circumstance is attempted to be defeated by CDC algorithms. A selection of those is presented in the following sections.

2.1.3. Rabin

The Rabin fingerprinting schema [19, 20] is an algorithm based on a hashing function that represents a polynomial over a finite field and a modulo operation with an irreducible polynomial. A popular application of this hashing function is the Rabin-Karp algorithm [21] which is a method to efficiently search for occurrences of a substring in a string. However, it is also interesting as an approach to CDC that has been applied in order to eliminate redundancies in storage systems [22, 23] and network traffic [18, 24, 25].

In the Rabin fingerprinting algorithm, a message $m = (m_1, \ldots, m_n)$ of n bytes is transformed to a polynomial of degree $n - 1$ according to the following schema:

$$H(m_1, \ldots, m_n) = m_1 x^{n-1} + m_2 x^{n-2} + \ldots + m_n \quad \text{with} \quad x \in \mathbb{Z}_2 \tag{6}$$

The variable x is commonly a prime. Further, a polynomial P of degree $n - 1$, irreducible over $\mathbb{Z}_2$ is randomly chosen. Finally, the Rabin fingerprint of m is defined as $H(m) \mod P$. It should be noted that while in this explanation m was divided into fragments of 1 byte, this is a parameter that can also be set to any other number of bits or bytes. For convenience, the unit of 1 byte will continue to be used in the remainder.

The hashing algorithm for Rabin fingerprints is the first prominent example in the category of *rolling hash* algorithms. Rolling hashes are computed iteratively on a sliding

window of w bytes over a file using a shift of typically 1 byte, deliberately overlapping $w-1$ bytes of the previous window. This process is pictured in Figure 2.

The hash function H, used to compute the Rabin fingerprint, is classified as a rolling hash function because the hash of the preceding window can be utilized to compute the hash of the next window, respectively. Let Equation 6 be the computation of the first window ($n = w$), the computation in the next iteration can be simplified as:

$$\begin{aligned} H(m_2, \ldots, m_{n+1}) &= m_2 x^{n-1} + m_3 x^{n-2} + \ldots + m_{n+1} \\ &= (H_{prev} - m_1 x^{n-1})x + m_{n+1} \end{aligned} \tag{7}$$

Used with a sliding window over a file, this property ultimately makes H an operation in $\mathcal{O}(1)$ as opposed to $\mathcal{O}(w)$.

In the context of chunking, this schema is applied to determine chunk boundaries. Let $b \in \mathbb{N}$ be a number of bits. The end of each sliding window whose fingerprint b least-significant bits equal zero mark a new chunk boundary. This can then be expected to occur for every window with a probability of 2^{-b}.

Example: A file of 1MB, i.e., a length $l = 10^6$, is chunked using the Rabin fingerprinting schema. Let the sliding window be 256 kB, i.e., $w = 256 \cdot 10^3$, and $b = 17$. This results in $x = l - w + 1 = 10^6 - 256 \cdot 10^3 + 1$ executions which are expected to result in $2^{-b}x \approx 6$ chunks which would then have an average size of ≈ 167 kB each.

The example illustrates how the parameters, primarily the choice of b, can be used to regulate the desired average size (target chunk size) of a chunk. More precisely, a target chunk size of μ can be attained with $b = log_2(\mu)$.

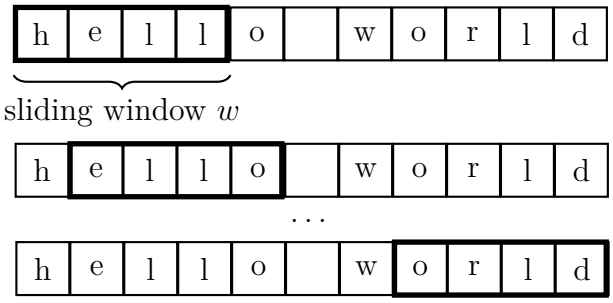


Figure 2: Rolling hash mechanism.

2.1.4. Buzhash

Buzhash is another algorithm based on polynomials that can be used for CDC. Its algorithm was proposed in the applications of text searching algorithms [26, 27].

An implementation of Buzhash requires a hash table that maps bytes to some integer value. Those values must be chosen randomly, but distinct. The table lookup can further be denoted as a hash function $h : [0, 255] \to [0, 2^{32}]$. Let $\rho^b(x)$ be a function that performs a binary rotation on x, i.e., all bits are shifted by b bit to the left and the first bits becoming the last ones.

Finally, the Buzhash value of a message $m = (m_1, \ldots, m_n)$ of n bytes can be computed through the following function:

$$H(m_1, \ldots, m_n) = \rho^{n-1}(h(m_1)) \oplus \rho^{n-2}(h(m_2)) \oplus \ldots \oplus h(m_n) \tag{8}$$

Further iterations of the rolling hash can again be computed utilizing the previous hash in the following fashion:

$$H(m_2, \ldots, m_{n+1}) = \rho(H_{prev}) \oplus \rho^n(h(m_1)) \oplus h(m_n + 1) \tag{9}$$

While the principle is very similar to that of Rabin fingerprints, Buzhash relies on bit shifts instead of multiplications, making it superior in computing efficiency.

2.1.5. FastCDC

FastCDC [9] aims at maximizing speed while intending to achieve similar deduplication ratios than other state-of-the-art CDC algorithms. Its ideas build on top of Gear-based chunking, first employed in Ddelta [28]. Gear-based chunking already is a speed-optimized approach to previous CDC algorithms. Similar to Buzhash, it employs a hash function h that maps every byte to a randomly predefined integer. The fingerprint of a sequence of n bytes is then calculated as:

$$H(m_1, \ldots, m_n) = h(m_1) \cdot 2^{n-1} + h(m_2) \cdot 2^{n-2} + \ldots + h(m_n) \tag{10}$$

Applied on a sliding window (here, n bytes), consecutive fingerprints can be computed in the manner of a rolling hash using the following formula:

$$H(m_2, \ldots, m_{n+1}) = (H_{prev} \ll 1) + h(m_{n+1}) \mod 2^n \tag{11}$$

Using only a bit shift with a modulo operation to remove the previously first byte of the operation and appending the newly added byte merely by an addition and a table lookup, rolling hashes can be computed with extreme efficiency. According to the authors of [28], this removes more than half of the computational overhead from Rabin-based chunking.

Subsequently and similar to the Rabin-based approach, the x *most*-significant bits are compared to a predefined value r in order to determine the next chunk boundary. This comparison can also be expressed as an operation $H \,\&\, \text{Mask} == r$ with a mask comprised of x one bits.

FastCDC essentially provides 3 modifications to this algorithm in order to improve for speed even further.

1. It pads the mask with additional zero bits, effectively enlarging the sliding window. This has the effect of less chunking position collisions (i.e., reduced randomness when determining positions as chunk cut-points) and therefore a better deduplication ratio. Furthermore, the hash judgment accounting for more than 60% of the CPU overhead in Gear [10] was optimized. Instead of

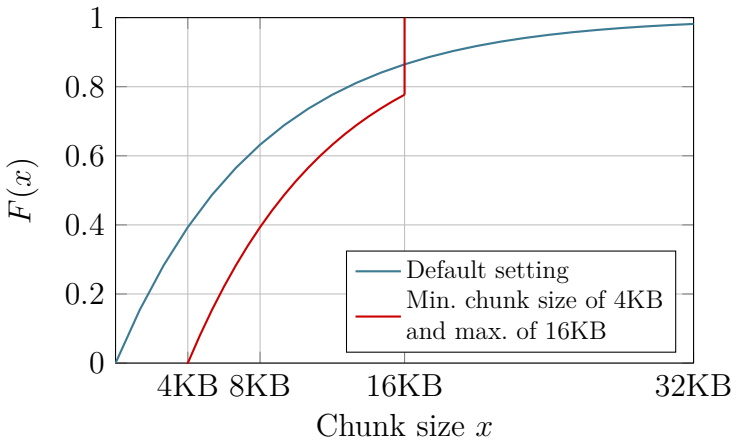


Figure 3: Cumulative distribution of chunk sizes in Rabin-based approaches with an expected chunk size of 8 kB.

H & Mask == r, FastCDC applies H & Mask == 0 which next is simplified to !H & Mask. This reduces the register space for storing r but also cuts off the unncessary comparison operation.

2. Rabin-based algorithms (including Gear) produce chunk sizes following an exponential distribution (see Figure 3). To avert pathological chunk sizes, it is common in CDC to determine minimum and maximum chunk sizes [24, 25, 8]. This aids to decrease chunk-size variance (cf. Section 2.1.1). FastCDC's intent for setting a minimum chunk size, here again, is to boost speed by skipping the hash computations for those beginning bytes (after each new cut-point). However, setting such boundaries inevitably decreases deduplication ratio. The reason is that chunk boundaries become position-dependant rather than really content-based [25]. In its extreme setting (i.e., the minimum and maximum chunk sizes lie very close to the desired average chunk size), the algorithm effectively turns into that of FSC (cf. Section 2.1.2). FastCDC counteracts the effect by its third key idea.

3. FastCDC normalizes the chunk-size distribution by implementing a new rule into the algorithm. While the current index is below the average chunk size, it applies a mask with *more* effective bits in the hash judgment. When it exceeds the average chunk size, the number of effective bits in the mask is decreased. As a consequence, the chunk-size distribution normalizes around the desired chunk size, making pathological chunk size asymptotically unlikely, as depicted in Figure 4. This finally allows FastCDC to skip cut-points of subminimum chunks without sacrificing deduplication ratio.

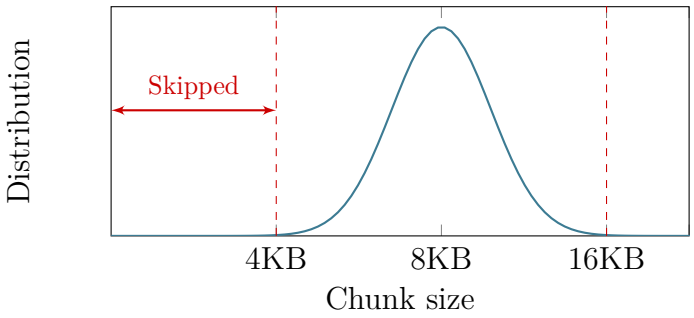


Figure 4: Conceptual graph of the normalized chunking in FastCDC with an expected chunk size of 8 kB, a minimum chunk size of 4 kB, and a maximum chunk size of 16 kB.

2.1.6. Asymmetric Extremum

Unlike the previously presented CDC algorithms, *Asymmetric Extremum (AE)* [8] is not based on a rolling hash function. It instead builds on MAXP [29] which is a novel approach to deal with the boundary-shift problem. MAXP refers to the idea of finding local extreme values to determine new chunk boundaries. In more detail, a fixed-size window w is slid over the input stream on a byte-by-byte basis. Inside the current sliding window, the central position i is set as a new chunk cut-point in the file f *iff* $\forall j \neq i, j \in [i-h, i+h] : F(i) > F(j)$, where h refers to the bidirectional "horizon" of i, i.e., $w = 2h + 1$, and $F(i)$ refers to the byte value at position i. In simpler terms, it is verified if the sliding window has a local maximum at its center.

This approach naturally solves the issue of pathologically small chunk sizes and altogether provides better chunk-size variance and less memory consumption than Rabin-based algorithms. However, it still suffers from low chunking throughput [8].

The authors of AE address this problem as they observed that it is sufficient (regarding the boundary-shift problem) to look for extreme values in an *asymmetric* local range rather than a *symmetric* local range as in MAXP. Symmetric and asymmetric refer to the left and right window of i. In AE, the left window is not fixed. Precisely stated, AE changes the equation in that it defines the respectively next cut-point to be the (smallest) position $i + w$ for which the following predicate is true.

$$\forall j \in (c_{prev}, i) : F(i) > F(j) \ \wedge \ \forall k \in (i, i+w] : F(i) \geq F(k) \tag{12}$$

The procedure can also be understood from Figure 5 where the right window w remains constant but the left window is not and as can be seen for c_4, can even be of size 0. Note how unlike in MAXP, not the local maximum i is chosen as a chunk boundary but the end of the sliding window $i + w$.

This is beneficial as it circumvents the necessity to recheck prior bytes and ultimately contributes to the superior chunking throughput in AE. However, the authors of AE admit that this strategy may decrease the deduplication ratio slightly [8].

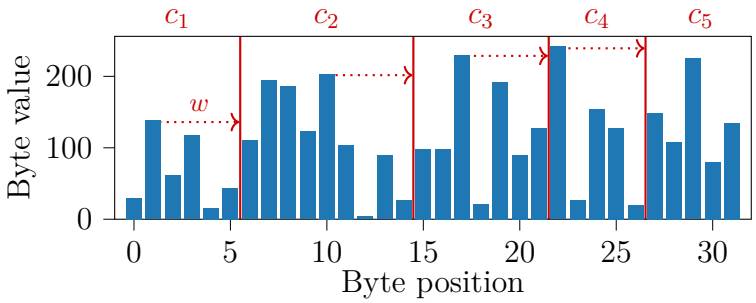


Figure 5: Demonstration of the AE chunking algorithm on a sequence of 32 bytes and a horizon $h = 4$. The vertical red lines mark the cut points which then determine the resulting chunks c_i.

2.2. IPFS

The InterPlanetary File System (IPFS) is a peer-to-peer distributed file system, first released in 2014 [30]. Its ideas address problems in the current HTTP web such as the hosting and distribution of large datasets and the centralization of content —the latter posing a vulnerability to censorships [31]. Instead of accessing central servers to retrieve a file (or a website), content references in IPFS are distributed in a DHT and the file is addressed by an identifier, called the CID (for *content identifier*), which essentially is the hash value of the file. This automatically reduces redundancies, e.g., in the case where assets from popular CSS or JS libraries do not have to be retrieved every time again.

Furthermore, IPFS introduces a versioned web with a data structure similar to the one used in Git to manage branches, commits, files, etc. This feature further helps reduce redundancies with updating versions of content. But it also helps to create a tamperproof web where data is inherently immutable.

In general, the features in IPFS contribute to the ideas of Web3, which is essentially a decentralized web.

2.2.1. Data Structure, Blocks, and CIDs

In IPFS, everything is a block and every block is addressed by its CID. Blocks are connected to each other in a directed acyclic graph (DAG). Every directory and every file in IPFS is represented by a block. In addition, IPFS will split bigger files into chunks, which then each become a new block, referenced by the file's root block. If a file or a directory link to so many blocks that it exceeds the block's size limit, IPFS will split the block into new blocks which each carry a subset of those references. In graph terminology, it is the leafs that represent the raw contents. Figure 6 shows a schematic representation of this structure. Note, how a node can have multiple parent nodes. E.g., node G could be a chunk of a file B and another file D. This is effectively what successful deduplication would look like.

A block is encoded in a format called *InterPlanetary Linked Data (IPLD)*. IPLD is

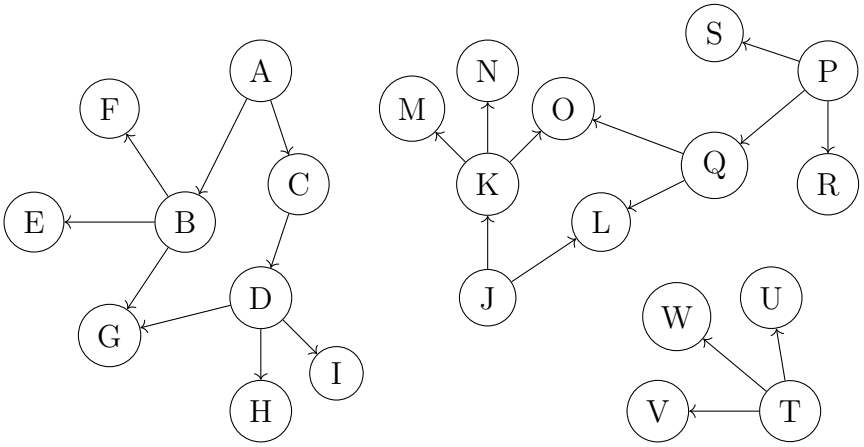


Figure 6: Conceptual view of the DAG of blocks in IPFS.

a data model that contains an array of links to other blocks, i.e., their CIDs, which assemble the current block. An IPLD can also contain an array of bytes, i.e., the raw contents of the block. However, in newer versions of IPFS, raw contents are not encoded as an IPLD anymore. The IPLD data model also reveals whether the block represents a directory or a file.

The CID, as previously explained, is essentially the hash value of the content of that block. In more detail, CIDs follow a special format that also appends some metadata to the hash. This helps to remain flexible with the algorithm applied and interpret the hash accordingly. Initially, this format included just the specifier of the algorithm (e.g., `sha256`), the length of the hash value, and the hash value itself, which in sum results in the so-called *multihash* format. The *multihash*, encoded in `base58btc`, produced the final CID string, now referred to as CIDv0. Finally, the formula to produce the CID (with CIDv0) for a block in IPFS looks like the following.

`base58btc(<algo><length><hash>)`

However, the data itself can be encoded with CBOR, Protobuf, JSON, and other formats. This posed a new requirement that led to the development of CIDv1, which now should include the codec in addition to the multihash. Furthermore, for the new format to remain future-proof, a version prefix is added. Moreover, it was noticed that different base encodings might be necessary depending on environment constraints (e.g., DNS names). So lastly, also the base encoding type became flexible with CIDv1. The final formula to produce a CID string (starting with CIDv1) can be described as the following.

`<base>(<version><codec><algo><length><hash>)`

You can say, that the values for `<base>` and `<codec>` are implicit in CIDv0, namely `base58btc` and `dag-pb`, respectively. So far, there has not been a CIDv2, so we only distinguish between CIDv0 and CIDv1. Having this self-describing CID has the other

benefit that some information about the block can be retrieved without having to query it from the network. A major difference between CIDv0 and CIDv1 addresses is that in CIDv0, every block is encoded as a *dag-pb*, whereas in CIDv1, raw contents (and raw contents only) are encoded as *raw*. This circumstance had an effect on the implementation of the empirical analysis (cf. Algorithm 1).

2.2.2. Deduplication and Chunking in IPFS

Globally identifying and removing redundancy in content makes for a more efficient distribution system and leaves space for redundancy with the intention of increasing reliability of a specific file. This is not a trivial task since even slight variations of a file produce totally different hash values, as can be the case with incremental updates of some content. Often also, websites bundle their CSS or JavaScript assets to a single file representing a concatenation of various libraries and snippets. Adding such an entity as a single object to IPFS will most likely not benefit from the performance effects promised in the introduction of this chapter.

This is one —although not the only— reason why files get chunked into smaller pieces before added to the system. Generally, smaller chunks increase the chances for duplicates.

As presented in Section 2.1, many algorithms for chunking exist. By default, IPFS uses FSC and splits files into chunks of 256 kB. However, they have included support for two content-defined chunking algorithms as well: Rabin-based and Buzhash.

The following shows an excerpt from the Kubo CLI (IPFS CLI) reference [32] and is an option that can be set with the `ipfs add` command, i.e., when a file is added/uploaded to the network.

```
-s, --chunker           string - Chunking algorithm, size-[bytes],
                                 rabin-[min]-[avg]-[max] or buzhash.
                                 Default: size-262144.
```

As can be seen, not only the chunking algorithm can be chosen, but in the case of FSC, the chunk size can be specified, and in the case of Rabin-based chunking, the minimum, maximum, and desired average chunk size can be configured. It is not clear why no setting exists for the application of Buzhash. The average chunk size for Buzhash was hardcoded to 256 kB [33]. The `--chunker` option can be utilized to improve deduplication when the type of file would benefit from it. For unknown reasons, neither Rabin-based chunking nor Buzhash was chosen as the default chunking algorithm. However, we suspect that most files in IPFS cannot benefit greatly from deduplication (e.g., because of file types with rather arbitrary byte structures due to compression) or just files in the set of files an individual user downloads rarely has (much) redundant content. Perhaps this is why IPFS prioritized chunking speed which, obviously, is the highest with plain FSC.

3. Related Work

Data storage and network traffic are expensive. Traditionally, compression algorithms were used to minimize file sizes. The problem with these algorithms, however, is that they work great on single files and also on set of files, but scale poorly with larger amounts of data in terms of compression and decompression efficiency [16]. For that reason, the research for techniques of deduplication in large-scale data systems and disk backup storage became pressing. Deduplication on that scale is usually achieved by finding and eliminating duplicated files or blocks of data, rather than the byte-level approach of traditional compression algorithms. However, the ideas of compression are interesting to study and important to discuss in the attempt of finding an optimal deduplication strategy, even for a large-scale system like IPFS. We give an overview of compression algorithms and discuss their utility and downsides in Section 3.1. On that reference, we afterwards present the idea of *near-duplicate detection* (NDD) in Section 3.2 which is addressing the issues of internal compression in multimedia files regarding system-wide deduplication.

While simplistic file-level deduplication can already achieve great effects on reducing storage costs [34], especially when a large proportion of the files rarely change, fine-grained (i.e., block-level) deduplication does create more opportunities for space savings. We can observe block-level deduplication in various productive and popular applications. For example, rsync, when it compares a file on a local system with the corresponding file on a remote system, splits files into chunks and then compares its hash values to only transfer the differing chunks, i.e., potentially only a subset of the file it wants to synchronize. In the original paper for rsync [35], the authors state that the file should be split into fixed-size chunks of 500–1000 B. However, from the current and official implementation of rsync [36], we derive that files are being split into fixed-size chunks of 32 kB.

Another example of an application of chunking is BitTorrent. It is especially interesting in our context because it has been the most popular realization of distributed P2P file sharing. In BitTorrent, files are split into fixed-size chunks they refer to as “pieces”, and then even further into so-called “sub-pieces” [37]. The size of a piece is chosen by the user, but is most commonly set to 256 kB, similarly to IPFS. However, it is important to understand that every torrent deploys its own swarm. Hence, there is no cross-torrent deduplication and the file (of a torrent) would in the most cases be compressed in the first place anyway. So the authors did not have the goal of deduplication with their chunking strategy in mind, but rather the advantages of parallelizing file downloads.

The effectiveness of chunking with the goal of deduplication is not always assured and depends largely on the targeted file type, as we will see in Section 3.1 and also again throughout this thesis. Therefore, it is not always certain whether file-level deduplication (in a large-scale file system) does not already achieve similar results but without the complexity added through chunking. The authors of [5] conducted in their study, in which the file system contents of 857 desktop computers over the time span of one month were analyzed, that the file-level strategy only achieves $\approx 89\%$ of space savings in live file systems compared to block-level deduplication. This analysis was based on fixed-size

chunks of 64 kB. The deduplication success of the block-level approach gets even better with smaller chosen chunk sizes. Furthermore, with their "most aggressive" approach (Rabin-based chunking and an expected chunk size of 8 kB), the file-level strategy only achieved 72% for live file systems and 87% for backup images, respectively, of space savings. However, as the authors concluded, the gains may not justify the performance cost and complexity introduced with more advanced chunking algorithms and also with smaller chunk sizes. Making a case against block-level deduplication in general, they also argue that, at least for primary storage systems, trading away sequentiality in disk reading for space savings, might not be worth it. We must however acknowledge that this study is already 13 years old, assumed the hardware conditions of its time and did not consider today's state-of-the-art CDC algorithms (like FastCDC [9] or AE [8]). This study was led in part by Microsoft Research, and in our efforts to learn how Microsoft manages deduplication today (in similar settings as previously described), we found that they introduced the option for deduplication in NTFS data volumes with Windows Server 2012 [38]. They also introduced network-wide deduplication schemes, which has some aspects that are similar to deduplication in IPFS. There also exist other modern file systems, e.g., ZFS, that integrate mechanisms to increase space efficiency on primary storages. We explore these ideas and discuss their adoptability for IPFS in Section 3.3.

Finally, we also discuss the utility of two chunking strategies that were proposed to improve computational efficiency (see Section 3.4) or deduplication performance (see Section **??**). Those strategies do not represent other chunking algorithms like FastCDC or AE but are algorithmic approaches that work on top of any given CDC algorithm.

3.1. File Compression

Many popular file formats, like ZIP or PNG use compression to minify the resulting file size. In most compression algorithms [13, 12, 39], the contents are scanned for common byte patterns and deduplicated by pointing to indexes referencing those bytes. Similar ways of compression are also integrated in most rich media file formats, like images [40, 41, 42, 43, 44], audio [45], and video [46]. Other media-specific approaches do exist and can be divided into the category of *lossy* and *lossless* compression technique. E.g., in videos files or image formats like JPEG, it is common to remove psycho-visual redundancies, i.e., eradicating information that is less significant in our visual processing [40, 41, 42]. This is an example of a lossy compression technique, where the result might be satisfactory to the viewer, but nevertheless, information is lost. Many techniques exist however, that are able to reduce the file size without losing any bit of information. Run length coding is a such a technique and is commonly applied to images [40, 41, 42]. The idea is to detect structural repetitions and replace data with tuples of the recurring value and the quantity of repetitions. Lossy compression has a greater potential to reduce file size. It might however not always be desirable in the context of file sharing because of the inevitable effects on the image, video, or audio quality. Lossless compression, on the other hand, could be interesting, even in the context of IPFS, since required storage space and network traffic would be

reduced. It is common for centralized file sharing platforms, e.g., Dropbox, to apply file compression on user-uploaded files [47, 48, 49]. Nevertheless, the computing costs of this operation need to be considered, which in the case of a decentralized system like IPFS, would always be executed on the client-side, i.e., whenever a file is added to IPFS. Compressing a large file, e.g., a movie, can take a lot of time. Depending on the hardware and network speed of the client, this process can easily take longer than the file upload itself. While we have intuitive concerns about the applicability of client-side compression of multimedia files in IPFS, we have not finally evaluated it and its analysis might be a topic of future work.

Compression of text-based files or rather file archives is a different story. Although this pursues the same goal as what we are trying to do for IPFS through the implementation of clever chunking algorithms, it actually interferes with the ideas of content-defined chunking. The reason being that small modifications of a file that gets compressed afterwards do not result in only local modifications of the file. They instead have the potential to change the whole layout of the file because of a reevaluation of common byte patterns.

In [6], the idea to first decompress a compressed file, then suppress duplicates on the result, and then again apply the compression on the file, was proposed. The authors measured the potential storage space savings on datasets of four respective file formats: GZ, BZ2, ZIP, and Z. Indeed, the proposed strategy was successfully increasing the storage space savings by up to 20% in the datasets of gz and BZ2, 138% for Z, and 328% for ZIP. The authors attribute the extreme result in the ZIP dataset to ZIP deflating one file at a time, missing out on the potential size reduction of intra-file compression. A similar experiment has been completed in [23]. Both papers came to the conclusion, that the results might not justify the computational overhead of the process. The authors in [23] further support this hypothesis by the fact that while the strategy has a great impact on zip or gzip files, those file types made up for a small percentage in their datasets (entire home directories, mail attachments, ...). In effect, the overall benefits of this strategy could only be accounted to 1-2% in storage space savings. The authors of [6] made the additional statement that for distributed environments in particular, it would be very difficult to implement such a strategy effectively and add a lot of complexity to it. It is for those reasons that we did not decide to further explore this idea in the context of IPFS.

3.2. Near Duplicate Detection with Multimedia Files

We assume a large portion of files exchanged on IPFS to be multimedia files, e.g., images and videos. According to [44], images are among the most common shared file types on cloud storages. While those files individually make very good candidates for deduplication by the means of compression, it is very difficult to deduplicate them across multiple files, i.e., when the same content appears in different versions. The internal compression that takes place in a PNG or in a MP3 file, for example, already negates the effects promised by CDC when a visual, e.g., a single pixel, or acoustic detail changes. But multimedia files can look entirely different even if the information

on display is exactly the same. This can happen through the use of a different encoder or file format (e.g., FLAC vs. MP3). The file could also represent or include a subpart of another file. It becomes even more complicated with more-dimensional resources such as videos. For example, two files could show the same movie, but have been recorded slightly differently. That is, they include the audio for another target-language, or perhaps include advertisements in different positions of the file (see Figure 7). Image, audio, and video resources can also furthermore appear in different resolutions. In the human perception, it is easy to spot duplicated content/information and intuitively, there seems to be the potential of deduplication. Digitally, i.e., from the point of view of a computer, this is not a trivial task.

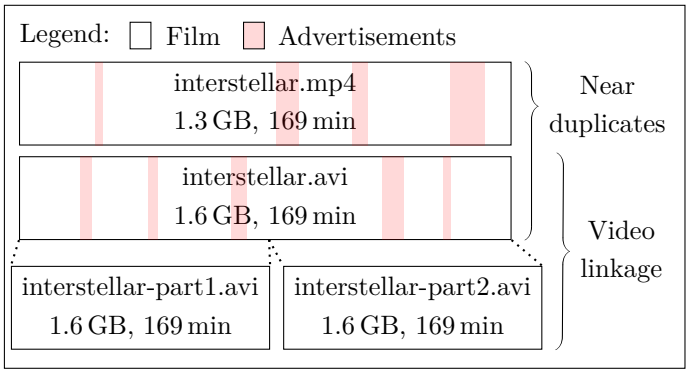


Figure 7: Near duplicates and video linkage. Adapted from [50].

Finding duplicates, in the forms just described, is generally known as the problem of *near duplicate detection (NDD)*. NDD algorithms do not exist as universal approaches but specific to resource types and can be found for images [51, 52], audio [53], as well as for video files [54, 55]. Those techniques are applied to reduce storage requirements [56] but also to detect copyright infringements, e.g., on platforms like YouTube [57, 58].

The approach of these techniques can conceptually be described as the extraction of features and normalization of the contents with a heuristic matching process afterwards. Querying for nearly-identical content is challenging for P2P systems, as they are based on very costly, pair-wise comparisons [59]. Commonly, this is solved by the deployment of textitLocality Sensitive Hashing (LSH) [60, 61] (e.g., in [53, 62, 63]).

Peer-to-Peer Optimized Near Duplicate detection (POND) is an LSH-based algorithm proposed in [50] that focuses on NDD in DHT-based P2P environments. It is furthermore extended by their approach to address the problem of video linkage (see Figure 7), i.e., linking together videos which share large near-duplicate segments.

As could be shown, the research regarding NDD strategies for multimedia content in P2P networks is very broad. We acknowledge that those strategies, when successfully deployed, have the power to increase the efficiency of file retrieval, e.g., by parallelizing downloads from multiple peers where each provides a segment that can be linked to the requested resource. However, we think that NDD approaches are employed better in systems specializing on multimedia file retrieval where the emphasis is less on the

uncorrupted authenticity of content and more on the perceived information of a media file. In our judgement, the approaches do not follow the vision of IPFS and moreover would add a lot of undesired complexity to it. Those ideas are however interesting to contemplate since multimedia files do in fact make a large proportion of files exchanged on the internet and they, by nature, carry greater file sizes than text-based files.

3.3. Deduplication in Modern File Systems

As many studies revealed [5, 25, 64], data deduplication can reduce space requirements in these environments by up to 40–60%. Therefore many advanced file systems support block-level deduplication to increase space efficiency on primary storages. For example, ZFS [65], GPFS [66], HDFS [67], and Ceph [68] employ fixed-size chunking[1] Windows [38, 70] enables data deduplication through a Rabin-based CDC algorithm and in combination with compression of the generated chunks. The CDC algorithm is parameterized to produce chunks between 32 and 128 kB and with an average chunk size of 64 kB. Another interesting feature of the implementation of data deduplication in Windows is that files of specific types can be excluded. This is recommended for file types like PNG that already have internal compression and therefore would not benefit much from the deduplication anyway.

Deduplication in primary storages, e.g., on personal computers or servers, comes with its own challenges: Chunking, fingerprinting, and indexing incur latencies when files are written and also, due to disk fragmentation, when files are read. Furthermore, those operations block resources (CPU, as well as disk I/O and RAM) which might interfere with other business processes, i.e., slow down the general performance. Many file systems counter this by pursuing strategies to ensure chunking takes place in the background at the moments when resources are not utilized otherwise.

The authors of [6] made a further interesting observation regarding the nature of primary storage file systems: Most file accesses are read-only and it is mostly only a small set of files that is written. Furthermore, as the authors of [71] add, most files are not re-accessed for a long time after the initial write. These insights allow modern file systems to organize files into frequently accessed files (active files) and rarely read/written files (inactive files) where the drawbacks of deduplication undergo less noticed. TDDFS [72] acknowledges this in a two-tier-system that maintains active files on a fast tier and inactive files on a slow tier. The tiers could further be implemented by a high perming disk (e.g., an SSD) for the fast tier and a more economical option for the slow tier. Files in the slow tier are deduplicated using CDC. When an inactive file is accessed, the missing data chunks are migrated from the slow tier to the fast tier. However, as also the fast tier maintains an index table about its data chunks, the data required from the slow tier can be further deduplicated to optimize migration speed. The goal in TDDFS is to provide maximal performance for operations on the active files, yet fast migrations of files from the slow tier to the fast tier when an inactive file

[1] As we understand from [69], ZFS applies fixed but variable chunk sizes depending on the file type. However, we do not consider this CDC.

is accessed. Windows implements the same concept to its file system as it bypasses chunking and deduplication until the file reaches a certain age (counting from the last time it has been written to).

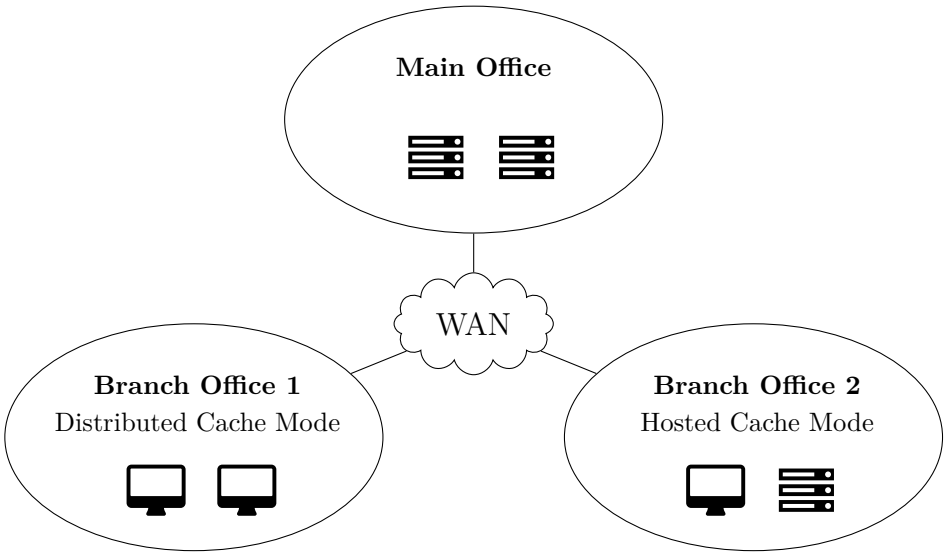


Figure 8: BranchCache network setting.

As with Windows, it furthermore can offer distributed deduplication in a Windows-Server–based intranet through a service called BranchCache [73]. BranchCache optimizes bandwidth utilization for file retrieval over the WAN by establishing a distributed cache system for data inside the intranet. See Figure 8 as an example for an intranet. When a PC in a branch office queries a file from a remote server, BranchCache will fetch the content from dedicated servers (hosted cache servers in the same office or in the main office) and cache the content on either a dedicated ("hosted cache") server or on PCs in the branch office. This depends on the mode BranchCache is configured with (hosted cache vs. distributed cache). It then in turn allows the PCs inside that branch office to retrieve the data locally instead of over the WAN. Subsequent network requests for a file will now be answered with the file information, i.e., hash values of the data chunks the file is comprised of. Those hash values—very much in the fashion of content addressing in IPFS—are then used to query the corresponding objects locally.

Although in a closed and centralized environment, we can draw many parallels to file sharing and deduplication in IPFS. Especially with the configuration for distributed cache, where data objects are stored at and retrieved from PCs (peers), the technological concepts are very similar. It is therefore interesting to see the choices Microsoft has made with this system, i.e., block-level deduplication using CDC, an average chunk size of 64 kB, compression of generated chunks, and additional options as file type exclusion etc. However, before adopting the same ideas for IPFS, we have to recognize that the systems have different settings and therefore different goals.

In the corporate setting, we assume multiple offices with peers connected to each other in a LAN. File retrieval over a LAN is of course much faster than over the internet,

so the main objective becomes to make data available inside those local networks.

Another aspect to consider is the fundamentally different interaction with the systems. In the Windows Server setup, local file deduplication happens in the background and when resources are available. That means, assuming a good implementation, that the user never experiences performance losses in their other tasks, nor do they ever have to wait for a deduplication process to finish. We can therefore think of the computing costs of this operation (the chunking algorithm in particular) as practically non-existent or irrelevant.

A further huge aspect is of course the nature of a centralized environment and the benefits that come with it. In the BranchCache network setting, there will be dedicated servers to coordinate intra-peer deduplication and file sharing, and presumably also servers with better bandwidths, e.g., in the main office. As we understand, file information and chunk indices are shared with the network in an automatic manner. When a peer requests a file over WAN, the hosted cache servers can execute the file retrieval in their place and then the system can check if it "knows" the file or a subset of its chunks, and further, if those chunks are available inside the peer's LAN (or the peer's PC itself). Because those checks only require to transmit a handful of hash values, some of it inside the LAN, it can all be performed very efficiently. A lot of those processes would violate privacy concerns in any decentralized environment, but they are perfectly possible in the corporate scenario and in fact allow for a great application of optimization techniques.

In summary, automatic local file deduplication and automatic file sharing as background processes are what make computation time and resource utilization of those processes (when implemented ideally) irrelevant. We conclude that this is the major reason for which we can find the employment of deduplication methods (namely, CDC, relative small target chunk sizes of 64 kB, and post-chunking compression) more expensive than what we could observe in other productive systems, like BitTorrent [37], rsync [35], and of course, IPFS itself.

3.4. Parallelized Chunking

Chunking speed is still a concern that hinders the adoption of CDC algorithms. Accordingly, especially in the last decade, there has been a great focus and effort in increasing computational speed of these algorithms. CDC algorithms like [9] and [8] focused on the optimization of the algorithmic mechanics, such that the number of required processor or memory access operations is reduced to a minimum, thus achieving higher chunking throughput. Since most computers nowadays run multicore processors, it is interesting to think about ways on how to exploit that fact for parallelized algorithm executions. The basic idea would be to split a large file into segments of equal size and run the chunking algorithm on all segments, each in another thread. Generally, this is not trivial for CDC algorithms because of the content dependency and therefore inherent sequentiality of these algorithms. That is, the last chunk boundary in a segment is determined because the segments ends there. Further, the next segment's first chunk is set on the basis of only its first bytes which then in effect sets a chunking boundary

that will also effect the subsequent chunk. This makes the execution of CDC difficult to parallelize. That is, as long as we demand *chunking invariability* [74] (see Definition 3.1).

While we can run CDC in parallel and neglect chunking invariability, it would have a direct, negative effect on the deduplication ratio. This is obvious since the chunk boundaries would not be set based on the file contents anymore, but rather arbitrarily.

Definition 3.1. *Chunking invariability* refers to the condition that the parallel execution of a CDC algorithm must lead to the same set of chunks as the sequential execution of the same algorithm with the same input file.

However, there is another way to still exploit parallelism in CDC, and furthermore, researchers have invented clever ways to coalesce adjacent segments that have run CDC without dispensing chunking invariability.

The first strategy is based on the realization that creating a chunk consists of more than one step [74, 75]. Chunking (i.e., determining the next chunk boundary) is merely the first stage in the process which can be subdivided into the following sequential steps:

S1. Chunking

S2. Hashing for fingerprinting

S3. Indexing

S4. Writing file metadata and chunk data

This pipeline can be run independently, which means in parallel, for each new chunk. In other words, we can compute the hash for the first chunk while the CDC algorithm is busy determining the second chunk. This idea is schematically visualized in Figure 9. Note also that some of the steps are CPU-demanding (S1 & S2), while others depend on memory and/or disk reading and writing capacity (S3 & S4). This is interesting because it allows us to not only make the most of having multiple CPU units but also of different hardware components now working in the same time. Note further that this strategy is applicable to *any* CDC algorithm and, of course, to FSC as well.

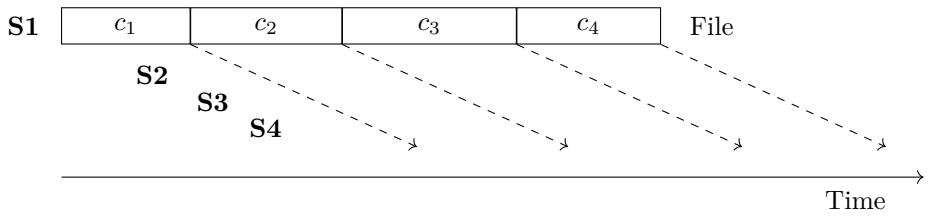


Figure 9: Parallel processing of the pipeline that creates a chunk after each new chunk boundary determination.

As mentioned previously however, there also exist algorithms that allow CDC of segments of a file in parallel which still preserve chunking invariability [74, 75, 76]. This can further be combined with the previous strategy. To this end, we add parallelization

on another level by not chunking the whole file sequentially, but sequentially chunking multiple segments of a file in parallel (see Figure 10).

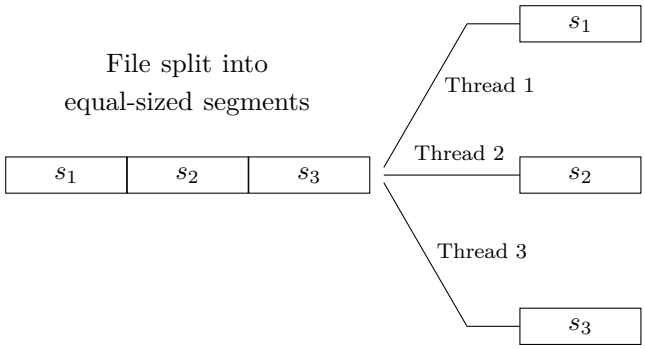


Figure 10: Concurrent chunking of segments of a file on multiple threads.

We find one algorithm in particular, named *Multithreaded Content-Based File Chunking (MUCH)* [74], interesting because it is agnostic to the CDC algorithm. The procedure is based on a method the authors call *Dual Mode Chunking* and it is applied on each segment. It consists of a *slow* mode and an *accelerated* mode. The only difference is that, during slow mode, the chunking algorithm will not enforce minimum and maximum chunk sizes. Chunking starts in slow mode and switches to accelerated mode as soon as it encounters the marker chunk c_m. The *marker chunk* is defined as the first chunk (or second, if $m = 0$) to fulfill the following condition.

$$MinChunkSize \leq |c_m| \leq MaxChunkSize - MinChunkSize \tag{13}$$

All subsequent chunks will then be chunked with the accelerated mode until the end of the segment is reached. After all threads have finished, the next step is to merge the resulting chunks produced for the segments in all threads without the final and complete sequence of chunks for the file to violate chunking invariability. The authors call this step *chunk marshalling*. To this end, for two adjacent segments A and B, we require the last produced chunk in A, let that be a_n, and the first chunks in B up until the marker chunk, i.e., $b_1, \ldots, b_m$. This range of chunks, let us refer to this sequence as $X = (a_n, b_1, \ldots, b_m)$, are likely to have boundaries corrupted through the method of parallelized chunking, and furthermore, chunk sizes that are not compliant with the minimum and maximum chunk size setting. Both issues are targeted in the procedure of chunk marshalling, which works in the following substeps:

1. **Chunk Coalescing:** For each $c_i \in X$, coalesce c_i with c_{i+1} if $|c_i| < MinChunkSize$.
2. **Chunk Splitting:** For each $c_i \in X$, split c_i into (c_{i_1}, c_{i_2}) with $|c_{i_1}| = MaxChunkSize$ if $|c_i| > MaxChunkSize$.

The algorithms of the substeps are simplified for better readability. Mind that during chunk coalescing, when c_i and c_{i+1} are getting coalesced, the algorithm continues at

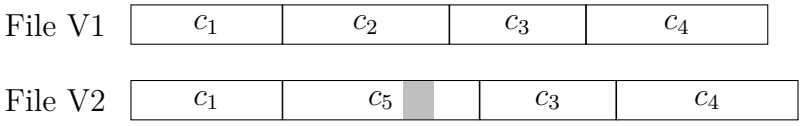


Figure 11: V1 and V2 are two files split using CDC. V2 is a version of V1 with a local insertion of new bytes (area marked in gray).

c_{i+2}. Similarly, when c_i is split into (c_{i_1}, c_{i_2}), the same procedure will recursively be applied to c_{i_2}. Because of the predicate set on the marker chunk (cf. Equation 13), the chunk marshalling is guaranteed to result in chunks compliant to the minimum and maximum chunk size restrictions. Finally, this methodology will also result in the coherent sequence of chunks that comply with chunking invariability.

In a experimental analysis of MUCH, the authors were able to conclude that their methodology is able to reach a performance benefit that scales linearly with the number of CPU cores. I.e., a quad-core CPU would achieve a performance gain of 400% (if the storage device is sufficiently fast).

Moreover, other techniques exist that are able to solve the issue of chunking invariability in multithreaded chunking which work on rolling hash CDC algorithms, such as Rabin and FastCDC.

The authors of [75] introduce *P-Dedupe*, a system to exploit parallelism in CDC leveraging the pipelining scheme we have seen before and another algorithm to coalesce adjacent segments. To this end, they describe a "divide, conquer, and connect" approach, which refers to the concept of chunking multiple segments of a file in parallelized threads, and then essentially re-chunking a small part at the end or the beginning of two adjacent segments, respectively. P-Dedupe achieves chunking invariability by processing a sequence of bytes whose length is determined by the rolling window size at the end/beginning of two adjacent segments. The authors claim near-linear performance gains with multiple parallelized threads. Similarly, although different in its technical details, operates SS-CDC, which is another algorithm to connect adjacent segments with parallelized CDC based on rolling hashes proposed in [76]. The authors in addition focused on the exploitation of instruction-level SIMD parallelism, which is also available in modern processors. In their experimental evaluation, they concluded a performance speedup which is much superior to existing systems like P-Dedupe, and scales superlinearly with multiple CPU cores.

3.5. Rechunking Non-duplicate Chunks

Block-level deduplication suffers from the dilemma of higher versus smaller target chunk sizes. Larger chunks are less likely to be redundant within a data stream. On the other hand, smaller chunks produce more metadata/index overhead (cf. Section 2.1.1). System designers therefore struggle to determine the right granularity.

As illustrated in Figure 11, any small modification in a file will change at least one chunk. In this example, only c_1, c_3, and c_4 could be deduplicated when storing files

$V1$ and $V2$. However, there is more potential for deduplication as c_2 and c_5 contain further redundant byte sequences. Researchers have therefore suggested to re-chunk non-duplicate chunks at finer granularity, i.e., choosing a much smaller target chunk size within this region. This approach creates a compromise between the benefits of larger chunk sizes and the increased deduplication ratio of smaller chunk sizes which are only used in specifically targeted regions of a file.

For example, bimodal content-defined chunking [77] employs two target chunk sizes which are then switched dynamically. Initially, a file is chunked with the larger of both chunk sizes (using a CDC algorithm). In the next step, the system is queried for the existence of a similar chunk sequence with a non-duplicate chunk between two duplicate chunks, as is the case with V1 and V2 from Figure 11. The non-duplicate but duplicate-adjacent chunks now get chunked again into finer pieces using the smaller target chunk size. This will potentially detect more redundancies and ultimately create more duplicated chunks. To cope with the drawbacks of having to maintain a lot of small chunks, the adjacent duplicates are further amalgamated back into single chunks. The authors were able to achieve a significantly increased deduplication ratio on their simulation of bimodal chunking on backup storage systems.

Although this strategy is interesting from a conceptual point of view, we do not think that its ideas are suitable for IPFS. Backup storage systems have the special characteristic of storing many versions of an evolving dataset. We are not convinced that this characteristic is shared by the data landscape of IPFs to the same extent. Furthermore, bimodal chunking relies on a system's ability for fast existence queries. Those would add costy network operations to the otherwise local chunking process in IPFS.

4. Analysis and Comparison of Chunking Algorithms

This section presents and discusses an analysis of the chunking algorithms introduced in Section 2.1. To this end, the performance of the two previously identified key quality metrics, i.e., the computing speed and (most interesting for us) the deduplication ratio, is evaluated on different kinds of data, both theoretically and experimentally. We have further extended our study by the metrics of the average chunk size and chunk-size variance that is generated with every algorithm in every setting. This helps us to understand and interpret the results produced for our key metrics in a better way.

The results of the following analyses were produced using a framework (collection of programs) that has been implemented as part of this thesis [78]. It is important to notice that the results of this comparison will depend on the implementations of the algorithms used. For example, there exist many different versions and derivates of Rabin. This analysis uses the exact libraries of FSC, Rabin, and Buzhash that currently are also used in the official implementation of IPFS [33] (written in Go). An implementation of FastCDC was found on GitHub [79]. We developed an implementation AE ourselves [80] as there was none in existence for Go.

4.1. Methodology and Datasets

The goal of this analysis is to see how well the chunking algorithms perform on various kinds of data, e.g., on image files versus on text files. This will also give insight to where CDC algorithms (e.g., Rabin or FastCDC) actually make sense, and where they do not. We expect the chunking speed not to differ between file types as none of the algorithms have a semantic understanding of the file contents. For this reason, the analysis of the computational performance and the analysis of the deduplication ratio follow separate methodologies.

Commonly in both methodologies, we aim to equalize the conditions of all chunking algorithms for better comparability. This requires configuring the same target chunk size, since it has a direct influence on both computational efficiency and deduplication performance. The same applies for the minimum and maximum chunk size, which are two more parameters available in every chunking algorithm's configuration (apart from FSC). It is even a requirement for FastCDC, which needs them to internally parameterize for normalized chunking (cf. Section 2.1.5).

In our analysis of deduplication performance, we furthermore want to experiment with target chunk sizes. However, we are restricted in our abilities to play with these settings because of the fixed implementation of the library that we use for Buzhash. In this implementation, Buzhash performs on a target chunk size of 256 KiB, a minimum chunk size of 128 KiB, and a maximum chunk size of 512 KiB. This means we can only compare Buzhash to the other algorithms aligning to this specific configuration.

We imposed the configuration of the Buzhash algorithm on the other algorithms. Furthermore, for every target chunk size x in our analyses, we specify $\frac{x}{2}$ as the minimum and $2x$ as the maximum chunk size. We make an exception for AE though, where the theoretical minimum chunk size is $x - \frac{x}{e-1}$. Here, we acknowledge that $x - \frac{x}{e-1} < \frac{x}{2}$ for

every $x > 0$ and that it also grows slower, i.e., the discrepancy grows with x. However, in the range of chunk sizes we look at, we consider this discrepancy as marginal and therefore to not noticeably contribute to the final results. Nevertheless, of course, it disqualifies Buzhash for our analyses with $x \neq 256\,\text{KiB}$.

4.1.1. Computational Efficiency

By computational efficiency we mainly refer to the speed by which a chunking algorithm is able to split a file into its chunks. For the sake of completeness, and because we get this data anyway, we also look at the memory efficiency by which the algorithms perform. However, we will only take brief notice of those results and rather focus on the computing speed.

To measure the computational efficiency, a file comprised of a random byte sequence of 100 MB is generated and the same file is chunked using each algorithm on a VM running Ubuntu with an Intel Xeon Gold 6154 CPU at 3.00GHz. We run the algorithms on the configuration of a target chunk size of 256 KiB. This is what IPFS uses by default, but it is also what the library of Buzhash, which does not allow any customization of the parameters, uses internally. While other target chunk sizes would produce other absolute results, we assume the relative differences between the results to be similar.

4.1.2. Deduplication Performance

The analysis of the deduplication ratio has different requirements for the test data. In order to get an accurate impression on how the algorithms would perform in the real world, the methodology for this analysis also requires realistic data. For that matter, different datasets were collected, listed in Table 1.

Dataset	Size	Description
TAR	19 GB	A total of 65 tarred versions of source code from the projects GCC, GDB, and Emacs (uncompressed).
LNX	15 GB	A selection of ISO images from different Linux distributions and versions.
WEB	19 GB	30 day-by-day snapshots (month of June 2022) of the website nytimes.com using wget and a depth level of 3.
DLF	13 GB	Files downloaded on personal computers.
PNG	1 GB	PNG files scraped from crawling the web.
JPEG	1 GB	JPEG files scraped from crawling the web.
PDF	5 GB	PDF files scraped from crawling the web.
MOV	7 GB	A selection of movies in MP4 or MKV format.

Table 1: Datasets.

The datasets can basically be split into two categories. In the first category, we have

TAR, WEB, LNX, and DLF. Especially for TAR, WEB, and LNX, it is the case that they naturally make good candidates for deduplication, since the content inside the datasets is consecutive. For example, we expect it to be likely for a website that at least parts of it, like the header and footer, but also the stylesheets, scripts, and other assets, will not change on a daily basis. The dataset WEB is particularly interesting when keeping in mind the use of IPFS in Web3. Furthermore, network traffic is only reduced with deduplicated content within an individual user's requested data. Therefore, the repeated/daily access of a specific website makes a realistic case example for the usage of IPFS.

Our goal with the DLF dataset was to create a sample of mixed data that would reflect the average user's behavior with data downloaded from the internet. To this end, the default folder to store data downloaded from the internet, from three personal computers, was composed, containing over 80 different file types, including executables, text and media files, archives, and much more. We made sure to delete any duplicated files from this dataset, i.e., where the exact same file existed again under another filename.

The second category was added to obtain data on the algorithms' performance on specific popular file types, namely PNG, JPEG, PDF, and MOV (which is actually a mix of MP4 and MKV files). We generally expect very low deduplication ratios in these datasets (especially PNG, JPEG, and MOV) because of the internal compression those file types use. In order to collect a sufficiently large amount of data, we leveraged `wget` to crawl the web, saving files of the desired type to disk. The goal with this was to retrieve a sample that is as unbiased as possible and that would reflect the objects exchanged in a decentralized network (i.e., IPFS) the best.

This meant, in the case for PNG and JPEG, we wanted the images to have any dimension and resolution, and depict various kinds of content (photorealistic pictures, illustrations, etc.). We decided to run wget on *4chan*, which is an anonymous image board website that is known for its very unfiltered and diverse content, which is a desired aspect for what we are trying to achieve. But foremost, we noticed that the image upload algorithm on 4chan is milder on the file than, e.g., on Reddit. That is, the impact on the original file through scaling or compression is less (however, it still exists). It is also easier to crawl than websites like Twitter which detect and block bots trying to crawl their website.

In similar fashion, we crawled `https://semanticscholar.org` to obtain a sample of PDF files. This website is a search machine for scientific literature and it was one of the few we found not to have mechanisms to defend against bots or crawlers. We assess scientific literature to generally be a good source for a set of data that is representable for PDF files, also because it has a good ratio of text and graphics/images.

Finally, movie files are naturally very big, so there was no need for the application of a crawler. For this dataset, a small selection of movies were downloaded from the internet.

The deduplication ratio (and to some degree, the computational efficiency) also depends on the configurations the algorithms are run with, i.e., the desired average chunk size, but also a chunk size's lower and upper limit. As previously mentioned, we

are restricted to a specific configuration and a target chunk size of 256 KiB in Buzhash. This means that while the analysis experiments with multiple configurations of the target chunk size, the most comparable results are achieved at 256 KiB. For the datasets TAR, WEB, LNX, DLF, and MOV, we chose a range (with exponential steps) of target chunk sizes between 32 KiB and 512 KiB. In our opinion, this is a range where the tradeoff between computation speed deduplication ratio is justified and makes sense for the application in IPFS. Furthermore, the smaller chunk sizes (32 KiB and 64 KiB) bring forth the differences in deduplication performance between each algorithm. The strategy in that regard for the datasets PNG, JPEG, and PDF was a little different. By nature, those files are typically rather small, often in the range of 20 to 500 kB. In the cases where the file is already smaller than the target chunk size, the algorithms would produce only a single chunk, and therefore the same results. So to better highlight the attributes of the CDC algorithms, we chose a much lower range of sizes, 256 B to 4 KiB.

In general, to determine the deduplication ratio of a chunking algorithm, all files f_i in the dataset are run through the algorithm to produce a set of chunks. Let this algorithm be $s(x)$, then $C = s(f_1) \cup \ldots \cup s(f_n) = \{c_1, \ldots, c_n\}$ is the set of all chunks in the dataset. The uniqueness of a chunk inside the set is proven by the SHA-1 hash of its content. Finally, the deduplication ratio is the result of $1 - \frac{\sum_{i=1}^{n} |c_i|}{|f|}$.

4.1.3. Average Chunk Size and Chunk-Size Distribution

Lastly, because we recognize that target chunk size does not have to mean average chunk size, we further conducted an analysis of the chunk sizes actually produced. To this end, we recorded the size of each produced chunk along when we run the analysis on each dataset. This is interesting because the average chunk size, as well as the chunk-size variance, can partly explain better or worse deduplication performance. In consequence, this might reveal an advantage or disadvantage of a chunking algorithm that was otherwise hidden. We care about this because it simultaneously hides the cost in computation speed and metadata that comes with chunk sizes lower or higher than anticipated. However, we also think it just generally helps better understanding the results.

4.2. Theoretical Analysis

Before we look into the results of the actual experiment, there are assumptions and expectations we can state in advance based on logical reasoning and existing literature.

4.2.1. Computational Efficiency

FSC follows the simplest algorithm with the least computational steps, which is why we would expect it to outperform any other chunking algorithm in terms of computing speed. Furthermore, Rabin is known to us as one of the earliest CDC algorithms. Buzhash, as well as FastCDC and AE, was proposed as an advancement of Rabin with

a focus to improve the poor computational speed of Rabin. Therefore, we expect Rabin to perform the worst.

We can further make a claim on how FastCDC and AE will perform in comparison to Rabin, based on their literature, as well as how FastCDC should perform in comparison to AE. The authors of AE claim it to be 3× as fast as Rabin [8]. Moreover, the authors of FastCDC claim to be 10× as fast as Rabin and 3x as fast as AE [9]. We have not found any literature revealing data on Buzhash. At this point, we only expect it to be faster than Rabin.

On memory consumption, we know from [8] that AE should be more efficient than Rabin. Other than that we have found no claims on the algorithms' memory performance.

4.2.2. Deduplication Performance

For obvious reasons, we can expect FSC to perform generally worse than any CDC algorithm, and significantly worse depending on the dataset. Depending on the correlation of the files inside a dataset, we expect generally higher deduplication ratios but also an extreme disparity between FSC and the CDC algorithms. High correlation is expected from TAR, WEB, and LNX, whose contents only comprise multiple versions of the same content.

On the other side, for very heterogenous datasets, we expect the deduplication ratio of the CDC algorithms to be lower or closer to that of FSC, as was also found in [6]. However, we still expect significant deduplication for DLF and PDF because of their partly text-based contents and the natural chances for redundancy in this type of data. In a downloads folder (DLF), we assume that it is also not too uncommon to have many versions of the same file, thus benefitting from the effect of correlated data, explained earlier. We expect the worst deduplication ratios to be found for PNG, JPEG, and MOV, because of the internal compression in these file types. With those file types, the result of CDC algorithms is expected to be rather arbitrary and the deduplication ratio not distinguishable from the performance of FSC.

Regarding target chunk sizes, higher chunk sizes will obviously decrease the chance of finding duplicated chunks, and it will do so increasingly. Moreover, it will converge to 0. Hence, we expect to see a general decline of deduplication ratio with higher chunk sizes.

Furthermore, we can set some expectations based on claims made by the respective authors of the different chunking algorithms. The authors of AE assert that their algorithm attains "comparable or higher deduplication efficiency" than Rabin-based approaches [8]. Their evaluation includes a dataset whose contents are very similar to our TAR dataset, and another dataset comparable to our MOV dataset. Both datasets in their experiment showed results almost equal for AE as for Rabin with target chunk sizes of 4 to 26 kB for TAR, and 200 to 2500 B for MOV, respectively (values are read off plots and hence only approximate).

The authors of FastCDC claim to achieve "nearly the same" deduplication ratio as Rabin-based approaches [9]. They base this claim on an experimental evaluation

that was performed on multiple datasets, three of those very similar to our TAR, LNX, and WEB datasets, respectively. The authors made this evaluation with target chunk sizes of 4, 8, and 16 kB. Although those values are below the range that we chose, by extrapolating the given results, we can approximately conduct an expected 31% in deduplication ratio for the TAR dataset, 95% for the LNX dataset and 91% for the WEB dataset, both at 32 KiB (similarly for Rabin and FastCDC).

4.2.3. Average Chunk Size and Chunk-Size Distribution

The target chunk size is the expected average chunk size, by definition. Yet, the literature has often shown that this is not true most of the time and depending on the dataset [8, 10]. In a simulation on randomly generated test data, we conducted following deviations from the target chunk size of 256 KiB: Rabin +28%, Buzhash 0%, FastCDC +18%, and AE 0%. These values remained stable with large data, different random data, and different target chunk sizes. Based on these results, we expect the average chunk size to lean towards those deviations, the higher the binary entropy in a dataset. For more ordered or correlated datasets on the other hand, we see a potential for recurring byte patterns which could pull the average chunk in a specific direction.

The authors of FastCDC, along with the deduplication ratios, stated the average chunk sizes that turned out with their analyses of different chunk sizes and different datasets on FastCDC and Rabin [10]. In every setting, the results show average chunk sizes varying from −51% to +60% deviation from the target chunk size with no observable pattern, i.e., the deviations do not seem to correlate with the target chunk size, and even between datasets, the results seem arbitrary. However, interestingly, the results of Rabin and FastCDC were almost always very similar to one another. We are not aware of any literature revealing similar statistics for Buzhash or AE. Finally, for FSC, obviously the average chunk size will match the target chunk size almost exactly[2].

Further, the paper on AE suggests a high chunk size variance in Rabin, and a great improvement upon that in AE [8], i.e., a reduced chunk-size variance. The same paper suggests a heavy-tailed distribution of chunk sizes for AE, as well as for Rabin. Contrastingly, by the means of normalized chunking, FastCDC is supposed (and expected) to produce a rather normal distribution. However, we were not able to find literature on its chunk size variance. We here again have no literature on the behavior of Buzhash. Intuitively, however, because it is conceptually similar to Rabin, we would set similar expectations. Finally, FSC is obviously not expected to produce any variance at all. An exception is to be made for the last chunk in every chunked file, which will have sizes lower than or equal the target chunk size.

[2] As with every chunking algorithm, the last chunk of every file is determined by the end of the same file. Therefore, its size is always less than or equal to what the algorithm would have decided otherwise and is likely to pull down the average chunk size.

4.3. Experimental Evaluation

In the following, we present the results of our experimental analysis concerning computational efficiency, deduplication performance, chunk size variance and average chunk size.

4.3.1. Computational Efficiency

The results of the computational efficiency of all five algorithms in comparison can be seen in Table 2. This table includes the speed, as well as the processed bytes, both which can be read as the inverse of each other. Further, it includes the the total allocated bytes, as well as the number of memory allocations which led to it.

It can be seen that FSC is by far the best performing chunking algorithm when it comes to speed and also that Rabin is significantly slower than the other more modern approaches. This is expected behavior, as outlined in the FastCDC paper, where it was said that FastCDC is about 10× faster than Rabin and about 3× faster than AE [9]. This was not exactly the case in our experiment, but that can have many reasons, e.g., the implementations of the algorithms used. FastCDC also performed extremely memory-efficient in comparison to every other algorithm. Buzhash also showed to be very fast. This was interesting to see, as the literature did not convey any information about its performance.

On memory consumption, we can see that AE did not, as anticipated, perform better than Rabin. Instead, it performed 3× worse than any other algorithm. It is likely that this does not say anything about the AE algorithm, but rather about our implementation of AE.

Algorithm	**Speed**	**Processed**	**Alloc. Bytes**	**Mem. Alloc.**
FSC	37.53 s/op	2,793.71 MB/s	104.86 MB/op	405 allocs/op
Rabin	610.60 s/op	171.73 MB/s	108.71 MB/op	968 allocs/op
Buzhash	103.71 s/op	1,038.99 MB/s	106.54 MB/op	421 allocs/op
FastCDC	66.48 s/op	1,577.21 MB/s	1.05 MB/op	3 allocs/op
AE	249.19 s/op	420.80 MB/s	318.19 MB/op	802 allocs/op

Table 2: Benchmark for speed and memory utilization of all chunking algorithms in comparison on a generated test file of 100MB and a target chunk size of 256 kB. An operation ("op") equals the process of splitting the test file into its chunks.

4.3.2. Deduplication Performance

Figure 12 shows the results for the deduplication performance on every dataset. The first important observation is that there is not one single algorithm that outperforms all the others with every dataset. We can also observe that, while the deduplication ratios decline with higher chunk sizes (as was expected), the trend regarding an algorithm's comparative superiority is not really affected by it. This is to say that a chunking algorithm A does not perform better than a chunking algorithm B with chunk sizes of α, and then suddenly A does worse than B with chunk sizes of β. The described scenario can be observed in some of the plots though, but only if two algorithms perform very similar nevertheless. We consider those occurrences as natural fluctuations that arise with the arbitrariness of the data in the datasets. However, the relative differences shrink with higher chunk sizes, i.e., they converge to a common value (and eventually to 0). This is best visible in Figure 12e and expected with the exponential decline of deduplication ratio with higher chunk sizes.

Concluding from that, we can use the result that Buzhash produced for 256 KiB, compare it with the performances of the other algorithms, and based off that, conduct which chunking algorithms performs the best on each dataset. Because of smaller target chunk sizes, we cannot make any statements about Buzhash's deduplication performance for the datasets of PNG, JPEG, and PDF.

As expected, FSC returned the worst deduplication ratios. It is interesting to see how even in the datasets with the least data correlation (cf. Figures 12e & 12f), where the deduplication success is generally expected to be close to zero, CDC algorithms still perform noticeably better than FSC. Even in MOV (Figure 12h), the dataset with the worst deduplication ratio, FSC is the only algorithm to score exactly 0% across all target chunk sizes.

We have not found a single algorithm to be significantly superior to any others on every dataset. However, on the TAR and MOV dataset, Buzhash seems to beat any other chunking algorithm by a significant margin. Unfortunately, we are missing data on Buzhash's performance on some datasets. Anyhow, we would expect it to behave similar in comparison to the other algorithms as on the other datasets (i.e., similar or better than Rabin and FastCDC).

Further, Rabin and FastCDC performed very solidly in the evaluation on every dataset. While Rabin is still a little ahead on some datasets, FastCDC often managed to achieve very similar results.

AE, while it still performs significantly better than FSC, underperforms compared to the other CDC algorithms in most datasets. This can be observed most strongly on the LNX dataset (Figure 12c).

In conclusion, the CDC algorithms performed quite similar in comparison most of the time, with some occasionally showing superiority on some datasets (Buzhash in TAR and MOV, and Rabin in WEB). Minor performance differences showed to be more extreme with smaller chunk sizes but even out at 256 KiB (cf. Figure 12b).

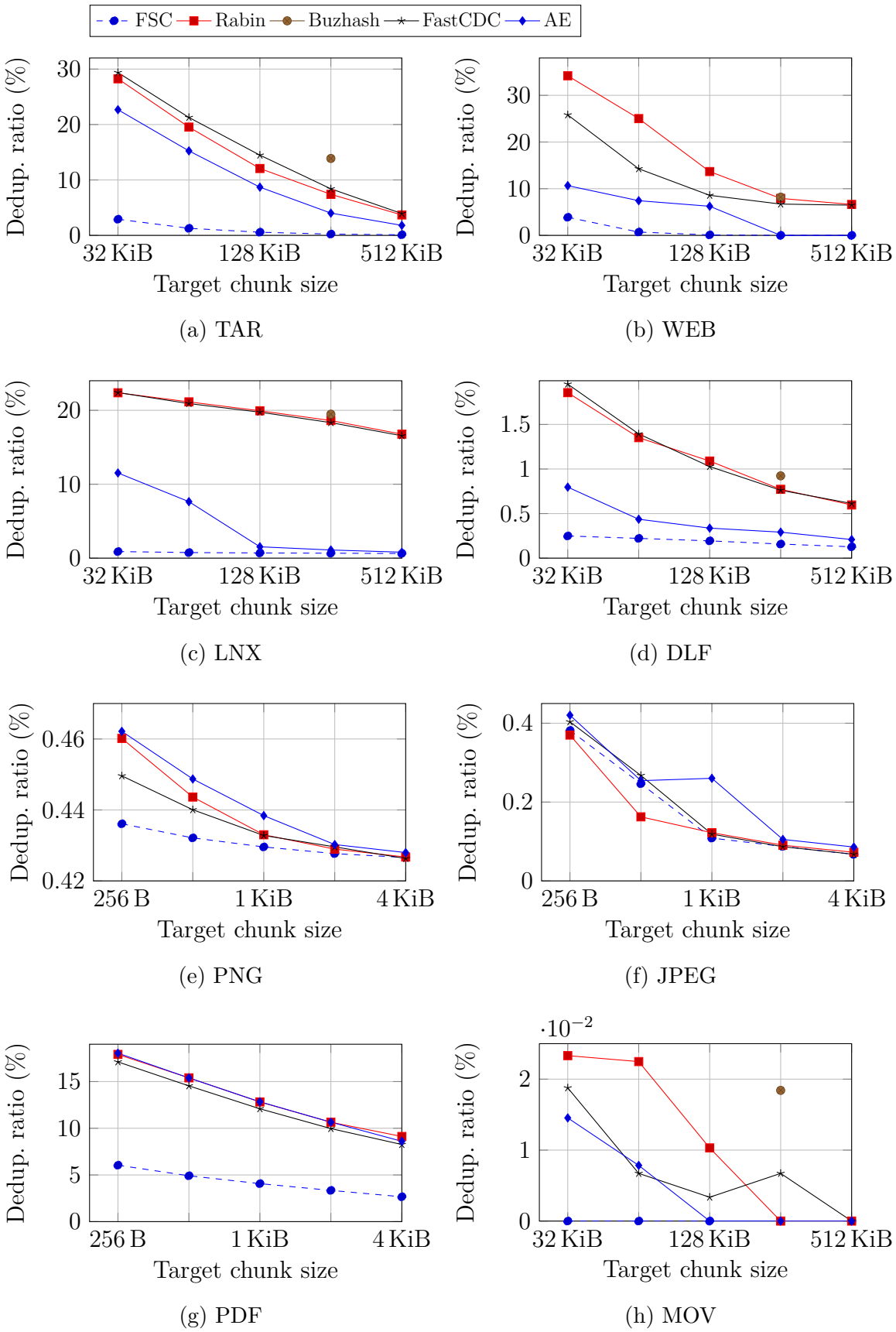


Figure 12: Results of deduplication performance on different datasets.

4.3.3. Average Chunk Size and Chunk-Size Distribution

We have listed the average chunk sizes in each setting (i.e., for each algorithm, datasets, and target chunk size configuration) in Table 4.3.3. This revealed a few things to us.

The most extreme values in this table are found for Buzhash across the TAR and WEB datasets with negative deviations up to 55%. We can derive from this that Buzhash produced much smaller chunk sizes for those datasets than its competing algorithms, and generally the smallest chunk sizes on every dataset on which it has been evaluated on. This potentially explains its superior deduplication ratio. Likewise, Rabin produced significantly smaller chunk sizes on the TAR and WEB dataset. However, it produced the highest average chunk size on the PNG, JPEG, and PDF dataset. It seems that Rabin's average chunk size correlates strongly with the degree of correlation in a dataset. FastCDC and AE show a positive deviation from the target chunk size on most experiments, which suggests a disadvantage in deduplication performance compared to Rabin and Buzhash. On some datasets this discrepancy caused the average chunk produced by FastCDC and AE to be almost two times as big as those created by their competitors.

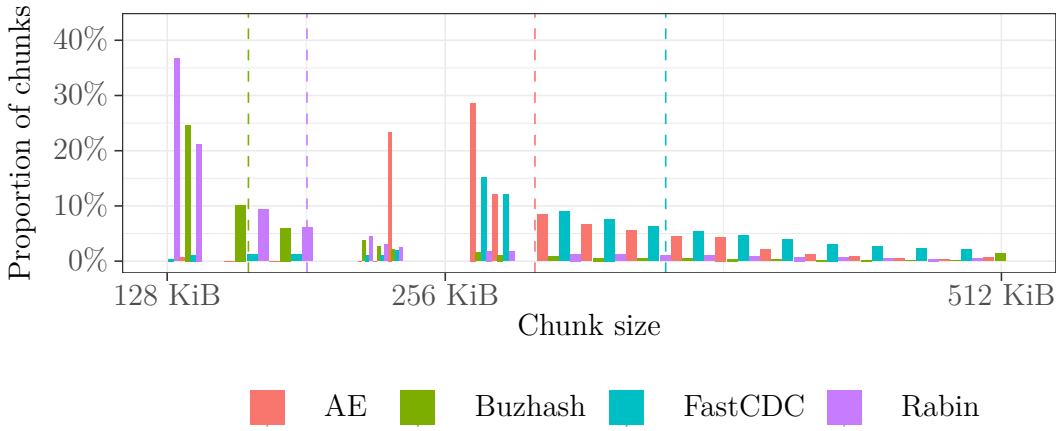
Finally, we can observe the deviation in almost every dataset growing or getting more extreme with higher target chunk sizes.

To further analyze and interpret the effects of chunk-size variance, we have looked at the distribution of chunk sizes. The general pattern in each algorithm seemed to be the same across datasets. However, we did notice differences between more correlated and less correlated datasets. To illustrate these differences and also to show how the algorithms' chunk-size distributions generally look, we depict those results for the TAR dataset (representative for a dataset with very high correlation) in Figure 13a and for the PNG dataset (representative for a dataset with very low correlation) in Figure 13b. The complete list can be found in Appendix A. FastCDC has expectedly shown the characteristics of a normal distribution. Rabin, Buzhash, and AE at least most of the time follow a heavy-tailed distribution. In Rabin and Buzhash, we see the median for chunk sizes forming in a range starting at the minimum chunk size, and then continuously decreasing.

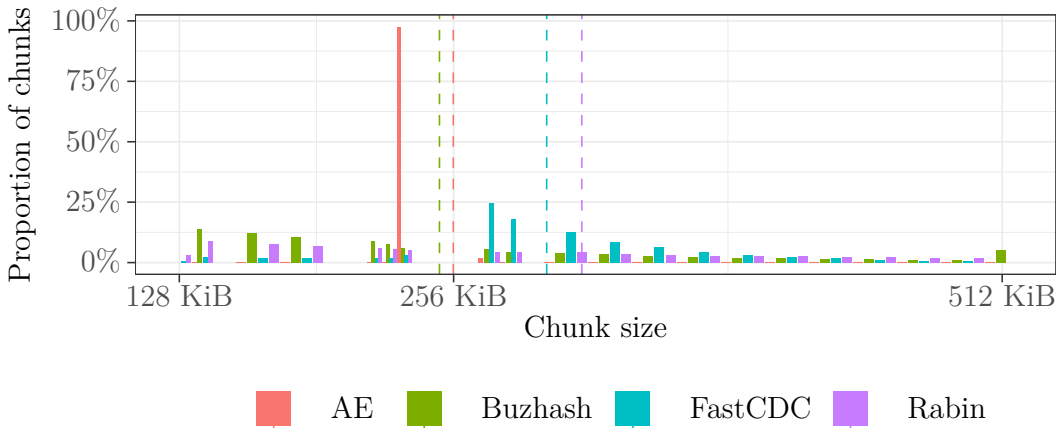
AE, however, appears less consistent throughout the datasets regarding its distribution shape. In many plots (WEB, LNX, DLF, and MOV), almost all chunks with AE are created within a very short range of sizes around the target chunk size. That is, the chunk size variance for AE on these datasets is almost minimal. We do not understand why the behavior is like this for those datasets, but then very different in other datasets. For example, we would have expected the distribution in the MOV dataset to be very similar to that of the PNG dataset, as both datasets are suspected to have similar binary entropy.

Data	Alg.	Target Chunk Sizes				
		32 kB	**64 kB**	**128 kB**	**256 kB**	**512 kB**
TAR	FSC	32 kB	64 kB	128 kB	256 kB	511 kB
	Rabin	31 kB (-3%)	57 kB (-12%)	105 kB (-22%)	192 kB (-33%)	354 kB (-45%)
	Buzhash	-	-	-	165 kB (-55%)	-
	FastCDC	45 kB (+28%)	89 kB (+28%)	177 kB (+28%)	357 kB (+28%)	708 kB (+28%)
	AE	36 kB (+11%)	73 kB (+12%)	147 kB (+13%)	297 kB (+14%)	599 kB (+15%)
WEB	FSC	32 kB	64 kB	128 kB	255 kB	509 kB (-1%)
	Rabin	33 kB (+4%)	60 kB (-8%)	109 kB (-18%)	206 kB (-25%)	393 kB (-30%)
	Buzhash	-	-	-	189 kB (-35%)	-
	FastCDC	39 kB (+18%)	77 kB (+17%)	154 kB (+17%)	310 kB (+17%)	614 kB (+17%)
	AE	32 kB (+1%)	64 kB	129 kB (+1%)	256 kB	513 kB
LNX	FSC	32 kB	64 kB	128 kB	256 kB	512 kB
	Rabin	32 kB	64 kB	127 kB (-1%)	250 kB (-2%)	496 kB (-3%)
	Buzhash	-	-	-	245 kB (-5%)	-
	FastCDC	38 kB (+15%)	75 kB (+15%)	150 kB (+15%)	300 kB (+15%)	601 kB (+15%)
	AE	32 kB (+1%)	64 kB	128 kB	256 kB	511 kB
DLF	FSC	32 kB	64 kB	127 kB (-1%)	252 kB (-2%)	494 kB (-4%)
	Rabin	32 kB	64 kB	126 kB (-2%)	245 kB (-5%)	469 kB (-9%)
	Buzhash	-	-	-	239 kB (-7%)	-
	FastCDC	37 kB (+15%)	75 kB (+15%)	150 kB (+15%)	299 kB (+15%)	597 kB (+14%)
	AE	32 kB	64 kB	127 kB (-1%)	251 kB (-2%)	493 kB (-4%)
MOV	FSC	32 kB	64 kB	128 kB	256 kB	512 kB
	Rabin	40 kB (+21%)	80 kB (+20%)	159 kB (+19%)	316 kB (+19%)	619 kB (+17%)
	Buzhash	-	-	-	249 kB (-3%)	-
	FastCDC	36 kB (+11%)	72 kB (+11%)	143 kB (+11%)	286 kB (+11%)	572 kB (+11%)
	AE	32 kB (-1%)	63 kB (-1%)	126 kB (-1%)	252 kB (-1%)	505 kB (-1%)
		256 B	**512 B**	**1 kB**	**2 kB**	**4 kB**
PNG	FSC	255 B	511 B	1023 B	2045 B	4085 B
	Rabin	325 B (+21%)	651 B (+21%)	1303 B (+21%)	2603 B (+21%)	5 kB (+21%)
	Buzhash	-	-	-	-	-
	FastCDC	299 B (+14%)	598 B (+14%)	1196 B (+14%)	2392 B (+14%)	5 kB (+14%)
	AE	340 B (+25%)	657B (+22%)	1238 B (+17%)	2307 B (+11%)	4 kB (+6%)
JPEG	FSC	255 B	511 B	1021 B	2039 B	4063 B (-1%)
	Rabin	326 B (+22%)	655 B (+22%)	1307 B (+22%)	2607 B (+21%)	5 kB (+21%)
	Buzhash	-	-	-	-	-
	FastCDC	300 B (+15%)	600 B (+15%)	1199 B (+15%)	2393 B (+14%)	5 kB (+14%)
	AE	330B (+23%)	633 B (+19%)	1195 B (+14%)	2256 B (+9%)	4 kB (+5%)
PDF	FSC	255 B	511 B	1023 B	2047 B	4093 B
	Rabin	309 B (+17%)	618 B (+17%)	1236 B (+17%)	2459 B (+17%)	5 kB (+16%)
	Buzhash	-	-	-	-	-
	FastCDC	314 B (+18%)	627 B (+18%)	1259 B (+19%)	2527 B (+19%)	5 kB (+19%)
	AE	318 B (+20%)	623 B (+18%)	1189 B (+14%)	2298 B (+11%)	4 kB (+8%)

Table 3: Real average chunk sizes produced during our experiments on different datasets and their deviation from the target chunk sizes.



(a) TAR



(b) MOV

Figure 13: Chunk-size distribution of TAR and MOV with a target chunk size of 256 KiB. The dashed lines mark the mean values in each algorithm. For better readability, FSC is not shown.

4.4. Conclusion

We have compared five implementations of different chunking algorithm on their performance on many realistic datasets and popular file types. The results revealed to us, among other things, the speed at which each algorithm performs, the deduplication ratio that each algorithm is able to achieve, and how this depends on the dataset. Most interestingly perhaps is the general disparity in deduplication ratio between FSC and CDC algorithms. This disparity seems negligible on media files, like PNG, JPEG, MP4, and MKV, files with internal compression and therefore high binary entropy. We can therefore infer the same with compressed archive files (e.g., ZIP), executables, and other formats. This makes a strong case for FSC as the appropriate chunking strategy in IPFS, where the computational speed is still 77% better than the most efficient CDC algorithm, which is FastCDC. However, we have also seen the potential of deduplication with CDC on datasets with high correlation. It gets lower with higher chunk sizes but is still significant at 256 KiB.

We conclude that CDC algorithms are worth the computational overhead for datasets with high correlation, especially of uncompressed file types. However, as the result for the DLF dataset (an organic mix of different file types) in Figure 12d suggests, this does not necessarily reflect the character of the set of files that users normally download from the web (however, it excludes actual websites).

Generally, considering computation speed and deduplication performance, FastCDC and Buzhash seem to be the best options for CDC. From those options, FastCDC is the computationally most efficient and Buzhash deduplication-wise the slightly stronger choice. However, it is important to not neglect the fact that Buzhash generally produced much smaller chunks than FastCDC did. Therefore, the argument could be made that at similar average chunk sizes, the deduplication ratio achieved by FastCDC might have been equal to or better than that of Buzhash.

5. Empirical Analysis on IPFS

The previous chapter has shown that the deduplication ratio depends very much on the data type and the correlation within a dataset. It could be shown that CDC algorithms can be used very effectively on some datasets to eliminate duplicated content. However, on compressed file types like JPEG, the result would not be very different from that of the primitive FSC approach. Even worse, the CDC algorithms in those cases perform much slower and with a higher chunk size variance than FSC. This is why it is important to additionally conduct an empirical analysis for the data on IPFS. In this analysis, we compile datasets reflective of the files downloaded by individual users over a range of time on IPFS. On this basis, we again conduct the deduplication ratios that could be achieved with every chunking algorithm.

5.1. Methodology and Dataset

The methodology can be conceptually described as a three-phase process. In the first phase, we collect traces by monitoring and recording requests made inside the IPFS network over a sufficient period of time[3]. To this end, the software of the work in [11] is leveraged. Secondly, the recorded traces are used to replicate the relevant parts of the system locally, essentially mirroring this part of the IPFS network for local access. That includes, among other things, actually downloading the file contents that have been requested. This is the pre-processing phase that enables the final phase to operate fully offline and deterministically. Finally, in the last phase, the traces in combination with the locally mirrored parts of IPFS are used to simulate the deduplication results with each chunking algorithm.

5.1.1. Collecting Traces

As mentioned before, this part of the methodology entirely leverages the work that has been done in [11]. For their study, the authors activated multiple nodes running a modified version of the IPFS client. Those nodes would not initiate any requests, but passively listen for requests from their peers and log the event to a JSON file. A recorded request is what will be called a “message” in the remainder. It contains a timestamp, the ID of the peer who issued the request, the CID of the requested block, along with some other metadata. Not every message in our traces will be evaluated in our analysis, as some represent irrelevant connection events, or canceled or duplicate requests. The same software used for the case study in [11] was used here again to produce recent traces. It is important to have very recent traces because data in IPFS is ephemeral and that can lead to failed download attempts in the subsequent phase of the process. While the risk of failed download attempts can never be eliminated fully, the goal is to keep the bias against short-lived data as minimal as possible.

[3]Due to limited time and slow processing of the traces, we were only able to use about eight hours of recordings.

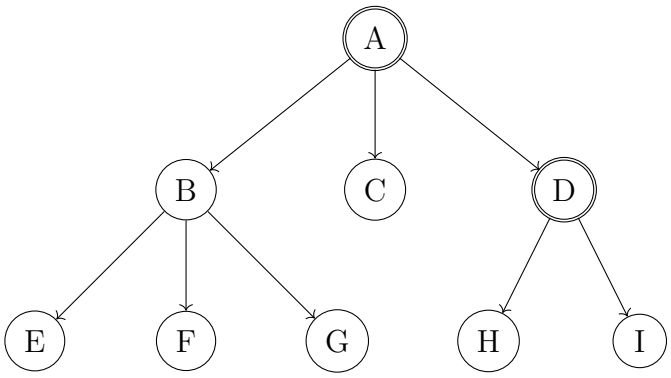


Figure 14: Conceptual visualisation of the DAG structure descending from an IPFS block (A), where double bordered nodes mark a directory, and single bordered nodes mark a file or chunk.

In the remainder, we let T denote the list of traces, and $m \in T$ have the following structure.

```
struct Message {
    int ts;              // Timestamp.
    string cid;          // Requested CID.
    string peer;         // Peer ID.
};
```

This is only an abstracted and simplified view of the traces produced in the JSON files, but it is sufficient to follow the descriptions of the algorithms in the subsequent sections.

5.1.2. Replicating IPFS Data Locally

The traces tell us who requested which CID when, but they do not convey anything about the contents of the CID, not even if it is a file or directory. In order to see the effect of a chunking algorithm, the actual binary content behind a CID must be retrieved. Furthermore, since the CID could be the root of a directory structure or a bigger file, and also the nodes of the tree can be interconnected with other files, it is crucial to understand the DAG structure and relationships of any given CID.

Figure 14 shows an example of a DAG, recursively retrieved from a block A. In this example, A is a directory and has two files (B and C) and another directory (D), which again includes two files (H and I). B is a bigger file. So the block with that CID does not contain the raw content of the file but references to three further files (referenced by E, F, and G), i.e., the chunks of the file. The whole file would be assembled by the ordered concatenation of all chunks ($E \parallel F \parallel G$). The file B could be even bigger to the point where it cannot carry the pointers to every chunk. In that case, the chunks

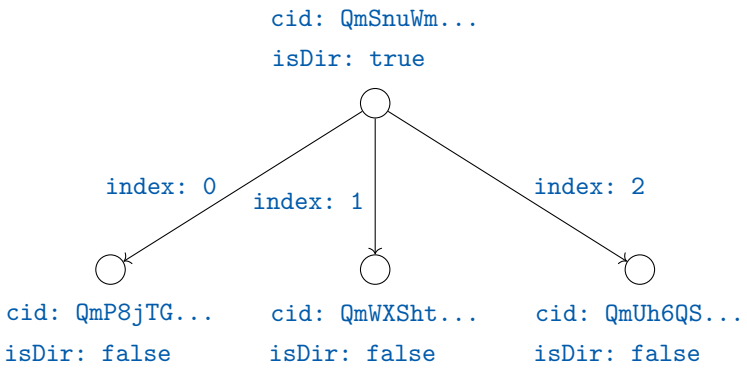


Figure 15: Exemplary view of the data model in the database, showing a block with links to three other blocks.

of B would become files that contain new chunks. This can repeat itself over many hierarchies. The restoration of the root file B then requires the concatenation of all chunks (the leafs of the graph descending from B) in the right order. Note how there is no explicit distinction between a file and a chunk. C, H, and I are the roots of a file, but also contain the raw content of it at the same time. Also note how in this conceptual drawing, e.g., E, is a chunk of B, but it could also be the chunk of many other files or even directories at the same time, and the same of course applies for files and directories. In fact, this is what enables deduplication.

Since not only the file contents, but also the references between IPFS blocks (i.e., directories, files, chunks) are ephemeral, it is not sufficient to only download the raw contents, but also the references that lead up to it.

In our implementation, we store the blocks and relationships that are relevant for the analysis as nodes and edges in a RedisGraph database. To this end, every requested block (as can be derived from the CIDs in the traces) and all its linked blocks need to be fetched from IPFS recursively. Eventually, the leafs of the descending DAG contain the raw bytes that get saved to disk, each leaf (block) as an individual file. This algorithm is described more formally (but simplified) in Algorithm 1. The functions CreateNode and CreateEdge will only create the respective entities in the database if they do not already exist. This means that every CID exists uniquely in the database, as it is supposed to, and that new traces whose DAG intersects with already known blocks can seamlessly be connected to existing structures.

Figure 15 exemplifies the database model. The colored labels comprise the properties/metadata that were set on the nodes and edges. Every node in the graph has a *cid*, and every edge/arc in the graph has a numeric *index*, indicating the order. In addition, we note whether a node represents a directory root or not (*isDir*), which is evaluated later in Algorithm 2. We do not distinguish between files/ file roots and chunks here.

After the relevant parts of IPFS have been replicated locally, the simulation can finally be performed offline and deterministically.

Algorithm 1 Process Traces

Recursive fetching of all CIDs present in the traces, initiated at line 26–28.

Function Parameters: CID of block to fetch *cid*; Parent node *p*; Index among parent references *i*

```
1: function DOWNLOAD(cid, p, i)
2:     if CIDISKNOWN(cid) then
3:         return                          ▷ Skip if CID already exists in our database.
4:     end if
5:     CREATENODE(cid)                     ▷ Create node in RedisGraph.
6:     if p ≠ null then
7:         CREATEEDGE(p, cid, i)           ▷ Create edge in RedisGraph from p to cid.
8:     end if
9:     if cid.type = Raw then              ▷ Check will not work on CIDv0, see line 20.
10:        raw ← GETRAW(cid)               ▷ Request raw bytes of the block.
11:        SAVE(cid, raw)                  ▷ Save bytes to disk in a new file.
12:    else
13:        raw, links ← GETDAG(cid)
14:        if links ≠ ∅ then               ▷ Returns links to children or...
15:            j ← 0
16:            for each link ∈ links do
17:                DOWNLOAD(link, cid, j)
18:                j ← j + 1
19:            end for
20:        else                            ▷ ...raw bytes (CIDv0 only).
21:            SAVE(cid, raw)
22:        end if
23:    end if
24: end function
25:
26: for each t ∈ T do
27:    DOWNLOAD(t.cid, null, 0)
28: end for
```

5.1.3. Simulation

The goal of this phase of the process is to determine and compare the deduplication ratio for the requested files had they been chunked using different algorithms before uploading to IPFS. To this end, we again go over the traces in chronological order and hold state for each peer's requested and downloaded files. By *file*, we refer to the original, complete files, i.e., the sum of its chunks.

The algorithm itself has a *preprocessing* and a *processing* step. In the *Preprocessing* step (see Algorithm 2), we iterate over the same traces that we processed in the previous step, again chronologically. During this step, we prepare three things that the subsequent *processing* step will utilize and read from:

- A folder containing every requested file (i.e., whole files, assembled from their chunks); we name the files by the SHA-1 hash of their binary content,
- and a Redis database mapping peer IDs to their set of requested files, represented by the files' fingerprints.

Furthermore, we fill another Redis database where we map peer IDs to their set of directly requested CIDs. However, this is only used during the *preprocessing* itself to check if we have already encountered this peer requesting this CID, in which case we skip it. This is a measure we take due to the many duplicates in our traces that happen due to periodical retries on the client-side when retrievals fail.

The whole process is described in more detail in Algorithm 2. In order to better follow along with this and subsequent algorithms, we want to introduce the following letters that we use to represent certain concepts. We refer to

- C as the set of CIDs,
- P as the set of peer IDs,
- D as the set of binary data objects (i.e., $\{0,1\}^*$),
- and F as the set of whole files.

The algorithm in 2 reads from and writes to three initially empty mappings, which we disclose in the following:

- $\mathcal{F} : C \to D^*$ represents a whole file, indexed by its root CID.
- $\mathcal{F}_r : P \to D^*$ represents each peer's requested whole files (as binary data objects).
- $\mathcal{C}_r : P \to C^*$ represents each peer's directly requested CIDs.

Note that the mappings of $\mathcal{F}_r$ and $\mathcal{C}_r$ represent what we previously introduced as instances of Redis key-value databases. Basically, what happens in the *preprocessing* is that in each iteration of the messages in our traces, we look up the CID in our graph database and from this node we try to retrieve all files to which this CID points. At

this point, we fill the mapping of $\mathcal{F}$. The node (i.e., the CID) itself could be a file and holds all its binary data, or it could be a file represented by a set of chunks, or it could be a directory, even a directory of directories, pointing to a set of files. Moreover, a file can lead over multiple hierarchies of pointers until it reaches the leafs which finally represent the chunks of the file. This is common with big files.

The first case (the node itself is a file) is determined when the respective node has no children it points to *and* we can find its contents on our disk (otherwise it is a broken reference and we skip it). Note that we cannot distinguish between a chunk and a potentially small file that the IPFS user was requesting. As for the other cases, we can see that the process demands a recursive approach. This particular function (to retrieve the files from a single CID) is shown in Algorithm 3. During the iterations of this function, the mapping $\mathcal{F}$ is generated. For every new file, $\mathcal{F}$ is extended by another key: the CID that represents the file. This CID then becomes the "CID of top-level file" f in further recursive calls to the function. The CID then maps to an ordered list of data objects D^*, i.e., the actually binary contents of the file's chunks. This result is used back in line 8 of Algorithm 2 where the data objects are reassembled, yielding the whole file, and then appended to $\mathcal{F}_r$.

Algorithm 2 Preprocessing

Input: Traces T
Output: Mapping of peers to their requested file contents R

1: **for each** $t \in T$ **do**
2: **if** $t.cid \in \mathcal{C}[t.peer]$ **then**
3: **continue**
4: **end if**
5: $\mathcal{F} : C \to D^* \leftarrow \emptyset$
6: $\mathcal{C}_r[t.peer] \leftarrow \mathcal{C}_r[t.peer] \cup t.cid$
7: RetrieveFiles($null$, $t.cid$)
8: $\mathcal{F}_r[t.peer] \leftarrow \mathcal{F}_r[t.peer] \cup \{\{f_1 \parallel f_2 \parallel \cdots \parallel f_n\} \mid F_i \in \mathcal{F}\}$
9: **end for**

The *preprocessing* phase gave us a chronological sequence of requested files per peer, formally $\mathcal{F}_r$. The idea is that each file/chunk that has previously already been requested by a peer, has also been downloaded by this peer, and thus does not have to be fetched again. By chunking every requested file and then simulating the sequences of requests for each peer, we can obtain the ratio of chunks (i.e., the sum of their sizes in bytes) that each peer actually has to download d_p to the total size of the requested files r_p. Accumulated over all peers, the result is the total deduplication ratio that could have been achieved with the respective chunking algorithm for the observed traces, $1 - \frac{d}{r}$, with $d = \sum_{p \in P} d_p$ and $r = \sum_{p \in P} r_p$. We execute this algorithm (described in more detail, however simplified, in Algorithm 4) once with each chunking algorithm and finally compare the resulting deduplication ratios.

Algorithm 3 RetrieveFiles(f, cid)

Input: CID of top-level file f; CID of current block cid

```
1: b_r ← GetNode(cid)
2: B ← GetChildren(b_r)                    ▷ Ordered by arcs' index property.
3: if not b_r.isDir then
4:     if f = null then
5:         f ← b_r.cid                     ▷ New file root detected.
6:         F[f] ← ∅
7:     end if
8:     if B = ∅ then
9:         F[f] ← F[f] ∪ b_r.cid           ▷ Current block is considered a chunk.
10:    end if
11: end if
12: for each b ∈ B do
13:    RetrieveFiles(f, b.cid)
14: end for
```

Algorithm 4 Simulation: Processing

```
1: r ← 0                                   ▷ Sum of requested bytes.
2: d ← 0                                   ▷ Sum of downloaded bytes.
3: for each p → F ∈ F_r do                 ▷ For each peer p and their set of requested files F.
4:     D ← ∅                               ▷ Data this peer has already downloaded.
5:     r_p ← 0
6:     d_p ← 0
7:     for each f ∈ F do
8:         r_p ← r_p + |f|
9:         for each f_i ∈ Split(f) do      ▷ f_i is a chunk of f.
10:            if f_i ∉ D then
11:                d_p ← d_p + |f_i]
12:                D ← D ∪ f_i
13:            end if
14:        end for
15:    end for
16:    r ← r + r_p
17:    d ← d + d_p
18: end for
```

5.2. Results

From the first step of our methodology, which was to collect the traces, we obtained one file per hour of a single day, each containing the traces recorded during that hour in chronological order.

A single trace file contains about 12 million objects. Those include canceled and also duplicated requests (i.e., retries), which are then ignored during the processing in our second step (replicating IPFS data locally). However, considering that every object requires fetching an arbitrary number of data blocks from IPFS, there are certain bottlenecks which slow down the progression of the traces. In order to increase the trace throughput, we ran eight instances of our program, each processing another trace file.

We let this run for about one month. Throughout this month, several changes were made to improve the efficiency of this process. This includes a RAM upgrade to our server, tuning of parallelization and timeout settings, and other performance tweaks in our code. Note therefore that the trace message throughput was much higher towards the end of the month than it was in the beginning.

When we interrupted the processing of the trace files, our program had gone over approximately 2.4 million traces in total. The number of traces is nearly equally distributed on each of the eight trace files on which we had run the program. That is, from eight hours of recordings on the IPFS network, we processed only a tiny fraction from within each hour. Assuming a uniform distribution of recorded messages over time, we approximate this to 1 minute per trace file, i.e., 8 minutes in total. Since network connections are inherently unreliable and data on IPFS is ephemeral, we were not successful retrieving all necessary data to replicate the data structure (i.e., the nodes, their relationships, and data) associated with every CID in the traces. An encountered CID can lead over multiple hierarchies of further nodes before reaching its leafs, i.e., the actual chunk data. Furthermore, a single hierarchy can reference hundreds of nodes. It is therefore possible that retrieving a single CID must be canceled in the middle and its retrieval will be left incomplete. In fact, the likelihood for this event increases rapidly the larger the content represented by the CID. However, we still create the data structure that leads up to this point. That is, we still consider the files that could be retrieved successfully. In our logs, we count around 1 million error cases that caused a retrieval process, be it of a single chunk or a block pointing to a new set of blocks, to stop and, ultimately, corrupt our results.

Finally, we were able to replicate the distributed IPFS data structure in a graph with 12,491,375 nodes, of which 1,701,146 are leaf nodes. The leaf nodes, due to the errors we just explained, do not necessarily represent file or chunk data. In the end, we were able to download a total of 773,068 data blocks (files or chunks) amounting to 127 GB.

Further on, the preprocessing step of the simulation yielded a total of 272,554 restored files. This number originates from a total of 1,049 peers that we recorded to make a request for 366,012 CIDs of which 273,555 CIDs were unique (counting only the direct requests). The number of unique CIDs is slightly above the number of restored files,

which we account to the natural error-proneness in the processing steps. We derive from that, that there were no big folder structures, but rather one CID accounted for one file.

Aggregated over every peer, we simulated a total of 254 GB of file downloads with every chunking algorithm, respectively. The result was a deduplication ratio of exactly 0%, i.e., not a single deduplicated byte with every chunking algorithm.

In order to better understand this extreme result (not a single duplicated chunk), we further conducted a few analyses, whose results we present in the following.

Recall that deduplication is measured from the perspective of a peer. This means that deduplication success depends on redundancies only found in the set of requested files of a peer. The chance of finding redundancies naturally rises with the size of this dataset. Furthermore, we know that redundancies are much more likely to occur with correlated data or data with low binary entropy (e.g., text-based content).

In order to gain more insights about the set of files, we have used Linux' standard utility `file` (version 5.39) to inspect the mime type as well as the encoding of every file. We give an overview of the results in Table 4. For around 87% of files, the tool did not recognize the encoding of a charset (they are "binary"). We assume high binary entropy for these files, as they usually are created with a mechanism of internal compression (e.g., JPEG). Most files (78%), as the table reveals, were classified with the MIME-type "application/octet-stream". These can be interpreted as cases where a MIME-type could not be recognized from the file contents. Although this can have multiple reasons, we think that this is most of the time caused by an incomplete file, i.e., cases where the requested CID pointed only to a subfile or chunk. In the cases where the original file would be binary encoded (instead of, e.g., ASCII or UTF-8) and the header of the file was not present, a classification of the type "application/octet-stream" is exactly what we would expect.

Occurrences (%)	**MIME-Type**	**Charset**
213,355 (78%)	application/octet-stream	binary
27,025 (10%)	application/json	us-ascii
9,624 (3.5%)	image/png	binary
4,550 (1.7%)	image/jpeg	binary
4,283 (1.6%)	application/pdf	binary
3,413 (1.3%)	text/plain	us-ascii
3,388 (1.2%)	application/json	utf-8
3,363 (1.2%)	application/x-dosexec	binary

Table 4: List of a file's MIME-type and charset within the set of restored files which occurred for $\geq 1\%$ of the files.

Furthermore, we analyzed the distribution of file sizes and found that 252,772 (93%) of the files have a size ≤ 256 KiB (the default chunk size for data blocks in IPFS). This

is another strong indication for us that most of the requested and restored files are not whole files but rather single chunks.

Those findings generally do not advocate for a good deduplication ratio. However, on the other hand, around 11% of files were classified as JSON files, where correlations inside and across files seem more likely. Moreover, while most restored files turn out to be small, big files make their occurrence too: 81 files have a size $\geq 100\,\text{MB}$, the largest file among the set even reaching 5.9 GB.

However, larger files would always be recognized with the charset "binary". As we conducted an analysis of the sizes of all files with a charset unequal to "binary" (mostly JSON), we found that they were mostly only a few bytes long with only a few exceptions which exceeded 1 kB. Actually, 85% of the findings only have 3 bytes or less. In fact, when looking at the file size instead of the file amount, binary files account for 99% of data in our dataset; the files that were specified as "application/octet-stream" make up 84% of the data.

Furthermore, a peer can have requested multiple CIDs, which would also increase the size of the dataset per user and therefore the likelihood of redundancies (although not in the same way[4]). In the accumulation of files requested by single peers, we have found many cases of large datasets with sizes up to 12 GB.

We made another analysis to understand how many CIDs (and therefore distinct files) were requested by how many peers. The results are depicted in Figure 16. Contrary to our initial expectation, which was that due to the small time frame of IPFS monitoring most peers would only map to a single CID request, we can see that in fact many peers (88%) requested two or more CIDs. At the most, a peer was recorded to have requested 13,224 distinct CIDs.

Finally, we have conducted an analysis for deduplication performance with every chunking algorithm on the complete set of files as a whole, i.e., on 127 GB. Furthermore, we have created an uncompressed TAR file of the dataset, such that the analysis would only perform on a single file. We did this analysis to remove factors that were expected to have a negative effect on the deduplication performance, such as small file sizes and small datasets per peer. Moreover, in favor of the deduplication ratio, we have reduced the target chunk size to 4 KiB.

Despite these measures, the deduplication ratio was again 0% with every chunking algorithm. Given the large dataset and the small chunk sizes, we did expect at least some redundant chunks, disregarding the file type or entropy. However, we have calculated that the probability of this event (no unique chunks) in this setting and with maximum entropy content is in fact likely with probability almost 1 (see Appendix B).

[4]Deduplication success on a set of multiple small files is expected to be inferior to a single file of the same total size because, in a dataset with many files, chunk boundaries will have to be set based on the ending of a file rather than purely content-dependant (as for CDC).

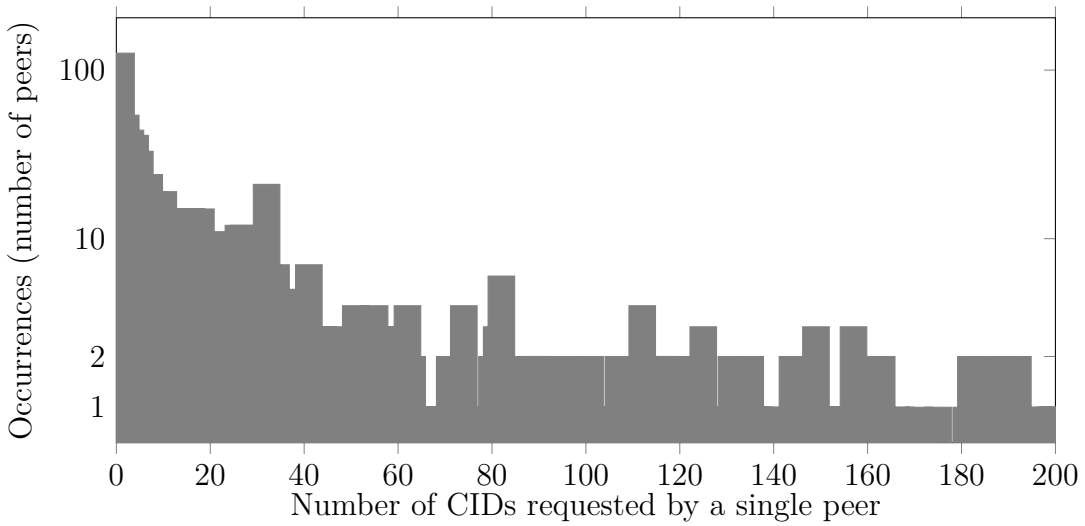


Figure 16: Distribution of the number of CIDs directly requested by single peers. The y-axis is logarithmic and the scale of the x-axis is limited to 200 for better readability.

6. Discussion of Results

The empirical analysis in Section 5 did not yield the expected results. According to this analysis, CDC would not support deduplication on IPFS at all. Concluding from that, the chunking algorithm with the best computational efficiency should be used, which is FSC. FSC, on further notes, also has the benefit of equal chunk sizes which also exactly meet the target chunk size (with exception of the last chunk in a file).

However, we have shown in Section 4 that CDC algorithms are capable of achieving very high deduplication ratios if applied on the right dataset. More precisely, these are datasets with high cross-file correlation and files with low binary entropy, i.e., preeminently files without internal compression. The question therefore is whether **(a)** the files exchanged on IPFS are mostly of that category that does not benefit from the deduplication efforts of specialized chunking algorithms (CDC), or **(b)** our methodology was insufficient to work out its potential effects on the larger scale.

The DLF dataset in our first analysis was supposed to mimic the dataset that reflects what a user is expected to download from IPFS. We have seen in the simulation of the chunking algorithms on this dataset that at a target chunk size of 256 KiB only less than 1% of redundant content could be eliminated. Therefore, the potential deduplication success in the empirical analysis was not expected to be very significant anyway. It can further be argued that the deduplication ratio would further decrease the smaller the dataset is. However, with the DLF dataset having a size of 13 GB, it is very comparable to the largest recorded dataset requested by a single peer, which was of 12 GB. Anyway, we would have most definitely expect to see some deduplication when we conducted the same simulation on the sum of all requested files in a single TAR of 127 GB file, and with a much smaller target chunk size of 4 KiB.

We know through our proof (outlined in Appendix B) that this is almost guaranteed for files with maximum binary entropy. It is nevertheless surprising to see a deduplication of exactly 0 because even in our analyses of the PNG, JPEG, and MOV dataset (consisting of file types which have internal compression) a deduplication ratio above 0 (at least for CDC) could be shown. This in turn suggests that PNG, JPEG, and MP4 and/or MKV files, despite their internal compression, do not nearly have maximum binary entropy. Recall that for at least 78% of the files in our empirical analysis we do not know the file type. Furthermore, for between 84–99%[5] of the data we do not know the internal byte structure and the repercussions it has on deduplication performance. We have also seen that most non-binary in our dataset are very small in file size to a point where CDC could not be applied effectively because the first and only chunk boundary would be defined by the file ending.

Hence, we conclude that indeed for the specific time period and this part of IPFS, CDC algorithms would not have added any benefit. On the contrary, it would have only added to the computation time in the chunking process as well as complexity due to unequal chunk sizes or a high chunk size variance. Thus, the default setting for chunking in IPFS, which is FSC at 256 KiB, can be considered the best choice for this dataset.

We conclude that, indeed, FSC is the best choice most of the time since most files, especially larger files, have high binary entropy due to internal compression. However, as we know from our first analysis, CDC algorithms has the potential to significantly reduce space requirements by the means of deduplication if the data is highly correlated. We have seen deduplication ratios above 10% for the day-by-day retrieval of a website, for the retrieval of multiple consecutive versions of source code, and for the retrieval of multiple versions of Linux as ISO images. Those datasets had in common that they were mostly text-based and therefore uncompressed, and that the files in the datasets had a very high correlation as they essentially represent multiple versions of themselves. In our empirical analysis, we were only able to evaluate approximately 8 minutes of user behavior (1 minute in every hour for 8 hours). We understand that the deduplication success of CDC algorithms depends on the user behavior over a long period of time, i.e., how many file sources (e.g., a website) does the user access repetitively and regularly, and how often and how much do these files change. The time frame of 8 minutes is too short to observe such access patterns and ultimately see the positive effects of CDC on a highly correlated dataset. Apart from this, it can be argued that the artificial datasets in our analyses are unrealistic to reflect the datasets users are going to request from IPFS, even on a large timeline. At least, it might be rare enough that it does not seem to be worth using CDC over FSC. However, it does seem likely for many users that they would use IPFS to request the same website regularly and that also the website would change regularly and gradually. Another thing to consider is the chunk size around 256 KiB. The WEB dataset consisted of TAR archives of the whole

[5] As we revealed previously, 99% of the data in the dataset comes from binary files, and 84% of the data comes from an unclassified file type. In the difference are various file types, including some for which we have conducted analyses (PNG, JPEG, and PDF) but also many for which we have not.

website. This includes HTML files, stylesheets, script files, and images, all wrapped into a single file. We have to consider that when a user requests a website in IPFS, those files will be retrieved as separate objects, i.e., each image, each CSS file, etc., as single files. While the content of an HTML, CSS, or JavaScript file can change, those files are typically very small, i.e., ≤ 256 KiB [81, 82]. Therefore any small modification to a website is likely to affect the same number of chunks with CDC as it would have with FSC.

Generally, the conditions for a file to be suitable to take advantage of CDC can be summarized as follows.

a. The content must be mostly non-binary.

b. The content must be of considerable length[6].

c. The content must be updated regularly (cross-version correlation).

d. The content must be accessed repeatedly by the same users.

It is also possible that a single file contains enough redundant content that CDC could attain considerable deduplication ratios on the file itself already. In this case, conditions (c) and (d) would not have to be true.

On the basis of these findings, we conclude that FSC at 256 KiB should remain the default chunking algorithm, as it is the computationally most efficient algorithm and as CDC algorithms most of the time would not effectively contribute to the deduplication ratio in almost all cases. However, we think that the option for a CDC algorithm should be given for the rare cases where it does make sense. Our analysis suggests that FastCDC is the best algorithm to choose for CDC. FastCDC has been among the algorithms with the highest deduplication ratios. At the same time, it is significantly faster than any other CDC algorithm: almost 10× as fast as Rabin, and 1.6× as fast as Buzhash. Buzhash appeared to be slightly superior to Rabin and FastCDC in terms of deduplication performance. However, we attribute this to the fact that Buzhash generally produced much smaller chunks than the target chunk size, whereas with FastCDC it has been the opposite. Therefore, we suggest to remove Rabin and Buzhash as the currently supported options for CDC in IPFS, and replace it with the singular option to run CDC using FastCDC. It should still support a configuration for the minimum, average, and maximum chunk size. In the CLI, this change could be expressed as follows.

```
-s, --chunker           string - Chunking algorithm, size-[bytes],
                                 cdc-[min]-[avg]-[max].
                                 Default: size-262144.
```

[6] We do not know what a considerable length would be, as even a file size of 512 KiB has a very high chance of producing a similar Δ with any small local modification of the file. So ideally, the file size is even much bigger. This is considering a target chunk size of 256 KiB.

The option for CDC and further the customization of its parameters can theoretically be used to achieve optimal results adjusted to a specific file or set of files. In practical means, the suitability of CDC and the appropriate values of its parameters are difficult to assess for a human. The user might also not be incentivized enough to make this effort for a, potentially minor, optimization on the global scale. We think that an automatic assessment for the right chunking option by evaluating, e.g., the byte structure of a file, its MIME-type, its size, and perhaps other features, could be a valuable addition to IPFS. However, we acknowledge that such an algorithm might not be able to assess if the conditions (c) and (d) apply. Furthermore, depending on the specifics of the algorithm, its time complexity might be an issue.

7. Conclusion

We have analyzed deduplication strategies in IPFS. To this end, we have researched state-of-the-art chunking algorithms and conducted a comprehensive analysis and comparison between FSC, Rabin, Buzhash, FastCDC, and AE. The analysis considered the metrics of computational efficiency, deduplication ratio, average chunk size, and chunk-size distribution. As most of these metrics (in CDC algorithms) depend on the dataset, we have conducted this analysis on eight different datasets. During this analysis, we were able to reproduce many of the results given by the authors of FastCDC [9, 10] and AE [8], which also drew comparisons to Rabin. Furthermore, we believe to be the first to show the chunk-size distribution with regard to different datasets for all examined chunking algorithms. To the best of our knowledge, we are also the first to present results for Buzhash on any metric. Our analysis has revealed to us that FastCDC is the fastest and most memory-efficient among the CDC algorithms. Regarding deduplication performance, we have also shown it to maintain similar levels as Rabin and Buzhash, and generally achieve better results than AE.

Moreover, we have collected an empirical dataset based on traces that we collected from IPFS. This allowed us to get some insights on the quality of data and the user behavior in there. Our evaluation of this dataset has led us to the realization that it cannot benefit from the features of CDC. We conclude that taken a greater dataset from IPFS, CDC could improve the deduplication ratio, but only in a negligible fraction of cases. Therefore, we suggest to not change the default chunking option within IPFS, which is FSC at 256 KiB. However, as an improvement to IPFS' current implementation, we suggest to replace Rabin and Buzhash as the two current options for CDC with FastCDC, as it proved to be much more efficient while achieving similar deduplication ratios.

8. Future Work

As we have elaborated in Section 6, to measure the potential of deduplication by CDC with correlated files, a much larger time frame would be required for monitoring. In order to keep the bias of ephemeral files small, the processing of the traces would have to happen simultaneously with the monitoring, ideally in time with the capturing of new messages. Besides that, we consider most of the data that our methodology captured as chunk data instead of whole files. Thereby, accumulating more data would generally produce more confidence with the results and potentially create new insights. Moreover, in future work we would like to get a better understanding of the data quality in IPFS. We haven take some initiative to that in this study by analyzing the file sizes and MIME-types. However, we think that a more sophisticated analysis could give new insights and ultimately lead to the formalization of better deduplication strategies. Furthermore, a better understanding of the binary structure of popular file types like PNGs and ZIPs could likewise lead to more sophisticated strategies. We have found that there exist approaches for what is called *content-aware* or *application-specific* chunking [4, 83]. Those algorithms aim to set chunking boundaries more naturally based on an understanding of the file type and are thereby able to significantly improve the deduplication ratio.

In relation to that, it would also have been interesting to see how the chunking algorithms perform deduplication-wise on files of a specific type rather than on a set of files of a specific type, which is what our analysis covered. That is, we lacked insights over the redundancies found inside single files (e.g., a single Linux ISO image or a single snapshot of a website) and instead focused on the correlation of files within a dataset. In our analyses, we have also not given much attention to the determination of the best chunk size, as we mostly sticked to what seemed to be generally accepted in these scenarios, which is around 256 KiB. Smaller chunk sizes, as we have seen in our analysis of the artificial datasets, can significantly increase deduplication performance. However, they introduce a greater index overhead, i.e., more metadata, and also affect the computational efficiency negatively. We think that it would provide interesting insights to quantify these effects in another analysis.

Furthermore, there are many ideas with the potential to improve deduplication that we have not explored in this thesis. The application of parallelized chunking (cf. Section 3.4) seems to be an easy to implement improvement on the computational speed of CDC algorithms. While it does not improve the deduplication ratio by itself, it can reduce the speed difference between FSC and CDC and thereby weaken the argument for FSC. Ultimately, this could lead to a more frequent or potentially the default application of CDC. Furthermore, it could also generally improve the performance, even for FSC, by pipelining and parallelizing the steps of chunking, fingerprinting, indexing, and writing. Another auspicious strategy, which we have seen employed in ZFS [69] and Windows [70], is the compression of chunks to further reduce storage cost.

In conclusion, we think there are yet untouched approaches to improve deduplication within IPFS and furthermore that our empirical analysis when expanded could lead us to a more sophisticated understand of the data landscape and needs in IPFS.

References

[1] Marcel Gregoriadis, Robert Muth, and Martin Florian. Analysis of arbitrary content on blockchain-based systems using bigquery. In *Companion Proceedings of the Web Conference 2022*, WWW '22, page 478–487, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447*, 2021.

[3] Protocol Labs. Protocol labs, 2022. `https://protocol.ai/`, visited on 2022-09-12.

[4] A Venish and K Siva Sankar. Study of chunking algorithm in data deduplication. In *Proceedings of the International Conference on Soft Computing Systems*, pages 13–20. Springer, 2016.

[5] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.

[6] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2004.

[7] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 269–282, 2008.

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

[9] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. Fastcdc: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, 2016.

[10] Wen Xia, Xiangyu Zou, Hong Jiang, Yukun Zhou, Chuanyi Liu, Dan Feng, Yu Hua, Yuchong Hu, and Yucheng Zhang. The design of fast content-defined chunking for data deduplication based storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2017–2031, 2020.

[11] Leonhard Balduf, Sebastian Henningsen, Martin Florian, Sebastian Rust, and Björn Scheuermann. Monitoring data requests in decentralized data storage systems: A case study of ipfs. *arXiv preprint arXiv:2104.09202*, 2021.

[12] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[13] Peter Deutsch. Deflate compressed data format specification version 1.3. Technical report, 1996.

[14] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9. IEEE, 2009.

[15] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. {SiLo}: A {Similarity-Locality} based {Near-Exact} deduplication scheme with low {RAM} overhead and high throughput. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[16] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[17] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[18] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[19] Michael O Rabin. Fingerprinting by random polynomials. *Technical report*, 1981.

[20] Andrei Z Broder. Some applications of rabin's fingerprinting method. In *Sequences II*, pages 143–152. Springer, 1993.

[21] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.

[22] Cezary Dubnicki, Krzysztof Lichota, Erik Kruus, and Cristian Ungureanu. Methods and systems for data management using multiple selection criteria, November 30 2010. US Patent 7,844,581.

[23] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX annual technical conference, general track*, pages 113–126. San Antonio, TX, USA, 2003.

[24] Neil T Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–95, 2000.

[25] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data {Deduplication—Large} scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[26] Jonathan D Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems (TOIS)*, 15(3):291–320, 1997.

[27] Daniel V Pryor, Mark R Thistle, and Nabeel Shirazi. Text searching on splash 2. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–177. IEEE, 1993.

[28] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014.

[29] Nikolaj Bjørner, Andreas Blass, and Yuri Gurevich. Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences*, 76(3-4):154–203, 2010.

[30] Juan Benet. Ipfs - content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[31] Amanda Erickson. Turkey just banned wikipedia, labeling it a 'national security threat', 2017. `https://www.washingtonpost.com/news/worldviews/wp/2017/04/29/turkey-just-banned-wikipedia-labeling-it-a-national-security-threat/`, visited on 2022-09-13.

[32] IPFS. Kubo CLI | IPFS Docs, 2022. `https://docs.ipfs.tech/reference/kubo/cli/#ipfs-add`, visited on 2022-09-13.

[33] IPFS. go-ipfs-chunker, 2021. `https://github.com/ipfs/go-ipfs`, visited on 2022-03-19.

[34] William J Bolosky, Scott Corbin, David Goebel, and John R Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. Seattle, WA, 2000.

[35] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[36] Wayne Davison. rsync, 2022. `https://github.com/WayneD/rsync`, visited on 2022-09-25.

[37] Bram Cohen. The BitTorrent Protocol Specification, 2008. `https://www.bittorrent.org/beps/bep_0003.html`, visited on 2022-09-25.

[38] Microsoft FileCAB-Team. Introduction to Data Deduplication in Windows Server 2012, 2019. https://techcommunity.microsoft.com/t5/storage-at-microsoft/introduction-to-data-deduplication-in-windows-server-2012/ba-p/424257, visited on 2022-09-24.

[39] L Anjar Fitriya, Tito Waluyo Purboyo, and Anggunmeka Luhur Prasasti. A review of data compression techniques. *International Journal of Applied Engineering Research*, 12(19):8956–8963, 2017.

[40] Rajandeep Kaur and Pooja Choudhary. A review of image compression techniques. *Int. J. Comput. Appl*, 142(1):8–11, 2016.

[41] Sachin Dhawan. A review of image compression and comparison of its algorithms. *International Journal of Electronics & Communication Technology, IJECT*, 2(1):22–26, 2011.

[42] Gaurav Vijayvargiya, Sanjay Silakari, and Rajeev Pandey. A survey: various techniques of image compression. *arXiv preprint arXiv:1311.6877*, 2013.

[43] Nasir D Memon, Xiaolin Wu, V Sippy, and G Miller. Interband coding extension of the new lossless jpeg standard. In *Visual Communications and Image Processing'97*, volume 3024, pages 47–58. SPIE, 1997.

[44] Fatema Rashid, Ali Miri, and Isaac Woungang. Secure image deduplication through image compression. *Journal of Information Security and Applications*, 27:54–64, 2016.

[45] Cristobal Rivero and Prabhat Mishra. Lossles audio compression: A case study. Technical report, Technical Report 08-415, Department of computer and information Science and ..., 2008.

[46] Chad Fogg, Didier J LeGall, Joan L Mitchell, and William B Pennebaker. *MPEG video compression standard*. Springer Science & Business Media, 2007.

[47] Fatema Rashid and Ali Miri. Deduplication practices for multimedia data in the cloud. In *Guide to Big Data Applications*, pages 245–271. Springer, 2018.

[48] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 internet measurement conference*, pages 481–494, 2012.

[49] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 205–212, 2013.

[50] Odysseas Papapetrou, Sukriti Ramesh, Stefan Siersdorfer, and Wolfgang Nejdl. Optimizing near duplicate detection for p2p networks. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10. IEEE, 2010.

[51] Yan Ke, Rahul Sukthankar, Larry Huston, Yan Ke, and Rahul Sukthankar. Efficient near-duplicate detection and sub-image retrieval. In *ACM multimedia*, volume 4, page 5. Citeseer, 2004.

[52] KK Thyagharajan and G Kalaiarasi. A review on near-duplicate detection of images using computer vision techniques. *Archives of Computational Methods in Engineering*, 28(3):897–916, 2021.

[53] Cheng Yang. Peer-to-peer architecture for content-based music retrieval on acoustic data. In *Proceedings of the 12th international conference on World Wide Web*, pages 376–383, 2003.

[54] Xiangmin Zhou, Xiaofang Zhou, Lei Chen, Athman Bouguettaya, Nong Xiao, and John A Taylor. An efficient near-duplicate video shot detection method using shot-based interest points. *IEEE Transactions on Multimedia*, 11(5):879–891, 2009.

[55] Hung-Khoon Tan, Chong-Wah Ngo, Richard Hong, and Tat-Seng Chua. Scalable detection of partial near-duplicate videos by visual-temporal consistency. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 145–154, 2009.

[56] Gousiya Farheen Shaik and Min Chen. Vidupe-duplicate video detection as a service in cloud. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 725–726. IEEE, 2019.

[57] Ying Yan, Beng Chin Ooi, and Aoying Zhou. Continuous content-based copy detection over streaming videos. In *2008 IEEE 24th International Conference on Data Engineering*, pages 853–862. IEEE, 2008.

[58] Xiao Wu, Alexander G Hauptmann, and Chong-Wah Ngo. Practical elimination of near-duplicates from web video search. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 218–227, 2007.

[59] Odysseas Papapetrou. *Approximate algorithms for efficient indexing, clustering, and classification in Peer-to-peer networks*. PhD thesis, Hannover: Gottfried Wilhelm Leibniz Universität Hannover, 2011.

[60] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[61] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[62] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660, 2005.

[63] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 744–755, 2009.

[64] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[65] ZFS Deduplication, 2022. `https://www.truenas.com/docs/references/zfsdeduplication`, visited on 2022-11-01.

[66] Frank Schmuck and Roger Haskin. {GPFS}: A {Shared-Disk} file system for large computing clusters. In *Conference on File and Storage Technologies (FAST 02)*, 2002.

[67] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[68] Myoungwon Oh, Sejin Park, Jungyeon Yoon, Sangjae Kim, Kang-won Lee, Sage Weil, Heon Y Yeom, and Myoungsoo Jung. Design of global data deduplication for a scale-out distributed storage system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073. IEEE, 2018.

[69] OpenZFS. OpenZFS, 2022. `https://github.com/openzfs/zfsc`, visited on 2022-11-01.

[70] Microsoft. Understanding Data Deduplication, 2022. `https://learn.microsoft.com/en-us/windows-server/storage/data-deduplication/understand`, visited on 2022-09-24.

[71] Andrew W Leung, Shankar Pasupathy, Garth Goodson, and Ethan L Miller. Measurement and analysis of {Large-Scale} network file system workloads. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[72] Zhichao Cao, Hao Wen, Xiongzi Ge, Jingwei Ma, Jim Diehl, and David HC Du. Tddfs: A tier-aware data deduplication-based file system. *ACM Transactions on Storage (TOS)*, 15(1):1–26, 2019.

[73] Microsoft. BranchCache, 2022. https://learn.microsoft.com/en-us/windows-server/networking/branchcache/branchcache, visited on 2022-09-24.

[74] Youjip Won, Kyeongyeol Lim, and Jaehong Min. Much: Multithreaded content-based file chunking. *IEEE Transactions on Computers*, 64(5):1375–1388, 2014.

[75] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. P-dedupe: Exploiting parallelism in data deduplication system. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, pages 338–347. IEEE, 2012.

[76] Fan Ni, Xing Lin, and Song Jiang. Ss-cdc: A two-stage parallel content-defined chunking for deduplicating backup storage. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 86–96, 2019.

[77] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Fast*, pages 239–252, 2010.

[78] IPFS Dedup Analysis Framework, 2022. https://gitlab.informatik.hu-berlin.de/ti/theses/student-content/marcel-gregoriadis-ma/ipfs-dedup-analysis.

[79] jotfs/fastcdc-go, 2020. https://github.com/jotfs/fastcdc-go, visited on 2022-08-04.

[80] Marcel Gregoriadis. mg98/ae-chunker-go, 2022. https://github.com/mg98/ae-chunker-go, visited on 2022-08-04.

[81] Ilya Grigorik. Making the web faster with http 2.0: Http continues to evolve. *Queue*, 11(10):40–53, 2013.

[82] Martí Bosch, Pierre Genevès, and Nabil Layaïida. Automated refactoring for size reduction of css style sheets. In *Proceedings of the 2014 ACM symposium on Document engineering*, pages 13–16, 2014.

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

Appendix A Chunk-Size Distributions

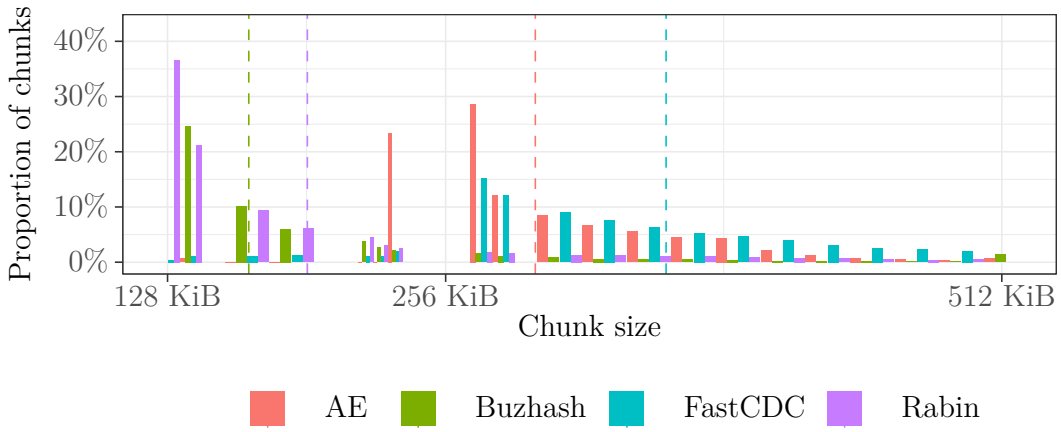


Figure 17: Chunk-size distributions on TAR.

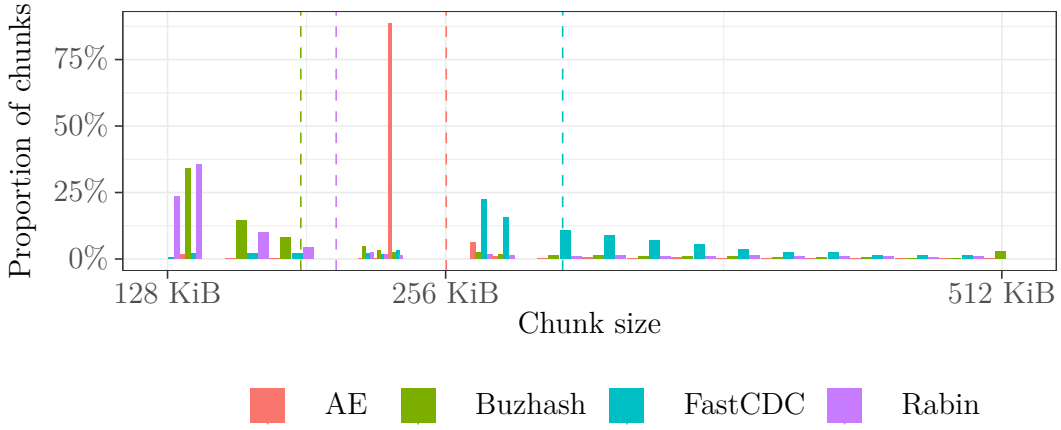


Figure 18: Chunk-size distributions on WEB.

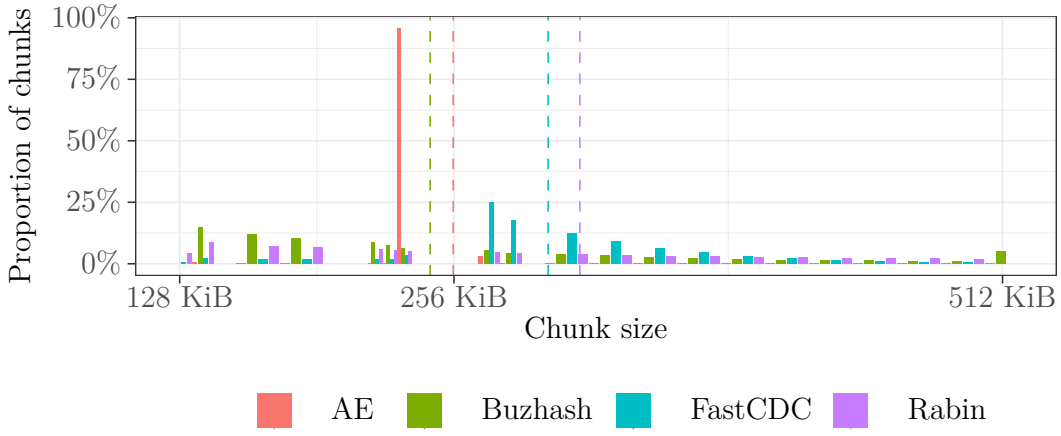


Figure 19: Chunk-size distributions on LNX.

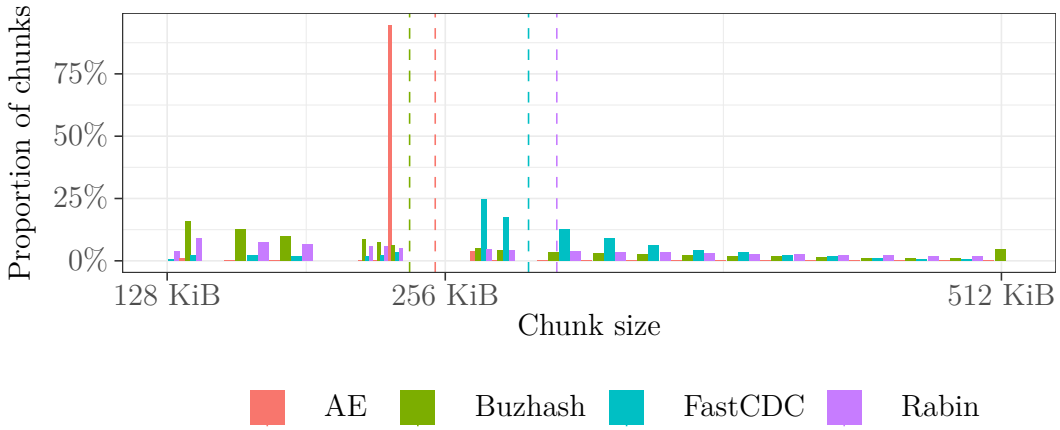


Figure 20: Chunk-size distributions on DLF.

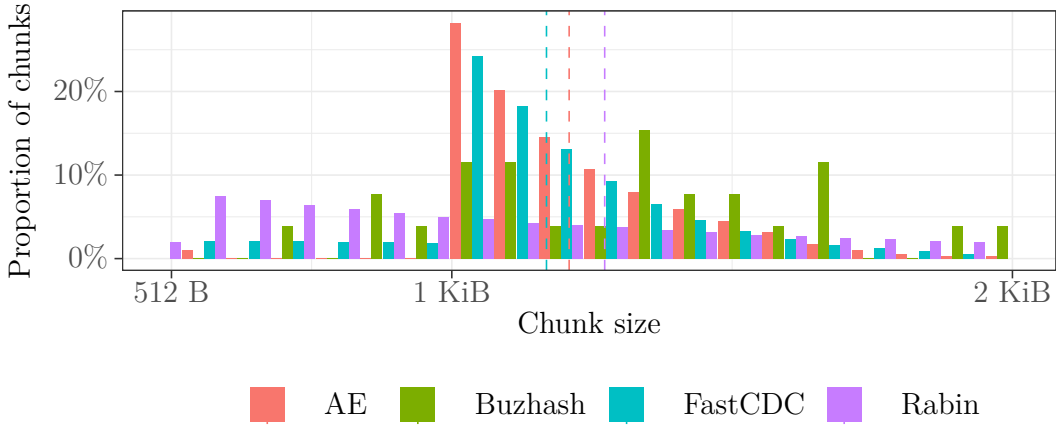


Figure 21: Chunk-size distributions on PNG.

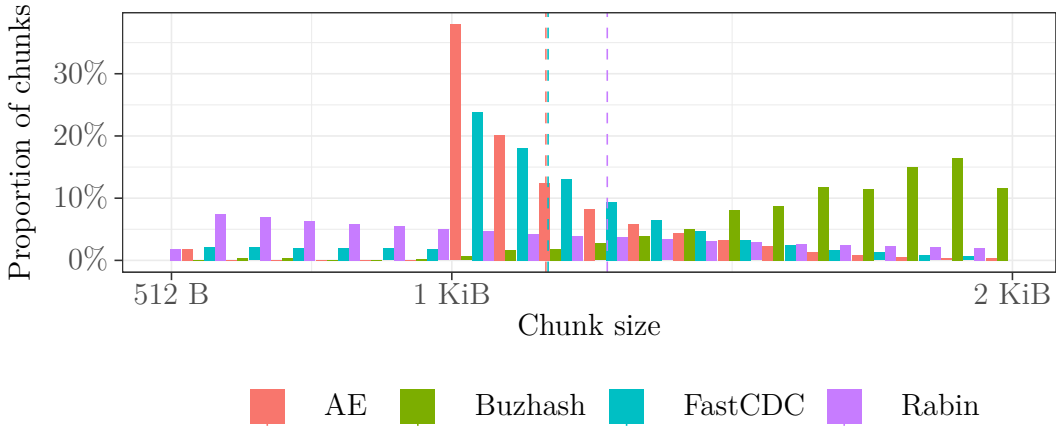


Figure 22: Chunk-size distributions on JPEG.

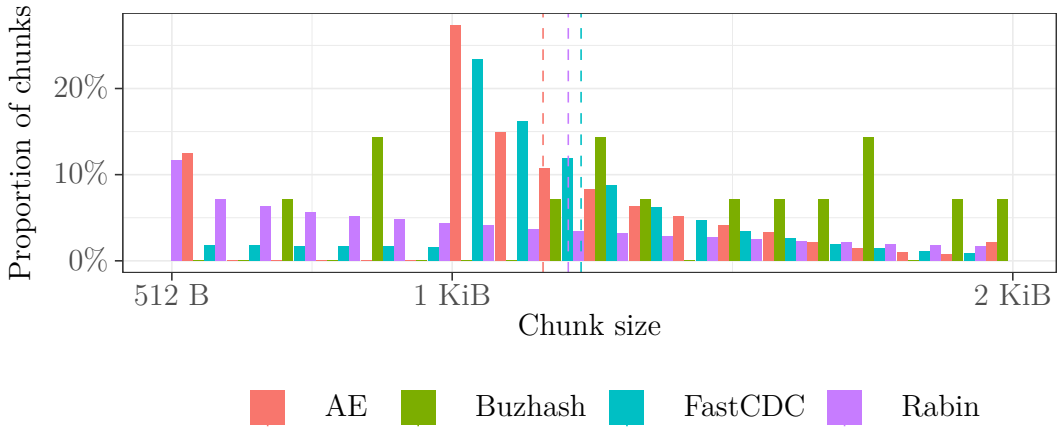


Figure 23: Chunk-size distributions on PDF.

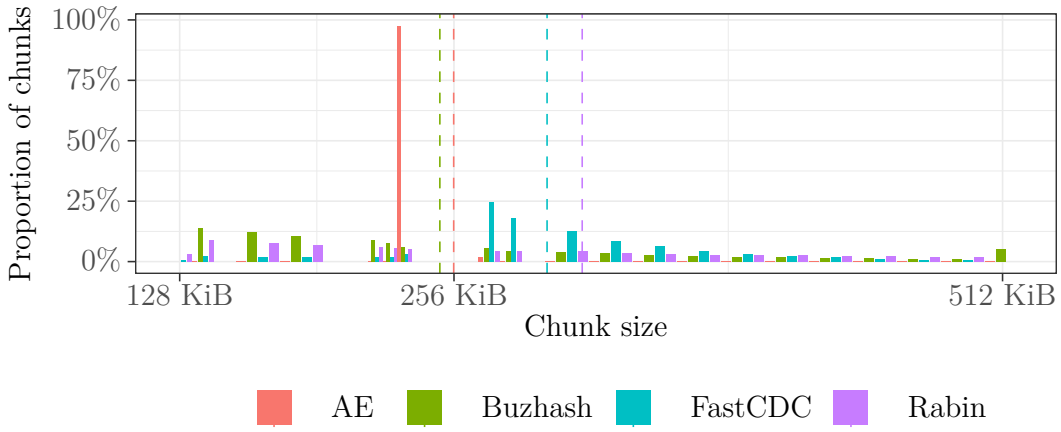


Figure 24: Chunk-size distributions on MOV.

Appendix B Probability of Only Unique Chunks

We have 127 GB of data split into chunks of 4 KiB. This gives us a total set of $n = 127\,\text{GB}/4\,\text{KiB} \approx 32$ million chunks. Let us assume maximum entropy within the data, i.e., every chunk has a random value in $[0; 2^{4096 \cdot 8})$. Let the number of possible values be $k = 2^{4096 \cdot 8} = 2^{32768}$.

We want to know the probability P that every chunk is unique. This is essentially an instance of the birthday problem and we know the formula of this probability to be as shown in Equation 14.

$$P = \prod_{i=1}^{n-1} \left(1 - \frac{i}{k}\right) \tag{14}$$

However, for practical reasons, it is infeasible to compute this equation with parameters n and k as large as in our case. Therefore, we strive to find an approximation for P.

To this end, let $x = \frac{i}{k}$. We can then make use of the inequality $1 - x \leq e^{-x}$. This gives us an approximation for P, which is excellent for small x (which it is for us because k is very large): $P \leq \prod_{i=1}^{n-1} e^{-x}$. The approximation error can be expressed as the difference of $e^{-x} - (1 - x)$. Substituting e^{-x} by its representation of a taylor series, i.e., $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots$, we can see that the approximation error is in $\mathcal{O}(x^2)$.

In Equation 15, we apply the approximation error on the formula for P. Here we can see that the approximation error is in $\mathcal{O}\left(\frac{n^3}{k^2}\right)$. With k so large, it is almost 0 and therefore negligible.

$$\prod_{i=1}^{n-1} \left(1 - \left(\frac{i}{k}\right)^2\right) \approx n \cdot \left(\frac{n}{k}\right)^2 = \frac{n^3}{k^2} \tag{15}$$

Now, by substituting $x = e^{-x}$ in the formula for P and with further transformations, we get to the term shown in Equation 16.

$$P \leq \prod_{i=1}^{n-1} \exp(-\frac{1}{k}) = \exp\left(-\sum_{i=1}^{n-1} \frac{i}{k}\right) = \exp\left(-\frac{\binom{n}{2}}{k}\right) \tag{16}$$

This expression shows us, as k is much larger than $\binom{n}{2}$, that P must be close to 1.