The Design of Fast and Lightweight Resemblance Detection for Efficient Post-Deduplication Delta Compression

Article *in* ACM Transactions on Storage · February 2023

DOI: 10.1145/3584663

CITATIONS
44

READS
141

7 authors, including:

Wen Xia
Huazhong University of Science and Technology
84 PUBLICATIONS **2,353** CITATIONS

SEE PROFILE

Xiangyu Zou
Harbin Institute of Technology Shenzhen
43 PUBLICATIONS **483** CITATIONS

SEE PROFILE

Philip Shilane
Dell Technologies
64 PUBLICATIONS **4,660** CITATIONS

SEE PROFILE

The Design of Fast and Lightweight Resemblance Detection for Efficient Post-Deduplication Delta Compression

WEN XIA, Harbin Institute of Technology, Shenzhen; Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, China
LIFENG PU, Harbin Institute of Technology, Shenzhen, China
XIANGYU ZOU, Harbin Institute of Technology, Shenzhen, China
PHILIP SHILANE, Dell Technologies, USA
SHIYI LI, Harbin Institute of Technology, Shenzhen, China
HAIJUN ZHANG, Harbin Institute of Technology, Shenzhen, China
XUAN WANG, Harbin Institute of Technology, Shenzhen; Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, China

Post-deduplication delta compression is a data reduction technique that calculates and stores the differences of the very similar but non-duplicate chunks in storage systems, which is able to achieve a very high compression ratio. However, the low throughput of widely-used resemblance detection approaches (e.g., N-Transform) usually becomes the bottleneck of delta compression systems due to introducing high computational overhead. Generally, this overhead mainly consists of two parts: ① calculating the rolling hash byte-by-byte across data chunks and ② applying multiple transforms on all the calculated rolling hash values.

In this paper, we propose Odess, a fast and lightweight resemblance detection approach, that greatly reduces the computational overhead for resemblance detection while achieving high detection accuracy and high compression ratio. Specifically, Odess first utilizes a novel Subwindow-based Parallel Rolling hash method (called SWPR) using SIMD (i.e., Single Instruction Multiple Data [1]) to accelerate calculation of rolling hashes (corresponding to the first part of the overhead). Moreover, Odess uses a novel Content-Defined Sampling method to generate a much smaller proxy hash set from the whole rolling hash set, and then quickly applies transforms on this small hash set for resemblance detection (corresponding to the second part of the overhead).

Evaluation results show that during the stage of resemblance detection, the Odess approach is ~31.4× and ~7.9× faster than the state-of-the-art N-Transform and Finesse (i.e., a recent variant of N-Transform [39]), respectively. When considering an end-to-end data reduction storage system, Odess-based system's throughput is about 3.20× and 1.41× higher than N-Transform and Finesse-based systems' throughput, respectively, while maintaining the high compression ratio of N-Transform and achieving ~1.22× higher compression ratio over Finesse.

CCS Concepts: • **Information systems** → **Deduplication**; **Extraction, transformation and loading**; **Similarity measures**.

Authors' addresses: Wen Xia, Harbin Institute of Technology, Shenzhen; Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, China, xiawen@hit.edu.cn; Lifeng Pu, Harbin Institute of Technology, Shenzhen, China, plfeng97@gmail.com; Xiangyu Zou, Harbin Institute of Technology, Shenzhen, China, xiangyu.zou@hotmail.com; Philip Shilane, Dell Technologies, USA, philip.shilane@dell.com; Shiyi Li, Harbin Institute of Technology, Shenzhen, China, lsy011025@gmail.com; Haijun Zhang, Harbin Institute of Technology, Shenzhen, China, hjzhang@hit.edu.cn; Xuan Wang, Harbin Institute of Technology, Shenzhen; Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, China, wangxuan@cs.hitsz.edu.cn.

1553-3077/2022/0-ART0 $15.00
https://doi.org/XXXXXXX.XXXXXXX

Additional Key Words and Phrases: post-deduplication delta compression, resemblance detection, SIMD, parallel rolling hash, content-defined sampling

ACM Reference Format:
Wen Xia, Lifeng Pu, Xiangyu Zou, Philip Shilane, Shiyi Li, Haijun Zhang, and Xuan Wang. 2022. The Design of Fast and Lightweight Resemblance Detection for Efficient Post-Deduplication Delta Compression. *ACM Trans. Storage* 0, 0, Article 0 (2022), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The amount of digital data is growing explosively, recent reports [11, 26] suggest that the total size of worldwide digital information has grown from about 33 ZB in 2018 to about 64.2 ZB in 2020, and that size will be more than 180 ZB in 2025. As a result of this "data deluge", efficiently eliminating redundancy to reduce data storage costs became one of the most challenging and important tasks in mass storage systems and can generally achieve a compression ratio of 10× ∼ 30× [3].

To eliminate redundancy, several compression techniques have been proposed. Among the approaches, general compression [42, 43] has a slow performance or impractical memory requirements for large scale data, though it has the potential to remove fine-grained redundancies. Deduplication [24, 30, 31, 38] is a widely studied alternative that identifies and removes coarse-grained redundancies when identical chunks of data (e.g., 8 KB in size) are replaced with references, though it misses redundancies smaller than the supported chunk size. Delta compression [7, 28] combines the advantages of coarse- and fine-grained redundancy reduction by detecting very similar items across a large storage system and then eliminating redundancy among them.

Delta compression has been widely applied in many applications, including databases [36], p2p systems [23, 40], backup storage systems [28, 39], remote synchronization [14, 27], memory management [10] and so on. For storage systems, the approach is to usually first detect the redundant data candidate (called resemblance detection), and then remove the redundant data relative to a candidate by using a delta encoding technique [18]. Here resemblance detection is always the key step in delta compression's workflow [39].

All of the state-of-the-art approaches for resemblance detection [2, 8, 10, 14, 23, 27, 36, 37, 39, 40] follow a similar framework of obtaining features from each to-be-detected item (a file or a chunk) and estimating the similarity of two items based on comparing their features. These approaches can be generally divided into three steps: *Characterizing, Selecting, and Indexing*. ① In the *Characterizing* step, a series of content characteristics of an item will be collected (i.e., hash values of all subchunks or all sliding windows). ② In the *Selecting* step, some of the generated characteristics will be sampled as the features of an item, where sampling techniques include "Fixed Position" [10], "Top-k" [2, 8, 14, 23, 36, 37, 40], "Top-1 on each linear transforms" [4, 15, 27], and "Top-1 on each subchunk" [39]. ③ In the *Indexing* step, similar items will be identified based on their features. To simplify this process, the features are organized for indexing, such as using the techniques of "Super-Feature" [15], "Bloom Filter" [14], "Sorting" [8], etc. Among these three steps, the first two steps focus on calculating and selecting features for items, and the last step is about finding similar candidates for delta compression according to their features.

Resemblance detection approaches usually generate several features for each item (i.e., a chunk or a file), and estimate the similarity (degree) between items according to the number of matched features. Most approaches [23, 36] generate **sequence-insensitive** features, which means features from two items may be matched out of order. For example, the i^{th} feature of Chunk A could match the j^{th} feature of Chunk B, which has similarities to the bag-of-words approach in natural language processing. Thus, for sequence-insensitive features, estimating similarity needs to calculate the intersection of the feature set for each item, which makes searching for similar candidates slow. On the other hand, N-Transform [4] is the only approach that generates **sequence-sensitive**

features, meaning the i^{th} features of items are directly compared. With the speed benefits of sequence-sensitive features (only considering in-position matching), N-Transform introduces the Super-Feature method, which calculates hashes over a list of features (e.g., four features) as Super-Features. When checking for similar items, Super-Features are directly searched across items, and items with any matched Super-Feature are similar candidates, which is very simple and quick. However, generating **sequence-sensitive** features involves applying several linear transforms on all the characteristics generated from the *Characterizing* step, which is computationally more expensive in the *Selecting* step [33, 39].

Finesse [39] aims at improving N-Transform for the *Selecting* step by an approximation method, which exploits the fine-grained locality inside similar items. Compared with N-Transform, Finesse achieves much higher speed (5× [39]) by avoiding the calculation of linear transforms in the *Selecting* step, but it generates sequence-insensitive features. Finesse calculates Super-Features (which usually require sequence-sensitive features) in the *Indexing* step for faster searches for similar candidates, although Finesse's features are sequence-insensitive. Thus, it leads to errors in detecting similar candidates (a lower detection efficiency) and also a lower compression ratio.

Thus, to combine the advantages of the above two approaches, we try to design an approach which generates sequence-sensitive features at high speed. Specifically, we focus on reducing the calculations of N-Transform, and several observed properties guide our research. The calculation of N-Transform mainly includes two parts (*Characterizing* and *Selecting*), and we optimize them separately. For the *Characterizing* step, we reduce the calculations in two ways. ① We introduce a rolling hash with a simpler calculation instead of Rabin, called Gear [34]. ② We utilize SIMD[1] (i.e., Single Instruction Multiple Data) and apply multiple sliding windows for data vectors, which supports parallel calculating with SIMD and hugely accelerates this step. For the *Selecting* step, we introduce a novel Content-Defined Sampling method to generate a smaller proxy (i.e., a sampled set) with consistently sampling the content-defined hash values generated by the *Characterizing* step. By running the *Selecting* step on our much smaller proxy, the calculation for the *Selecting* step is dramatically reduced. At the same time, we verify that, compared with N-Transform, our approach, with a reasonable sampling rate (e.g., 1/128), has only a slight impact on the detecting efficiency, in theory and evaluation.

Finally, we propose Odess, a fast and lightweight resemblance detection approach, which generates sequence-sensitive features by using a novel Content-Defined Sampling method on a novel Subwindow-based Parallel Rolling (SWPR) hash method using SIMD. Generally, the contributions of this paper are four-fold:

- We introduce a novel SubWindow-based Parallel Rolling (SWPR) hash method using SIMD, which runs much faster than Rabin and the original Gear while maintaining the high hashing efficiency. This method reduces calculations for the *Characterizing* step.
- We introduce a Content-Defined Sampling method to reduce the calculations on the *Selecting* step. We demonstrate that our method still follows Broder's Theorem [4], which ensures high detection accuracy and produces sequence-sensitive features. It is also more scalable than previous approaches by using Content-Defined Sampling: e.g., it can efficiently support using more features for better detection efficiency.
- We propose Odess, a fast and accurate resemblance detection approach. We verify that Odess can provide approximate compression ratio similar to the state-of-the-art approaches, e.g., N-Transform, while greatly improving the speed of resemblance detection.
- Evaluations suggest that Odess generates features about 31.4× faster than N-Transform with a comparable resemblance detection efficiency and 7.9× faster than Finesse with a better detection accuracy. In a data reduction system combining with deduplication and delta

compression, Odess achieves about 1.41× faster throughput and 1.22× higher compression ratio than Finesse, and about 3.20× faster throughput and nearly the same compression ratio as N-Transform.

The rest of this paper is organized as follows. In Section 2, we introduce necessary background and related works. In Section 3, we discuss the deficiency of the state-of-the-art approaches and provide the design and implementation details of this work. In Section 4, we discuss the evaluation results of Odess, including the comparison to state-of-the-art approaches: Finesse and N-Transform. Finally, in Section 5, we conclude this paper.

2 BACKGROUND AND RELATED WORKS

2.1 Background

Redundancy elimination is an important technique to control storage costs, and there are generally three approaches: general compression, data deduplication, delta compression.

General compression is designed for reducing redundancy data at the byte/string level by using traditional algorithms, e.g., dictionary coding [42, 43] and Huffman coding [12]. Because of fine-grained redundancy searching, general compression usually achieves a good compression ratio. However, because general compression has a window within which it can find repeated patterns (the window size is kilobytes to megabytes typically), general compression cannot identify data redundancies that are distant from each other.

Data deduplication [31] is able to identify duplicate regions across an entire storage system, though at a coarse granularity. Data deduplication works by splitting data into chunks (or fixed-sized blocks) and replacing duplicate chunks with references. An index of content hashes and other mechanisms are used to identify duplicates. Compared with general compression (e.g., GZIP's typical throughput is 100MB/s), deduplication has a processing (i.e., compressing) speed up to the GB/s level and is practical for large storage systems. But, because deduplication cannot identify redundancies smaller than the chunk size, it cannot remove redundancies between similar, non-identical chunks.

Delta compression [7, 27] bridges the gap between data deduplication and general compression and is usually implemented along with both to maximize the data compression ratio. Items (i.e., deduplication unit, a file or a chunk) are represented by features, and the similarity of two items is estimated based on comparing their features. When similar items are found, delta encoding [18] will be applied to find the byte-wise diff, which is typically much smaller for highly similar items. This approach can eliminate fine-grained redundancy across an entire storage system. It has been applied in many works from different domains, e.g., storage systems [15, 27, 39], databases [36, 37], p2p systems [23, 40], memory [10], etc.

2.2 Existing Resemblance Detection Approaches

As delta compression is able to maximize redundancy elimination among similar chunks in storage systems (compared with general compression and deduplication), we focus on resemblance detection for delta compression in this work.

Many approaches of resemblance detection have been proposed that differ in important ways, which are summarized in Table 1. All of these approaches generate features from each item and then compare features to estimate similarity. **Detection accuracy** reflects the approximation degree between the actual similarity and the estimated similarity. Among these approaches, they share three common steps:

(1) **Characterizing**: characteristics of the item (e.g., hash values of all subchunks or all sliding windows) are generated from an item.

Table 1. Approaches for resemblance detection.

Approaches	Characterizing	Selecting	Indexing	Type of feature
Coarse-grained Top-k [23, 36, 37]	coarse-grained	Top-k	Bucketing	sequence insensitive
Fine-grained Top-k [2, 40]	fine-grained	Top-k	Bucketing	sequence insensitive
Coarse-grained BF [14]	coarse-grained	Top-k	Bloom Filter	sequence insensitive
Coarse-grained Sorting [8]	coarse-grained	Top-k	Sorting	sequence insensitive
N-Transform [15, 27]	fine-grained	Top-1 on each linear transform	Super Feature	sequence sensitive
Fixed-position [10]	fine-grained	Fixed Position	Super Feature	sequence insensitive
Finesse [39]	fine-grained	Top-1 on each subchunk	Super Feature	sequence insensitive

(2) **Selecting**: an extracting method is applied to reduce the scale of the characteristics, thus features (a.k.a., signature or other terms in some papers) are extracted from characteristics generated from the 1^{st} step.
(3) **Indexing**: item's features are added to a similarity index.

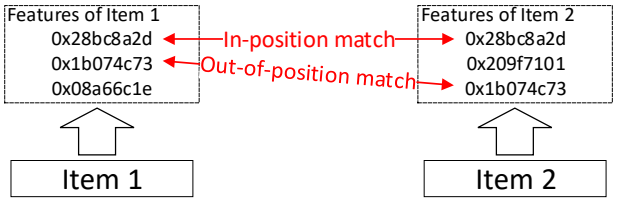


Fig. 1. An example of feature matching between two items in Top-k. Note that these features are sequence-insensitive, which means in-position and out-of-position matches are both possible and complicate resemblance detection.

Coarse-grained Top-k is used in many works [23, 36, 37]. Two works [23, 37] apply this approach to detect similar files, and another [36] applies this approach to detect similar records in databases. The *Characterizing* step for this approach is separating a to-be-detected item into several blocks with Content-Defined Chunks (CDC), and calculating hash values for each block. All hash values consist of the characteristics of an item, and similar items will share some of the same characteristics, which means they are likely to share some of the same blocks. Because this *Characterizing* identifies commonality at the block level, this is a coarse-grained approach. The *Selecting* step for this approach selects up to the maximum k characteristics as features, which we called Top-k. Similar items will share some or all of the features, and items that have more features in common tend to have a higher similarity degree. This kind of feature is **sequence-insensitive**, because the order of features is ignored, and matched features may or may not have positional correspondence as shown in Figure 1. Therefore, the intersection and union of features are required to calculate the Jaccard Similarity metric:

$$Similarity(item1, item2) = \frac{|\{Features\ of\ item1\} \cap \{Features\ of\ item2\}|}{|\{Features\ of\ item1\} \cup \{Features\ of\ item2\}|}. \quad (1)$$

Thus, features' matching between items is complex, which makes indexing mechanisms also complex. The usual *Indexing* step for this approach is Bucketing, in which each feature maps to a bucket. $Bucket(f)$ records all the items which includes the feature f. When looking up a similar candidate for a given item k, we first generate the features of k (such as f_1, f_2, f_3), next we check the

corresponding buckets ($Bucket(f_1)$, $Bucket(f_2)$, $Bucket(f_3)$), and finally we select the item that has the most matching features from these buckets. Thus, finding candidates needs to traverse all the corresponding buckets, which is time-consuming. Meanwhile, this indexing is memory-intensive since all of the features need to be saved in buckets.

Fine-grained Top-k has been applied in previous work [2, 40]. The first step in this approach is running a rolling hash method on the to-be-detected item, sliding the window of rolling hash byte-by-byte, and recording the hash value of each sliding window. Because this kind of *Characterizing* step generates characteristics at the byte/string level, we call it fine-grained style *Characterizing*. The second and the third steps of this approach are also Top-k and Bucketing, and the drawbacks are the same as the Coarse-grained Top-k approach.

Coarse-grained BF [14] has been applied to detect similar chunks at a large granularity (e.g.16MB). The first step is chunk-based (Coarse-grained: at the chunk level) and the second step is Top-k, which is the same as the first approach. The novel part of this technique is introducing a Bloom filter [14] in the *Indexing* step. After features are generated with chunk-based *Characterizing* and Top-k style *Selecting*, they will be inserted into a Bloom filter. Therefore, each item is represented by a Bloom filter, and the similarity of two items can be estimated with the Hamming Distance between their Bloom filters. Compared with 'Bucketing', this indexing solution reduces the memory cost, but it also has serious shortcomings. First, because of the hash collisions, we can not acquire the exact similarity from Bloom filters. Second, there is no straightforward way to index Bloom filters (unlike features), so finding a similar item may involve comparing against all the previous Bloom filters (one per item already processed).

Coarse-grained Sorting [8] focuses on offline similar file detection. The 1^{st} and the 2^{nd} steps also use chunk-based *Characterizing* and Top-k *Selecting*. Because it is designed for offline situations, sorting is used to estimate the similarity relationship between files. Features of all files in the system are recorded in a feature-file, and an external sort is applied to all the features. After sorting, identical features will be consecutive, and the corresponding files are likely similar. This kind of *Indexing* is also time-consuming and thus is applied offline in resemblance detection.

N-Transform was first proposed by Broder [4], and it has been applied in several works [15, 27], which focus on detecting similar chunks (about 8KB in general). It is based on Broder's Theorem that requires several predefined linear transforms. In the *Characterizing* step, this approach follows the fine-grained style. In the *Selecting* step, N-Transform, which is very different from the above approaches, calculates linear transforms on hash values of each sliding window and records the minimum values of each linear transform's result as features. Specifically, each linear transform maps to a feature. This kind of *Selecting* is based on Broder's Theorem (detailed in Section 2.3), and generates **sequence-sensitive** features because matched features only come from sliding windows with the same content and the same linear transform. Thus, we only need to consider positional matches of features in N-Transform and this kind of features are friendly to indexing. Finally, if the number of the features for an item is n and the number of matched features is k, the Jaccard Similarity can be simply estimated by

$$Similarity(Item1, Item2) = \frac{k}{n}. \tag{2}$$

The *Indexing* step of N-Transform is often implemented with Super-Features, first proposed by Kulkarni et al. [15], in which features are grouped into several **Super-Features**. For example, if we have features f_1, f_2, f_3, f_4, they can be grouped into two Super-Features as $SF_1 = hash(f_1, f_2)$ and $SF_2 = hash(f_3, f_4)$. The numbers of features and Super-Features are configurable, and two items sharing at least one Super-Feature will be regarded as similar items. Compared with other indexing approaches, Super-Features lead to much faster similarity search and lower memory costs since

there are many fewer super-features than features to index. N-Transform has downsides though, as the linear transforms make it slow in the *Selecting* step.

Fixed-Position [10] is designed for redundancy detection and elimination in memory. When virtual machines run on a physical machine, memory pages of different virtual machines may be identical or very similar. This approach picks data blocks in predefined positions within pages and calculates hash values for each data block. Because the positions are random, the characterization of this approach could be regarded as fine-grained, and we call the *Selecting* step of this approach as fixed position, but it suffers from not only the known 'boundary-shift' problem [20], which means the position of identical contents in two items may be shifted (due to content modifying, inserting, etc.), but also poor detection efficiency and thus low compression ratio. Note that the *Indexing* used in this approach is also "Super-Feature".

Finesse [39] is designed to significantly improve the *Selecting* performance of N-Transform. The first step of Finesse is also fine-grained style *Characterizing*. The second step of Finesse separates the item into several subchunks and records the minimum hash value in each subchunk as features, which generates sequence-insensitive features. The third step of Finesse also follows the idea of the Super-Feature method, and to alleviate the disadvantages of sequence-insensitive features, a sorting step is applied before grouping Super-Features. However, the accuracy of Finesse is still not as high as N-Transform, and we classify Finesse as sequence-insensitive, despite the sorting step, since sorting features does not overcome insertion and deletion changes.

In summary, among approaches we reviewed, N-Transform is the only option with sequence-sensitive features (simple and fast to match) and high detection accuracy, which makes it the most attractive option to pursue resemblance detection [15, 17, 27, 33]. But generating sequence-sensitive features is time-consuming in N-Transform. In the following subsections, N-Transform and the faster version called Finesse will be presented in greater details to motivate our work.

2.3 N-Transform Details

As mentioned above, N-Transform is the most popular method for detecting very similar chunks for redundancy elimination [15, 17, 27] due to generating sequence-sensitive features.

N-Transform, as Figure 2 shows, relies on Broder's Theorem [4, 5], which states that the probability of the two sets A and B having the same minimum hash element is the same as their Jaccard similarity coefficient [13], as detailed below:

Broder's Theorem. *Consider two sets A and B, with H(A) and H(B) being the corresponding sets of the hashes of the elements of A and B respectively, where H is chosen uniformly and randomly from a min-wise independent family of permutations. H maps an element in the set to an integer. Let min(S) denote the minimum element of the set of integers S. Then:* $Pr[min(H(A)) = min(H(B))] = \frac{|A \cap B|}{|A \cup B|}$

Based on this theorem, Broder proposed a resemblance detection approach that extracts a fixed number of features from a chunk. As introduced in Section 2.2, N-Transform also includes three steps: *Characterizing*, *Selecting*, and *Indexing*.

In the *Characterizing* step, a rolling hash (often Rabin [25]) is applied to generate a series of fingerprints, and these fingerprints make up a *characteristic set* (i.e., sets A or B in Broder's Theorem).

Linear transforms (denoted as $LT()$) are used as the hashes (i.e., $H()$) in Broder's Theorem to generate features in N-Transform. For example, Feature i of a chunk (length = L) is uniquely generated with a corresponding linear transform, which is defined as

$$LT_i(x) = (m_i \times x + a_i) \bmod 2^{32}, \tag{3}$$

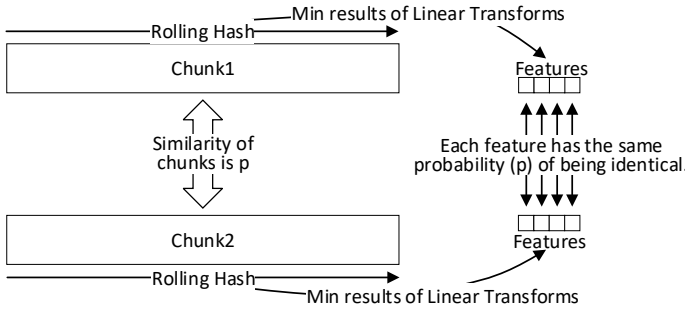


Fig. 2. General principle of feature extraction by N-Transform.

where m_i and a_i are predefined pairs of integers whose values are randomly chosen and are guaranteed not to repeat pairs (which has been discussed in previous work [6] as suitable for practical applications). The Feature i with a sliding window size of 32 bytes is acquired as

$$Feature\ i = Min_{j=1}^{L}\ LT_i(Rabin_j)\ mod\ 2^{32}. \tag{4}$$

Here, the linear transforms utilize the overflow mechanism of integer calculation, so the results of different linear transforms on the same item have different values. This results in extracted features that correspond to different positions of the item, as shown in Figure 3.

N-Transform generates features for each chunk, and their i^{th} features are all produced by the same linear transform. Thus, the linear transform could be regarded as $H()$ in Broder's Theorem, and the similarity of two chunks is equal to the probability of the minimum values of transformed *characteristic sets* (i.e., $LT(A)$ and $LT(B)$) being the same.

Just like we can estimate the probability of the result of a coin flip by tossing it multiple times, here, we use multiple linear transforms to generate multiple features and observe the number of matches to estimate the similarity between chunks. Each feature can be regarded as a series of independent, repeated trials. Thus, we can summarize that **the principle of N-Transform is running independent repeated trials on the whole chunk with different linear transforms**. N-Transform generates **sequence-sensitive** features since different features of the item to be matched are generated by different linear transforms acting on its hash value set. Feature i of different items can be directly compared without searching across all features the items.

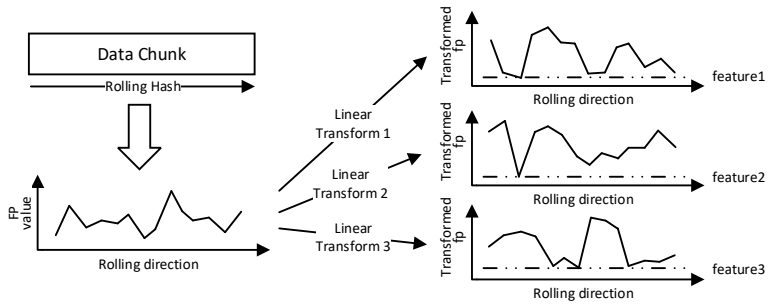


Fig. 3. An example of extracting three features in N-Transform.

Therefore, the *Selecting* step of N-Transform runs, as shown in Figure 3. The 'Rolling Hash' is applied to the 'Data Chunk' and generates a series of 'FP values'. The *Characteristics set* consists of 'FP values', and several predefined linear transforms are applied to the fingerprints, which generates several series of 'Transformed FPs'. After that, N-Transform records the minimum value

of several series of 'Transformed FPs' as features. The minimal value from applying each linear transform function is retained as a feature. When estimating the similarity of two chunks, we refer to features from two chunks generated by the same linear transform as **"a pair of features"**. Then, if N-Transform produces n features (from n linear transforms) for each chunk, and there are k pairs of features from two chunks that are identical, we can get the **Maximum Likelihood Estimation** of the actual similarity between two chunks as $\frac{k}{n}$.

Note that features in N-Transform are sequence-sensitive because we can only match features generated by the same linear transformation function. For example, it is valid that $feature_1$ of $item_1$ matches $feature_1$ of $item_2$, but if it matches $feature_2$ of $item_3$, this is not a meaningful match. To be clear, a pair of features are not guaranteed to come from the same offset in their corresponding items. This allows partial matches to be detected even if subchunks of similar chunks are shifted.

In the *Indexing* step, Features are grouped into a small number of Super-Features as explained in Section 2.2, and Super-Features may be indexed by a hashtable or other structures. Finally, if two chunks share the same Super-Feature, they are likely to be highly similar.

2.4 Finesse Details

Zhang et al. [39] proposed a method named Finesse, which improved the feature-generation throughput of N-Transform. The main idea of Finesse is consistent with the backup stream locality, which is observed by many studies [9, 16, 19, 29, 32, 41] and refers to the fact that the corresponding subchunks of chunks and their features appear in the same order among the similar chunks with a very high probability.

Finesse also can be partly explained by Broder's Theorem, as shown in Figure 4. As we discussed in Section 2.3, N-Transform applies a series of independent repeated trials on chunks to estimate the similarity. Finesse, which is based on the assumption that the differences between similar chunks are evenly distributed, runs another kind of independent repeated trial. It first splits a chunk into several equal-sized subchunks. According to the assumption, Finesse considers that the similarity between two chunks' $i-th$ subchunks should be equal to the similarity of two whole chunks. After that, Finesse no longer uses non-trivial linear transforms to generate features but takes the minimum value from a set of hash values of each subchunk. This is equivalent to using a trivial linear transform ($LT(x) = 1 \times x + 0$, i.e., $m_i = 1$ & $a_i = 0$ in Equation 3), and it applies Broder's Theorem on each pair of subchunks. A rolling hash generates a *characteristics set* for each subchunk, and the trivial linear transform is applied to the *characteristics set*. Then, the minimum values of each transformed *characteristics set* are picked as features. Because the assumption suggests that all pairs of corresponding subchunks from two chunks share the same similarity, all pairs of features from corresponding subchunks also have the same probability of being identical. Thus, the similarity of the two chunks also could be estimated as N-Transform does.

Figure 5 shows an example of how Finesse generates features. By dividing the item to be detected into N parts and extracting a feature for each part, Finesse can also obtain N features like N-Transform. Since Finesse no longer needs to perform multiple linear transforms (N-Transform requires N linear transforms per byte), the overhead of repeated calculations is greatly reduced. Hence its speed is much higher than N-Transform.

Moreover, we can also summarize that **the principle of Finesse is running independent repeated trials on different subchunks with a trivial linear transform**.

However, because Finesse uses fixed-sized subchunks, it is obvious that Finesse will encounter the known 'boundary-shift' problem [20] when extracting features, and the features of Finesse should be regarded as sequence-insensitive. Therefore, Finesse is different from N-Transform, and features of two chunks are not in one-to-one correspondence. For example, if feature #3 of chunk A

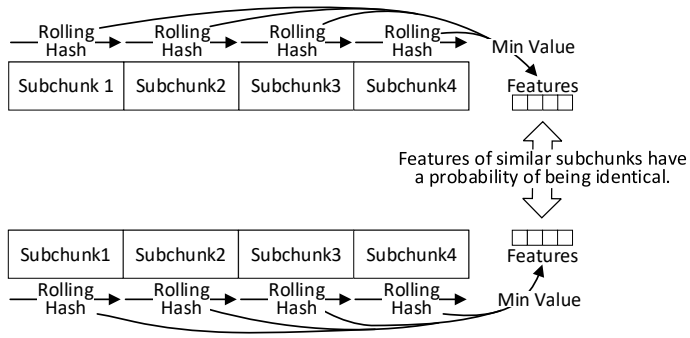


Fig. 4. General principle of feature extraction by Finesse.

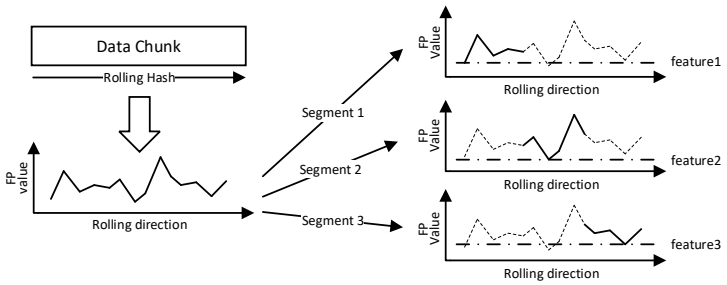


Fig. 5. An example of extracting three features in Finesse.

is equal to feature #4 of chunk B, it means the corresponding Subchunk #3 is similar to Subchunk #4. The more features that are shared by chunks A and B, the higher the similarity of chunks A and B is estimated to be, and we can measure the similarity of two chunks by the number of shared features (see Equation 1).

Finesse also uses Super-Features to simplify the process of matching features. Because its features are sequence-insensitive, Finesse uses a different Super-Feature technique. For example, there are six features $f_1, f_2, f_3, f_4, f_5, f_6$. To form three Super-Features, Finesse begins by dividing features into two lists f_1, f_2, f_3 and f_4, f_5, f_6. Next, it sorts each list of features numerically: assuming the sorted results are f_2, f_1, f_3 and f_6, f_4, f_5. Finally, it groups Super-Features as $SF_1 = hash(f_2, f_6), SF_2 = hash(f_1, f_4), SF_3 = hash(f_3, f_5)$. The sorting step alleviates the negatives from sequence-insensitive features, and supports generating Super-Features. However, Finesse still suffers from the 'boundary-shift' issue since subchunks are created at fixed boundaries instead of content-defined boundaries.

In summary, Finesse accelerates the *Selecting* step since no linear transforms are required, but reduces the detection accuracy due to generating sequence-insensitive features.

3 MOTIVATION AND DESIGN

From our review of existing approaches, it can be seen that N-Transform has a better detection accuracy but suffers from slow feature generating speed; Finesse has a higher feature generation speed but suffers from a worse detection accuracy. Thus, our goal is to design a new approach that achieves both a high feature generation speed and a better detection accuracy.

As discussed in Section 2.3, in order to speed up N-Transform, we need to reconsider the computational overhead of all three steps of resemblance detection in N-Transform: *Characterizing* and *Selecting* are very time-consuming when generating fine-grained sequence-sensitive features,

and there are opportunities to speed up these two steps, while super-feature based *Indexing* is very efficient for looking up similar candidates, and there is no need to optimize it. Therefore, our focus is on reducing calculations in these two steps.

3.1 Overview

As discussed earlier, there are two steps introducing computational overheads for resemblance detection: the *Characterizing* and *Selecting* steps. We address each step separately.

For the *Characterizing* step, the overhead is mostly caused by the Rabin rolling hash [20, 25], which requires about 5 calculating operations and 2 memory accesses each time the window is slid by one byte. Previous works have discussed the calculation cost of Rabin rolling hash [34, 35], and a much faster approach has been proposed: Gear rolling hash [34], which requires approximately half of the operations per byte of data. Properly using Gear hash could accelerate the *Characterizing* step, and moreover, we also utilize SIMD (i.e., Single Instruction Multiple Data, which has been successfully applied in the content-defined chunking algorithm [22]) to further achieve a higher rolling hash speed.

For the *Selecting* step, the overhead is caused by the linear transform calculation, which is required for each window slide of the rolling hash (a slide per byte). For example, if 12 linear transforms are required to generate 12 features, and the number of characteristics generated by the *Characterizing* step is N, the total calculation in *Selecting* is $36N$ (requiring multiply, add, and compare operations for each characteristic). From this example, we learn that the calculation of linear transforms is related to the number of characteristics. Therefore, we could reduce computational overhead by decreasing the scale of characteristics. Specifically, generating a **proxy** for characteristics is considered, which means producing a subset of characteristics to replace the original characteristics in the *Selecting* step.

In the following subsections, we will discuss these two points in details.

3.2 Generating Proxies by Content-Defined Sampling

In this subsection, we will discuss how to generate proxies for characteristics.

To ensure using proxies instead of characteristics will not reduce the resemblance detection accuracy, the generated proxies must satisfy the following preconditions:

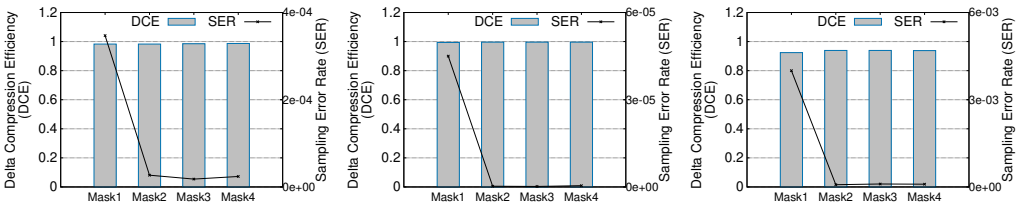
- **Certainty.** Identical items must generate identical proxies. For example, random sampling can not promise this.
- **Retention of similarity.** Similar items must generate similar proxies, and the similarity degree of two proxies should approximate the similarity degree of their corresponding items (an extension to Broder's Theorem).
- **Speed.** The generation of proxies must be fast.

Sampling is a direct roadmap for generating proxies, but according to these preconditions, many sampling methods are inapplicable, including random sampling, systematic sampling, etc. Even if we use a fixed random seed, the generated random sampling point is also fixed, and it will be similar to 'Fixed-Position', which is introduced in Section 1 and can not handle the 'boundary-shift' issue. Inspired by the principle of Content-Defined Chunking (CDC) [25, 35], we find that a 'Content-Defined Sampling' is feasible to satisfy the Certainty precondition.

More specifically, in CDC, we calculate fingerprints on each sliding window, and check whether the fingerprint *fp* of the bytes within the window satisfies a predefined condition (e.g., Odess sets the condition as '$fp \,\&\, mask == 0$', *mask* is an user-specified number [35]) to set a chunking cut point. Similar to this, we could generate proxies with Content-Defined Sampling: we check each characteristic (i.e., *fp*) and decide whether it is in a proxy by also checking whether *fp* satisfies the

predefined condition. Note that this condition is related to the sampling rate, which is decided by the value of *mask*. For example, if we use a condition that '*fp & 0x1c == 0*', the sampling rate is about 1/8 in our Content-Defined Sampling, which is decided by the number of '1' bits in '0x1c' (i.e., three bits set to '1' causes a sampling rate of $\frac{1}{2^3}$).

For small items (e.g. 1 KB), our sampling method may fail to sample with a small probability for some small items, but we find this will only cause a negligible deduplication ratio loss. Such items account for a small amount of data in the system and contribute little to the data reduction efficiency. We use the sampling error rate ($SER = \frac{number\ of\ sampling\ failures}{number\ of\ chunks\ to\ be\ detected}$) to measure the impact of sampling failure. As shown in Figure 6, in the TAR and VMA datasets, SER is as low as 10^{-5}, while in the LNX dataset, SER is as low as 10^{-7}, which supports our point.



(a) TAR (b) LNX (c) VMA

Fig. 6. Impact of the position of '1's in the *mask* in Odess. The sampling rate is set to 1/128. Mask1 is a mask with '1's concentrated in lower bits (e.g., 0x000000f7), Mask2 is a mask with '1's concentrated in the middle bits (e.g., 0x0007f000), Mask3 is a mask with '1's concentrated in the higher bits (e.g., 0x7f000000) and Mask4 is a mask where '1's are scattered (e.g., 0x40030341).

The choice of the *mask* is also an issue worth considering because different *mask*s (corresponding to the same sampling rate) may lead to different similarity matching results. In our method, we prefer to set the middle bits of the *mask* to '1'. This is because of the characteristics of rolling hash, in *fp*, the higher position bits (i.e., the leftmost bits) contain more information (i.e., they are impacted by the most bytes from the sliding window), but they also have a higher collision rate. In contract, the lower bits (i.e., the rightmost bits) have a lower collision rate but also contain less information. For example, the least significant bit of the hash value only contains the information of the most recently accessed byte because the information of the previous bytes is moved to the higher bits with the left shift operation. Figure 6 shows the experiment of sampling with different *mask*s, where DCE ($DCE = Avg.(1 - \frac{chunk\ size\ after\ delta\ compression}{chunk\ size\ before\ delta\ compression})$, the higher, the better) represents the average similarity between each chunk and its similar chunk, and SER (the lower, the better) is the ratio of sampling failures mentioned above. We observe that mask2 and mask3 perform very close on the three real datasets. Meanwhile, we find that the performance of mask2 and mask3 depends on the characteristics of the datasets. Thus, we finally choose a more reasonable *mask* like mask4 (the '1's are more evenly distributed), which achieves high DCE and low SER on all datasets.

3.3 Retention of Similarity in Content-Defined Sampling

In this subsection, we will discuss the impact of Content-Defined Sampling on the resemblance detection accuracy.

Content-Defined Sampling satisfies Certainty and Speed as discussed in the last subsection, and we discuss whether it ensures Retention of similarity in this subsection. Specifically, we have introduced a Content-Defined Sampling method in the *Selecting* step (before applying transforms),

and the estimated similarity of our approach will vary around the estimated similarity of N-Transform. Thus, we analyze estimated similarity of N-Transform and our approach to demonstrate whether our approach maintains similarity (after sampling).

As described in Section 2.3, N-Transform estimates the similarity of two chunks by comparing n feature from two chunks. For example, assume the actual similarity of two chunks is p, the number of identical features from the two chunks is k, and the estimated similarity is s. According to Broder's Theorem, each feature has an equal probability (i.e., p) to be identical. Thus, checking whether features are matched can be treated as a sequence of n independent experiments, and the count of matched features should follow the Binomial distribution. Thus, according to the probability mass function of the Binomial distribution, we learn that when $p = p_0$, the probability of k_0 features to be matched will be

$$Pr(k = k_0 \mid p = p_0) = C_n^{k_0} p_0^{k_0} (1 - p_0)^{n-k_0}. \tag{5}$$

Here $C_n^{k_0}$ means choosing k_0 elements from n elements. Because $s = k/n$, therefore we could learn:

$$Pr(s = \frac{k_0}{n} \mid p = p_0) = C_n^{k_0} p_0^{k_0} (1 - p_0)^{n-k_0}. \tag{6}$$

Thus, we get the relationship (Equation 6) between estimated similarity (i.e., s) and other specified parameters (i.e., p and n) for N-Transform.

On the other hand, we also consider the relationship between estimated similarity (i.e., s') and other specified parameters for a proxy-based approach. We denote the sampling rate (e.g., 1/32) as r, and the chunk size as c. Thus, we can get that the size of proxies will be $r \times c$. We also denote the number of identical elements in two proxies for two items as m, and the similarity of proxies as p'. Firstly, we consider the distribution of m: As we know, for two items that share characteristics, each element in the proxies has a probability (i.e., similarity p) to be sampled from the shared part between two items, so the similarity between the proxies should also follow the binomial distribution. Thus, according to the probability mass function of the binomial distribution, the number m also follows the Binomial distribution, and we learn that when $p = p_0$ and $r = r_0$, the probability of $m = m_0$ is

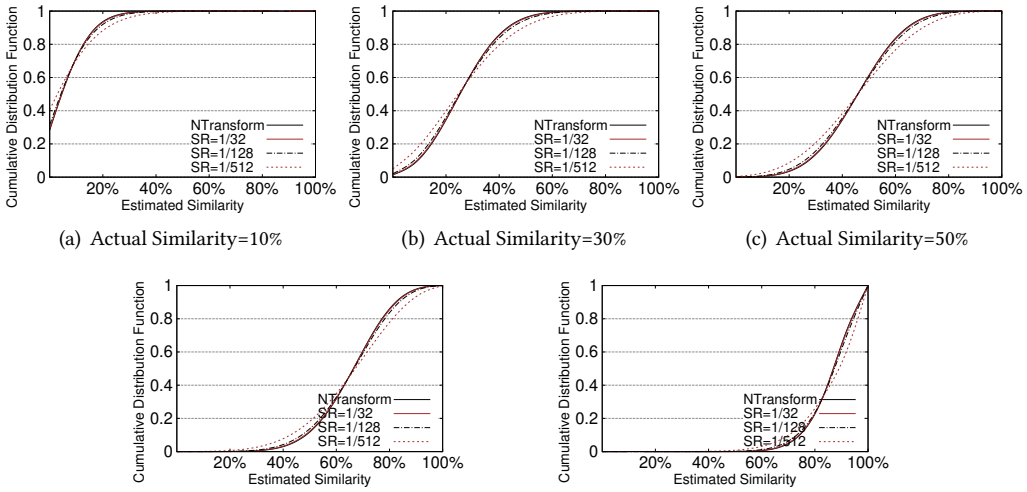
$$Pr(m = m_0 \mid p = p_0,\ r = r_0) = Pr(p' = \frac{m_0}{r_0 \cdot c} \mid p = p_0,\ r = r_0) = C_{r_0 \cdot c}^{m_0} {p_0}^{m_0} (1 - p_0)^{r_0 \cdot c - m_0}. \tag{7}$$

Next, assuming the number of matched features is k', we further consider the relationship between estimated similarity ($s' = k'/n$) and other specified parameters for our approach. Combining with the law of total probability [44], we could learn:

$$\begin{aligned} & Pr(s' = \frac{k_0'}{n} \mid p = p_0,\ r = r_0) \\ = & \Sigma_{all\ possible\ p_0'} Pr(s' = \frac{k_0'}{n} \mid p' = p_0') Pr(p' = p_0' \mid p = p_0,\ r = r_0) \\ = & \Sigma_{m_0=0}^{r_0 c} Pr(s' = \frac{k_0'}{n} \mid p' = \frac{m_0}{r_0 c}) Pr(p' = \frac{m_0}{r_0 c} \mid p = p_0,\ r = r_0). \end{aligned} \tag{8}$$

According to Equation 5, we get the result of $Pr(s' = \frac{k_0'}{n} \mid p' = \frac{m_0}{r_0 c})$, and according to Equation 7, we get the result of $Pr(p' = \frac{m_0}{r_0 c} \mid p = p_0, r = r_0)$. Thus, we finally derive Equation 9,

$$Pr(s' = \frac{k_0'}{n} \mid p = p_0,\ r = r_0) = \Sigma_{m_0=0}^{r_0 c} C_n^{k_0'} (\frac{m_0}{r_0 c})^{k_0'} (1 - \frac{m_0}{r_0 c})^{n-k_0'} C_{r_0 c}^{m_0} {p_0}^{m_0} (1 - p_0)^{r_0 c - m_0}. \tag{9}$$



(a) Actual Similarity=10% (b) Actual Similarity=30% (c) Actual Similarity=50%

(d) Actual Similarity=70% (e) Actual Similarity=90%

Fig. 7. Cumulative distribution function of estimated similarity for N-Transform and our Content-Defined approach when varying p (Actual Similarity) and sampling rates of 1/32, 1/128 and 1/512. We set the expected average chunk (item) size as 8KB and the required number of features as 12. The line with a sampling rate of 1/32 and the N-Transform line almost coincide, and the results suggest that a more frequent sampling rate (i.e., 1/32) is close to N-Transform while a less frequent sampling rate increases estimation errors, and 1/128 has an acceptable error rate.

Finally, using Equations 6 and 9, we study the relationship between the actual similarity and estimated similarity approach. Figure 7 is generated by these two equations with specific parameters, which shows the CDF (Cumulative Distribution Function) of the estimated similarity of different approaches on different actual similarity values between two items (e.g., Figure 7(b) shows that when using N-Transform with the actual similarity set to 30%, approximately 60% of chunks have an estimated similarity of less than or equal to 30%). From the results shown in Figure 7, we first see that estimated similarity's expectation of our approach is the same as N-Transform, which leads to an inference that Inference 1: compared with N-Transform, our Content-Defined Sampling approach will generate the same similarity on average. Meanwhile, results show that a more frequent sampling rate (i.e., 1/32) will lead to a result which is closer to N-Transform, and a less frequent sampling rate will increase the estimation errors, which reflects another inference that Inference 2 : the sampling rate determines the robustness of our approach.

Thus, we learn that the Content-Defined Sampling method maintains similarity and all three preconditions are satisfied.

3.4 Subwindow-based Parallel Rolling Hash with SIMD

In this subsection, we will discuss how to accelerate the *Characterizing* step.

As we mentioned in Section 3.1, the Rabin rolling hash [25] is the bottleneck in the *Characterizing* step, and there already exists a faster approach called Gear [34]. We replace Rabin with Gear, and further introduce SIMD in Gear to achieve a higher speed on rolling hash calculations. SIMD is a technique for parallelizing calculations by using a single instruction to perform the same operation simultaneously on each of multiple data items (packed into a data vector).

As shown in Equation 10, Gear generates a series of characteristics through a sliding window. $Gear_i$ (i.e., the hash value of window at position i) could be calculated by adding the left shifted

$Gear_{i-1}$ with an addend, which is generated by a predefined array $G[]$ (called the Gear array) and b_i (i.e., the byte at position i).

$$Gear_i = (Gear_{i-1} \ll 1) + G[b_i]. \tag{10}$$

The calculation of traditional Gear runs byte-by-byte in accordance with the sequence of the to-be-compressed byte stream, which has the potential to be further improved by the instruction set architecture in modern CPUs. Specifically, we propose a novel fast Subwindow-based Parallel Rolling hash method (**called SWPR**), which combines the techniques of Gear with SIMD and aims to achieve much higher hashing speed.

An useful idea is to divide the item to be detected into segments and calculate in parallel between these segments, which is proposed in SS-CDC [22]. However, such coarse-grained parallel methods may suffer from the 'boundary-shift' issue, resulting in a decrease in the accuracy of resemblance detection. To solve this problem, SS-CDC makes each segment contain some extra bytes ($rolling\_window\_size - 1$) to avoid losing chunking points. The extra overhead introduced by this method is negligible in file-level chunking algorithms, but is hard to ignore in chunk-level resemblance detection. In addition, we hope to further reduce the memory access frequency to better benefit from SIMD technology. Therefore, we need a more high-performance, parallel approach that maintains high resemblance detection accuracy.

An intuitive idea is to evenly divide the item to be detected into four interleaved groups (bytes that are three positions apart belong to the same group). Then process Gear hash on each group independently and simultaneously using four sliding windows (of the same size). However, directly applying Gear hash to this parallel hash method can cause severe problems.

First, Equation 10 states that Gear needs to access memory frequently, which will seriously affect the parallel execution efficiency of our algorithm. Specifically, before performing a hash calculation, the Gear-based algorithm must read the contents of each byte and then seek the corresponding mapping value in the Gear array, which requires each calculation to wait for two memory accesses to proceed. This part is difficult to parallelize, leading to a substantial decrease in the overall performance of our algorithm. Second, this method has an inherent weakness that leads to low efficiency in "information extraction". For example, in Gear, the characteristic generated by an 8-byte window contains the information of eight bytes $B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7$. But in this method, each sliding window rolls 4 bytes at a time. Thus the characteristic generated at the same position only contains the information of B_3 and B_7 (the other three windows contain $\{B_0, B_4\}$, $\{B_1, B_5\}$ and $\{B_2, B_6\}$ respectively), which means that the information extraction efficiency is only about $\frac{1}{4}$ of the original, which reduces the resemblance detection accuracy.

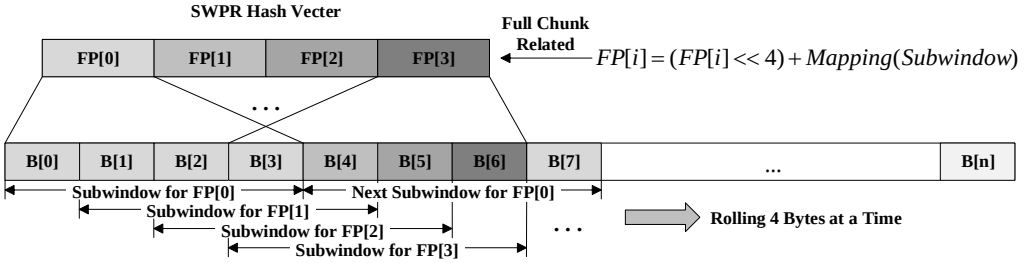


Fig. 8. General description of SWPR's principle (the case of four subwindows), note that each "FP" in SWPR Hash Vector is an 32-bit integer, and "B" represents "Byte". The function Mapping() is given in Equation 11.

For the first problem, we redesign the workflow of SWPR by using the subwindow rather than a single byte for Gear hashing. Specifically, SWPR divides each window into several subwindows

of equal size (the size is the same as the number of parallel sliding windows). As the example in Figure 8, each 32-byte sliding window used by Gear is divided into 8×4-byte subwindows in our four-window parallel algorithm. We let the four windows slide simultaneously, and each window is staggered by one byte and slides one subwindow size byte at a time. Although subwindows of different sliding windows in SWPR are overlapped, we can assign the corresponding content to all windows at the same time by rearranging the bytes in memory to avoid repeatedly accessing the same data for different subwindows. For example, assuming the subwindow size is set to 4 bytes and the byte stream is as $b_1, b_2, ..., b_n$. When the four subwindows slide to positions $b_i, b_{i+1}, b_{i+2}, b_{i+3}$ respectively, the contents of the subwindows are now $\{b_{i-3}, b_{i-2}, b_{i-1}, b_i\}$, $\{b_{i-2}, b_{i-1}, b_i, b_{i+1}\}$, $\{b_{i-1}, b_i, b_{i+1}, b_{i+2}\}$ and $\{b_i, b_{i+1}, b_{i+2}, b_{i+3}\}$. If we still follow the byte-by-byte memory access method in N-Transform, we need at least 7 memory accesses. In comparison, N-Transform only needs 4 memory accesses to process the sliding windows at the same positions. To reduce the number of memory accesses, we access more consecutive bytes at a time and combine the last accessed bytes (i.e., $\{b_{i-4}, b_{i-3}, b_{i-2}, b_{i-1}\}$ in this case) with the newly accessed bytes (i.e., $\{b_i, b_{i+1}, b_{i+2}, b_{i+3}\}$ in this case). Then we assign the corresponding contents to the four subwindows at the same time. In this way, we need to cache the last accessed bytes used in the following hash calculation after the sliding process to reduce the memory accesses to only one. Thus, SWPR achieves fewer memory accesses at the cost of a few bytes of extra space by reading several bytes at a time and rearranging them in memory.

For the second problem, in order to make the generated hash values contain more bytes of content, we change the mapping method of the subwindow. The new mapping value will be calculated directly by the content of the subwindow and no longer need to query the Gear array. By performing Gear hashing on the mapping value set of all subwindows, our method achieves higher efficiency in "information extraction". Because the length of the mapping value to be generated is the same as that of the subwindow, we can directly use a simple mapping method. Specifically, we concatenate the 4 bytes within the subwindow together to get a 32-bit mapping value, so that a subwindow consisting of four bytes is mapped as

$$Mapping(\{b_i, b_{i+1}, b_{i+2}, b_{i+3}\}) = (b_i \ll 24) + (b_{i+1} \ll 16) + (b_{i+2} \ll 8) + b_{i+3}. \tag{11}$$

Since we can obtain the mapping value directly in the process of accessing the bytes, no additional calculation operation is required, which further reduces the overhead of calculating characteristics.

SWPR uses 4 32-bit integers as hash values (packed into a 128-bit data vector) corresponding to four subwindows respectively, which shifts 4 bits at a time, to generate characteristics. From the example in Figure 8, the SWPR hash vector is used to hold the four hash values. The subwindows currently processed by the four sliding windows are $B[0] \sim B[3]$, $B[1] \sim B[4]$, $B[2] \sim B[5]$ and $B[3] \sim B[6]$, while the next subwindow corresponding to FP[0] is $B[4] \sim B[7]$. In this way, SWPR achieves approximately “byte-by-byte” parallel calculation (fine-grained). For each subwindow, we use the predefined sampling condition to generate a proxy of characteristics by Content-Defined Sampling (introduced in Section 3.2) with the sampling condition ‘$fp \geq boundary$’, which is easier to implement and more flexible in adjusting the sampling rate ‘sr’. We define the sampling condition as ‘$boundary = Max_{int} - (Max_{int} - Min_{int} + 1) \times sr$’, in which ‘$Max_{int}$’ and ‘$Min_{int}$’ respectively represent the maximum and minimum values that can be accommodated by the integer type used. For example, if we define the condition of ‘$boundary = 223$’ (i.e., $255 - 32$) for an 8-bit unsigned integer, the sampling rate is approximately equal to $\frac{1}{8}$ in the Content-Defined Sampling, which is determined by the ratio of the interval greater than $boundary$ to the range of integer values.

Finally, compared with Gear, our SWPR method requires fewer memory accesses and provides better support for parallel calculation, and thus achieves much faster speed. Moreover, our method generates characteristics at almost an identical level to that of Gear (namely fine-grained style

Characterizing) during the resemblance detection, so according to Broder's Theorem, our method can theoretically provide detection accuracy comparable to Gear's. There may be a few bytes (less than 16) discarded at the end of the item, due to misalignment, but the following experiments will demonstrate that the impact of this loss on the chunk-level resemblance detection accuracy is negligible.

We refer to our overall resemblance detection approach as "Odess" and the variant of Odess using SWPR optimization as "Odess+" to distinguish them in the following Sections. Overall, utilizing SWPR in the feature generation of resemblance detection (Odess+) is a better approach than utilizing Gear Hash, with faster speed and almost the same resemblance detection ability.

3.5 Implementation

Combining with SWPR (in the *Characterizing* step) and Content-Defined Sampling (in the *Selecting* step), we propose a new resemblance detection approach, named Odess, with a much higher speed for generating sequence-sensitive features while maintaining a high detection accuracy. In general, **the principle of Odess is to quickly generate a smaller proxy for each item and run independent repeated trials on the proxy with different linear transforms**.

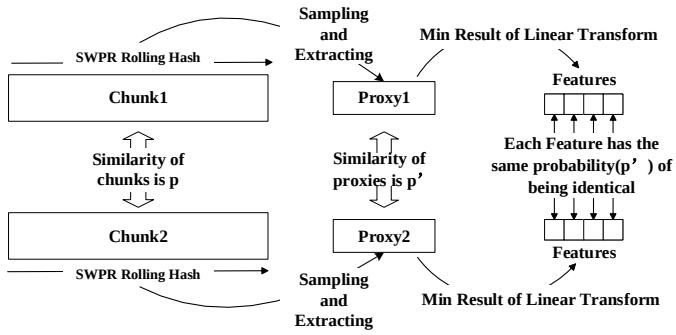


Fig. 9. General description of Odess's principle (SWPR+Content-Defined Sampling).

The general principle of Odess is shown in Figure 9, also following the typical three steps:

(1) **Characterizing:** Odess introduces SWPR which is a rolling hash implemented with SIMD. SWPR is fine-grained and can quickly collect characteristics (i.e., all the rolling hash values extracted from the sliding windows) for to-be-detected items.
(2) **Selecting:** Odess generates a proxy with Content-Defined Sampling for each item. After that, several predefined linear transforms are applied on the proxy to reduce calculations, and the minimum values of each linear transform are selected as features.
(3) **Indexing:** The Super-Feature technique is used[15], so several features are grouped into fewer Super-Features to detect very similar items.

Compared with N-Transform, the calculation of Odess is reduced substantially. In the *Characterizing* step, SWPR requires approximately half the calculations of Rabin [34], and it is further accelerated by SIMD. In the *Selecting* step, the calculation is reduced by the sampling rate in generating Proxies. The sampling rate is usually configured as 1/128, so the *Selecting* step of Odess has approximately only 1/128 of the calculation overheads of N-Transform. The implementation details of each sliding window of Odess are as shown in Algorithm 1.

Algorithm 2 shows how Odess+ processes 4 sliding windows in parallel simultaneously. Odess+ uses a vector containing four hash values for feature calculation, and each hash value corresponds to a parallel sliding window. At the beginning of the execution, 16 bytes of data are loaded into a

Algorithm 1: Extracting features for a single sliding window of Odess.

Input: chunk content, Str; length of the chunk, L; randomly value pair for linear transformation, m_i and a_i; a boundary to sample, *boundary*; number of the sliding windows, k ($k \in [0, 3]$)
Output: An output array of N features, Feature[0,...,N-1]

```
Feature[0, ..., N − 1] ← 0;
n ← 0;
while n + 16 < L do
    pos ← n + k − 3; //Str[pos]=0, if pos ≤ 0
    for m = 0 to 3 do
        v ← {Str[pos], ..., Str[pos + 3]}
        FP ← (FP ≪ 4) + Mapping(v); //GearFunction in SWPR
        if FP ≥ boundary then
            for i = 0 to N − 1 do
                Transform[i] ← (m_i ∗ FP + a_i) mod 2^32;
                if Feature[i] ≥ Transform[i] then
                    Feature[i] ← Transform[i];
        pos ← pos + 4;
    n ← n + 16;
return Feature;
```

temporary vector by the loading instruction (Load function), and converted into 4 intermediate vectors according to a transformation matrix (Shuffle function). Each intermediate vector (tmpVector) contains the contents of the 4 subwindows required, which belong to the 4 parallel sliding windows. This allows us to perform four vector operations after single memory access, minimizing the impact of loading data. **SIMD allows us to use 1 vector operation to complete the processing of 4 parallel sliding windows**, for example, "*_mm_slli_epi*32" instruction will shift the 4 hash values in the vector to the left at the same time, which enables our method to be parallelized. We also rearrange the order of computation operations to group the vector computations of the same type together to benefit from the instruction pipeline.

Algorithm 2: Extracting features with Odess+.

Input: chunk content, Str; length of the chunk, L;
Output: An output array of N features, Feature[0,...,N-1]

```
Feature[0, ..., N − 1] ← 0;
hashVector ← {0, 0, 0, 0}; //16-byte vector, storing the hash values of the 4 subwindows
predata ← {0, 0, 0, 0}; //16-byte vector, caching last loaded bytes
n ← 0;
while n + 16 < L do
    data ← Load(Str + n, 16); //access 16 consecutive bytes
    //a memory load of 16 bytes is followed by 4 hash calculations
    for m = 0 to 3 do
        tmpVector ← Shuffle(data, predata, m); //assign content to 4 subwindows
        hashVector ← SWPRHashing(hashVector, tmpVector); //simultaneous hashing
        Feature[0, ..., N − 1] ← Sampling(hashVector); //content-defined sampling
        predata ← data;
    n ← n + 16;
return Feature;
```

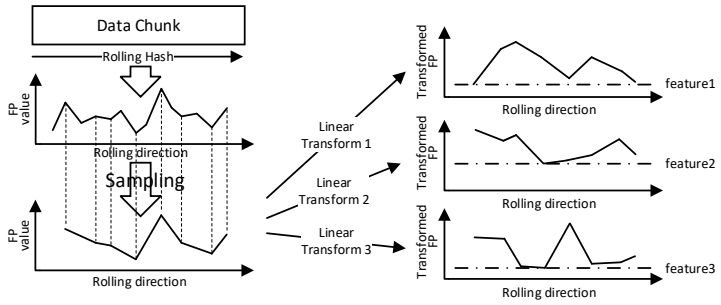


Fig. 10. An example of extracting three features in Odess.

In the *Characterizing* step (SWPRHashing function), the SIMD instructions are used to directly perform the hash calculation on the vector with the subwindow as the basic unit. Due to the characteristics of SIMD technology, each sliding window (e.g., $\#k, k \in [0, 3]$) corresponds to only one part of the vector (each part of the 128-bit vector is a 32-bit integer). Thus, operations on vectors are equivalent to processing four sliding windows simultaneously. In the *Selecting* step (Sampling function), since the characteristics generated in Odess are stored in the register in the form of a vector, we can also use the comparing instruction to sample for four sliding windows simultaneously to generate proxies for the to-be-detected items. Content-Defined Sampling in generating a proxy introduces more calculations (i.e., Content-Defined Sampling), but it is offset by the advantage of generating the smaller proxy. Figure 10 also provides an example of extracting three features in Odess, which suggests our Content-Defined Sampling can reduce the calculation of linear transforms in generating sequence-sensitive features.

4 EVALUATION

In this section, we first evaluate the properties of Odess using an artificial benchmark where we carefully control the similarity of synthesized data. From this study we verify that our approach reaches its design goals : a much faster resemblance detection than N-Transform with a better detection accuracy and a higher compression ratio than Finesse. Then, we perform an overall comparison between Finesse, N-Transform, and Odess in a prototype data reduction system combining deduplication and delta compression.

4.1 Experimental Setup

We perform our evaluation on a server with two Intel Xeon Gold 6113 processors @ 2.1G HZ, 128GB memory, and four Intel S4600 1TB SSDs. The server runs Ubuntu 16.04, and the prototype data reduction system has a five-stage pipeline, including reading, chunking, hashing, deduplicating, and writing stages. For the consideration of CPU compatibility (including older CPUs and AMD's CPUs), we use the SSE[1] instruction set to implement our parallel hash algorithm (i.e., SWPR). In order to keep the window size consistent with previous works (requiring 32-bit rolling hash), we implement SWPR with 4 parellel subwindows, since SSE's maximum vector size is 128 bits.

In experiments, we built a benchmark to generate data chunk pairs with varying similarity distributions to study the detection accuracy of Odess (detailed in Section 4.2). Then we also build a prototype data reduction system (deduplication + delta compression) to demonstrate the impact of Odess at a system level (detailed in Section 4.3).

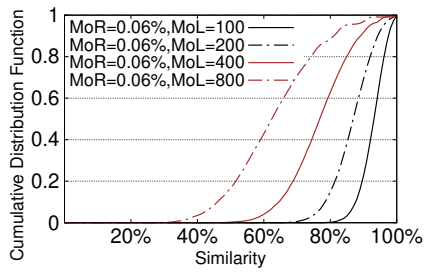
Table 2. Workload characteristics of the six tested datasets in our evaluation. Dedup Ratio is reported with an average chunk size of 8KB.

Name	Size	Dedup Ratio	Workload descriptions
TAR	167 GB	2.00	492 tarred files from several open source projects such as GCC, Emacs, GDB, OpenJDK, Node.js, etc.
LNX	116 GB	1.93	265 versions of Linux kernel source code. Each version is packaged as a tar file.
RDB	535 GB	10.95	99 backups of the redis key-value store database.
SYN	497 GB	21.96	75 synthetic backups by simulating file create/delete/modify operations.
VMA	138 GB	1.62	91 virtual machine images of different OS release versions, including Fedora, CentOS, Debian, etc.
VMB	314 GB	7.73	21 backups of an Ubuntu 12.04 VM image.

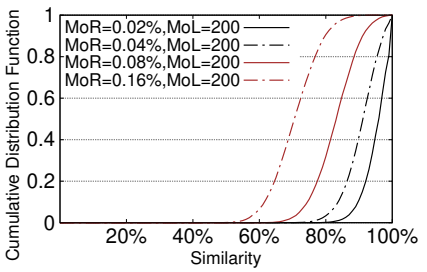
For the prototype system experiments, six datasets are used as shown in Table 2. These datasets represent various typical backup workloads for redundancy elimination, including website snapshots, source code projects, virtual machine images, database images, and a synthetic dataset. These datasets have been used in several data reduction studies [21, 35, 39].

4.2 Detection Accuracy on Benchmark Datasets

In this subsection, we evaluate four resemblance detection approaches on a benchmark, focusing on the detection accuracy as measured by two metrics, **Average Estimating Error** (lower is better) and **Standard Deviation of Estimating Error** (lower is better). **Estimating Error** is the absolute value of the difference between the actual similarity (recorded by benchmark) and the estimated similarity (calculated after detecting by the tested approaches with Equation 2). Average Estimating Error shows the general detection accuracy of approaches and Standard Deviation of Estimating Error shows the stability of approaches.



(a) Similarity distribution of the first series of datasets with MoR=0.06% and varied MoL



(b) Similarity distribution of the second series of datasets with MoL=200 and varied MoR

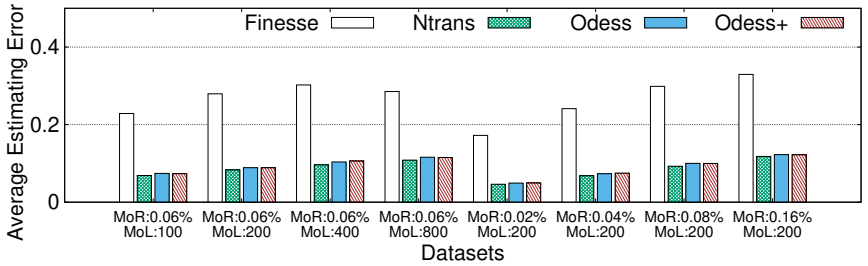
Fig. 11. Similarity distribution of generated pairs of similar chunks in two series of benchmark datasets generated by varying the **MoL** and **MoR**. The similarity represents the actual similarity between pairs of similar chunks which is recorded by benchmark.

In order to better evaluate detection accuracy as similarity varies in a controlled manner, we create a benchmark instead of using existing datasets. We generate a number of chunks consisting of random data and apply a series of modifications on each chunk to derive pairs of similar chunks in our benchmark, in which the actual similarities of these pairs of chunks are recorded. Modifications randomly occur on each byte with a probability that we call the **"modification rate"** (denoted by **MoR**). Each modification will perform an insertion, a deletion or a replacement with an equal probability, and we call the length of insertion, deletion or replacement the **"modification length"** (denoted by **MoL**).

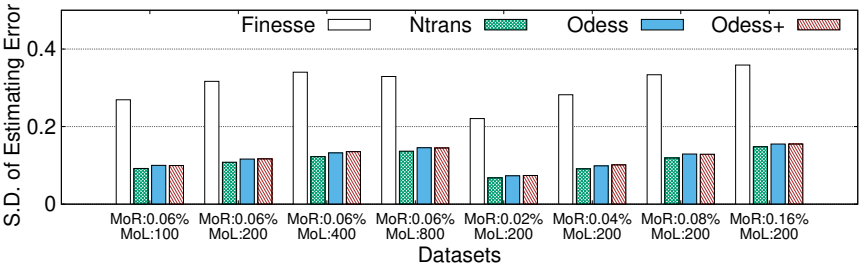
Two series of benchmark datasets are generated by varying the **MoL** and **MoR** as illustrated in Figure 11(a) and Figure 11(b). Adjusting the modification length has a larger impact on similarity distributions, causing them to be more spread than changing the modification rate.

4.2.1 ***Comparison among Four Resemblance Detection Approaches.*** We compare the resemblance detection accuracy of four approaches: Finesse, N-Transform, Odess (i.e., **Gear Hash + Content-Defined Sampling**), and Odess+ (i.e., **SWPR + Content-Defined Sampling**). Note that the last three approaches generate sequence-sensitive features while Finesse generates sequence-insensitive features.

Each approach estimates the similarity degree of pairs of similar chunks in its own way. For N-Transform, Odess, and Odess+, positional matching is enough, but for Finesse, a sorting is required before positional matching, as introduced in Section 2.2.



(a) Average Estimating Error



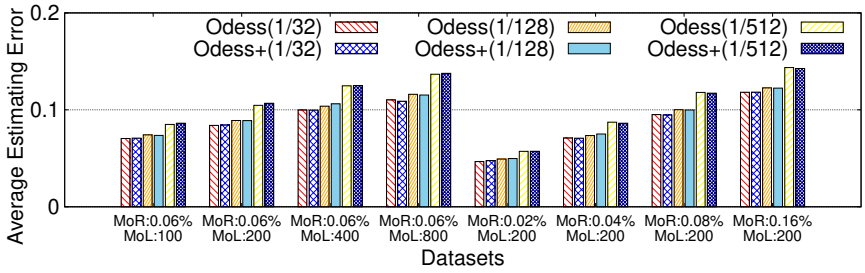
(b) Standard Deviation of Estimating Error

Fig. 12. Comparison on detection accuracy of the four approaches on benchmark datasets. The sampling rate of Odess and Odess+ is 1/128.

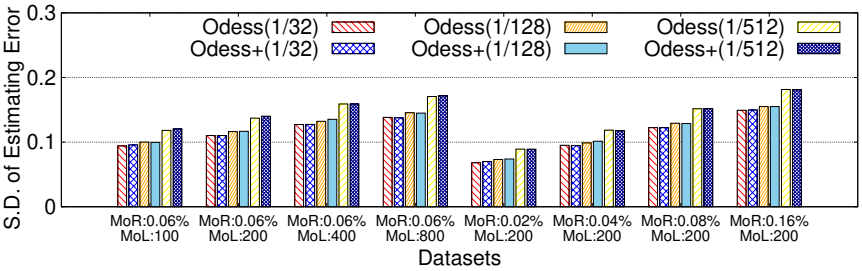
Figure 12 compares the detection accuracy of four approaches on these two series of benchmark datasets. The sampling rate for Odess and Odess+ is configured to 1/128. Each approach is set to generate the same number of Super-Features (i.e., 3). All four approaches show a similar pattern of increasing error as MoR and MoL increase. Odess+ achieves very similar results to Odess and N-Transform in Average Estimating Error and Standard Deviation of Estimating Error, and all three significantly lower Average Estimating Error than Finesse. For Finesse, when MoL increases to 800, the relationship between the size of MoL and Finesse's subchunk is subtle. Some combinations of adjusting the chunk size will align with existing sub-chunk boundaries (reduced Average Estimating Error) while other adjustments cause a 'boundary-shift' problem [20] (increased Average Estimating Error).

Therefore, the **Inference 1** in Section 3.5, "compared with N-Transform, our Content-Defined Sampling approach will generate the same similarity on average", is verified in Figure 12. Meanwhile,

Figure 12 also verifies that our approach satisfies one of the design goals: both Odess and Odess+ have a much higher detection accuracy than Finesse.



(a) Average Estimating Error



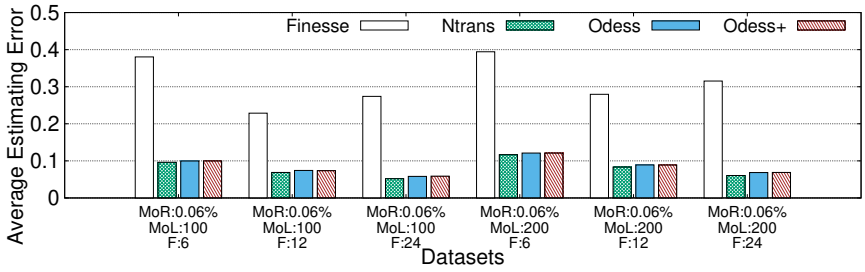
(b) Standard Deviation of Estimating Error

Fig. 13. Comparison on the detection accuracy of Odess and Odess+ with different sampling rates on benchmark datasets.

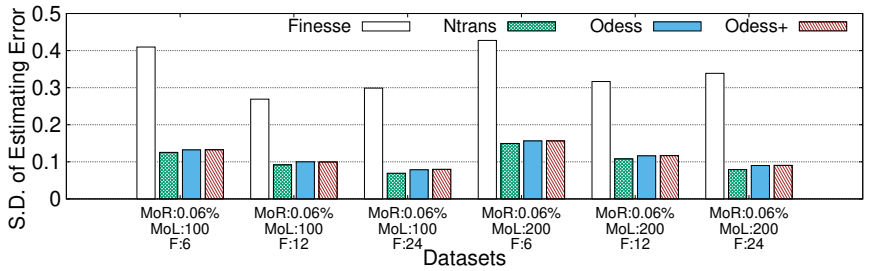
4.2.2 ***Impact of Odess's Sampling Rate.*** We explore the impact of the sampling rate in Odess and Odess+, and the result is shown in Figure 13. In general, both Odess and Odess+ have stable results across sampling rates, though error does increase for less frequent sampling rates (1/512). A more frequent sampling rate leads to smaller Average Estimating Error and Standard Deviation of Estimating Error, which means a better robustness; It is consistent with the Figure 7 and shows that 1/128 is a reasonable choice for sampling rates. Therefore, **Inference 2** in Section 3.5, "the sampling rate decides the robustness of our approach", is also confirmed.

4.2.3 ***Impact of the Number of Features.*** Figure 14 compares the four approaches when generating different numbers of features. We find that Odess and Odess+ perform better when the number of features increases and both have nearly the same Average Estimating Error and Standard Deviation of Estimating Error with N-Transform. This is because adding more features increases the detection accuracy of these three approaches. But Finesse does worse as the number of features increases from 12 to 24, which is because Finesse splits chunks into a subchunk per feature and more features requires smaller subchunks. It makes the 'boundary-shift' problem more serious, which outweighs the benefits of increasing the number of features.

In general, whether changing the sampling rate of the Content-Defined Sampling approach or changing the number of features, Odess+ always has almost the same detection accuracy as Odess. It confirms that, "Compared to Odess only using Content-Defined Sampling, our Subwindow-based parallel rolling hash method using SIMD will yield the same resemblance detection accuracy on average, while greatly improving the hashing speed".



(a) Average Estimating Error



(b) Standard Deviation of Estimating Error

Fig. 14. Comparison on the detection accuracy of the four approaches with different feature numbers on benchmark datasets. The Sampling rate of Odess and Odess+ is 1/128.

4.3 Experiments in Prototype Data Reduction System

In this subsection, we apply our experiments on a prototype data reduction system combining deduplication and delta compression.

Table 2 lists the used datasets in this subsection. We implement a post-deduplication delta compression prototype system as an experimental platform. The general workflow of our prototype system is shown in Figure 15, consisting of five parallel pipeline stages(i.e., reading, chunking, hashing, deduplicating and writing). Deduplication is designed to reduce the chunk-level redundancy, and delta compression is designed to reduce redundancy between pairs of very similar but not identical chunks.

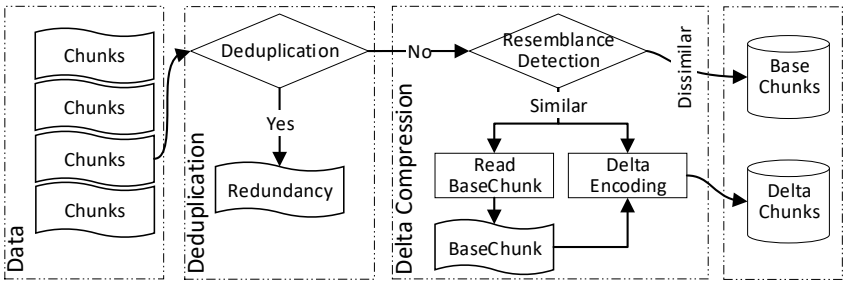


Fig. 15. General workflow of a prototype data reduction system.

We use FastCDC [35] to split data streams into chunks (default average size is 8KB). In the hashing pipeline, each chunk generates a strong hash value (e.g., SHA1) as a unique identifier, called a fingerprint. Deduplication and delta compression are performed sequentially in the duplicating

pipeline. It maintains three main data structures, 1) a fingerprint map for identifying duplicate data, 2) a super-feature store for detecting similarity, and 3) a cache for exploiting locality. In delta compression, we modified Xdelta3 [18] to only perform data reduction across similar chunks and no longer perform inner data reduction on the input chunks to better show the impact of similarity-detection and remove the impact of local compression (such as LZ [42]).

The four resemblance detection methods all create 12 features and group them into 3 Super-Features, which is recommended by previous work [15, 17, 27]. A chunk may have multiple similar chunks, and we select the first matched chunk as its base chunk for delta compression, which is also known as 'FirstFit' [15]. The window size of the rolling hash in these approaches is configured to 32 bytes. In Finesse [39], detected chunks with considerably different sizes will be regarded as dissimilar chunks to avoid the detection weakness, but other works do not filter size mismatches. Thus, in this part, we avoid this limit for a high compression ratio.

4.3.1 ***Data Reduction Efficiency.*** First, we focus on the data reduction efficiency of the system when applying different approaches with different chunk sizes. We evaluate this experiment with three metrics, **Delta Compression Ratio (DCR)**, **Delta Compression Efficiency (DCE)** and **Similar Chunk Ratio (SCR)**. DCR reflects the total space saved by resemblance detection and then delta compression:

$$DCR = \frac{total\ size\ before\ delta\ compression}{total\ size\ after\ delta\ compression}. \tag{12}$$

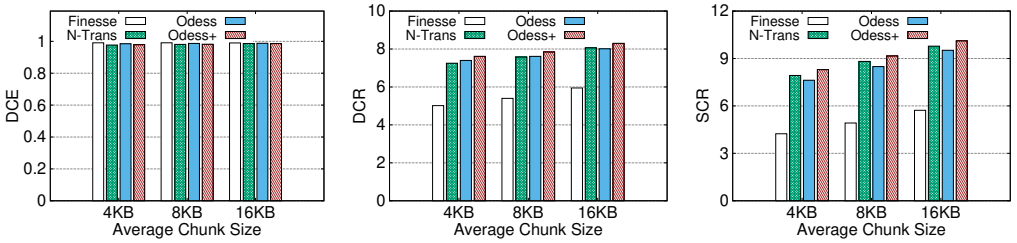
DCE is used to estimate the average similarity degree between the detected similar chunks:

$$DCE = Avg.(1 - \frac{chunk\ size\ after\ delta\ compression}{chunk\ size\ before\ delta\ compression}). \tag{13}$$

SCR is used to study how many similar chunks will be detected by the tested approaches:

$$SCR = \frac{the\ number\ of\ detected\ similar\ chunks}{the\ number\ of\ unique\ chunks\ (dissimilar)}. \tag{14}$$

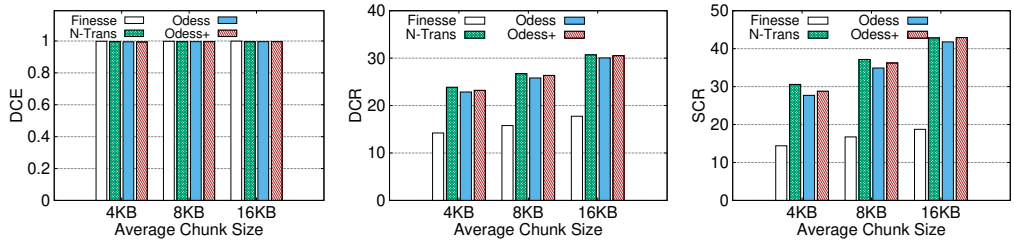
We mainly focus on DCR, namely Delta Compression Ratio, which directly represents the amount of space the delta compression method can save. The other two metrics are meant to explain changes in DCR to identify the strengths and weaknesses of the approaches. In general, the higher the three metrics are, the better the approach is at savings space.



(a) DCE (b) DCR (c) SCR

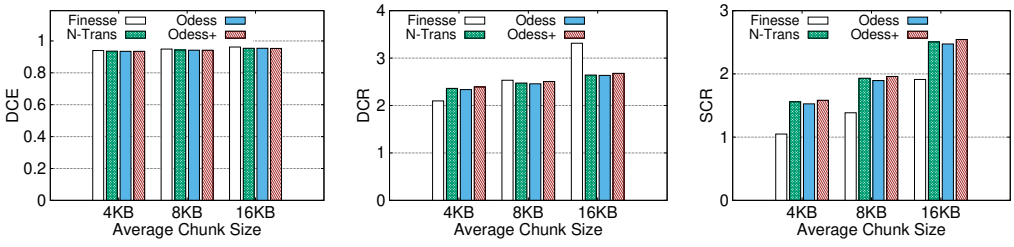
Fig. 16. Comparison of resemblance detection efficiency among the four approaches on the TAR dataset.

Figures 16, 17, 18, 19, 20, and 21 show the DCE, DCR, and SCR of all four approaches (i.e., Finesse, N-Transform, Odess and Odess+) on six datasets, with varying chunk sizes (Note the different scales of the y-axis for DCR and SCR between different datasets). N-Transform is presented as the baseline. Finesse usually reaches a higher DCE but a lower DCR and SCR, which means Finesse finds very similar chunks but misses less similar chunks. Figure 18 shows different results because



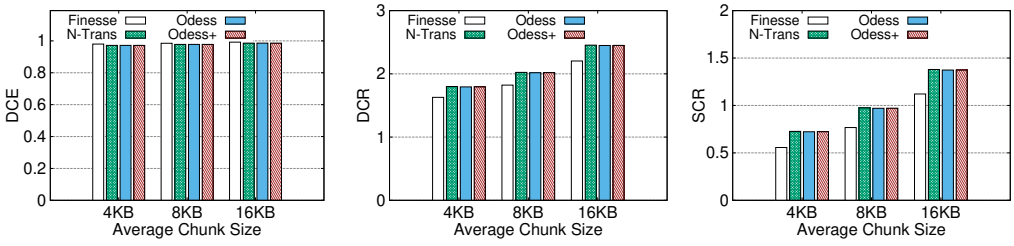
(a) DCE (b) DCR (c) SCR

Fig. 17. Comparison of resemblance detection efficiency among the four approaches on the LNX dataset.



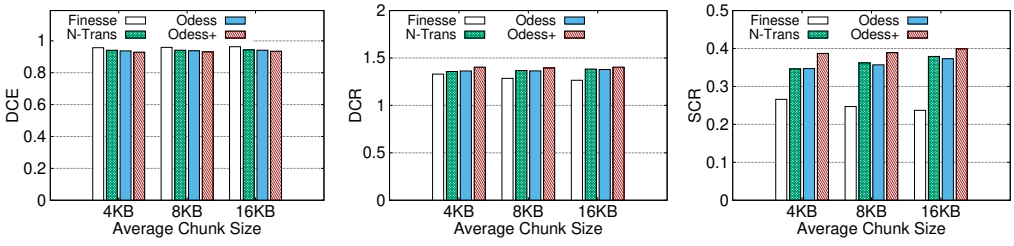
(a) DCE (b) DCR (c) SCR

Fig. 18. Comparison of resemblance detection efficiency among the four approaches on the RDB dataset.



(a) DCE (b) DCR (c) SCR

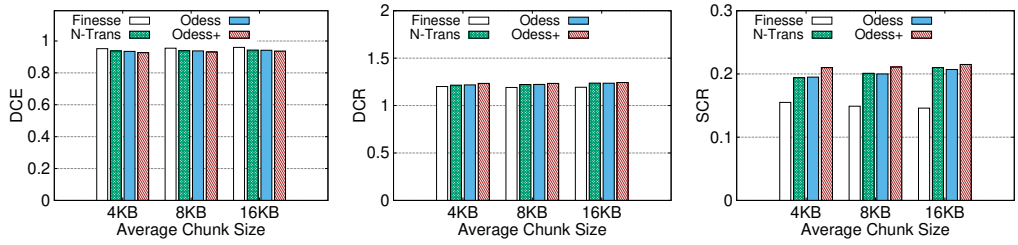
Fig. 19. Comparison of resemblance detection efficiency among the four approaches on the SYN dataset.



(a) DCE (b) DCR (c) SCR

Fig. 20. Comparison of resemblance detection efficiency among the four approaches on the VMA dataset.

RDB dataset's backup stream is highly localized, which is in line with Finesse's design idea as discussed in Section 2.4. As shown by other data sets, this situation is rare in real data backup scenarios. Thus the weakness of Finesse is obvious, and it can not realize the potential of delta compression as much as possible.



(a) DCE (b) DCR (c) SCR

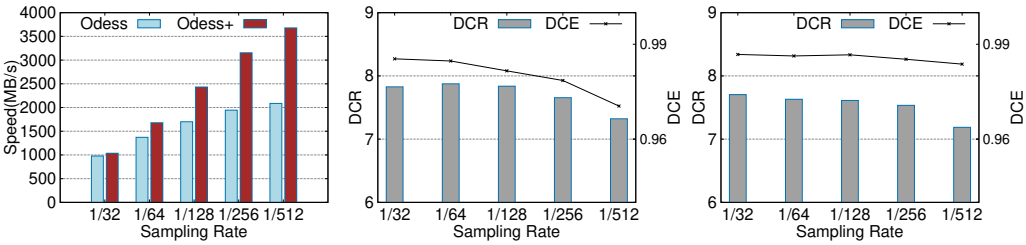
Fig. 21. Comparison of resemblance detection efficiency among the four approaches on the VMB dataset.

N-Transform, Odess and Odess+ find more similar chunks, which can save more storage space. For these three approaches, DCE and DCR usually increase with the chunk size, because a bigger chunk size has more similar but not identical chunks [28]. Odess+ has slightly lower DCE and finds more similar chunks than Odess, thus Odess+ achieves a slightly higher DCR. In general Odess+ and Odess achieve approximately the same data reduction efficiency.

Compared with Finesse, Odess+ achieves 1.246× DCR for the 4KB chunk size, 1.224× DCR for the 8KB chunk size, and 1.198× DCR for the 16KB chunk size, which is a very similar result to N-Transform and Odess.

4.3.2 ***Impact of Sampling Rate in Odess+ on the Speed and Detection Accuracy.*** This experiment runs on the TAR dataset, and the chunk size is configured as 8KB.

Figure 22(a) compares the feature generating speed of Odess and Odess+ using different sampling rates. The smaller the sampling rate is, the lower the computational overhead and the higher the feature generating speed will be. At the upper limit of sampling, Odess nearly reaches the performance of Gear, which is the feature generation algorithm used before sampling and linear transforms are applied. Compared with Odess, Odess+ achieves 1.06× ∼ 1.76× feature generating speed with the increase of sampling rate. The reason is that the increase of sampling rate reduces the time proportion of linear transforms in the algorithm, which makes the improvement of the parallel rolling hash approach more obvious.



(a) Features generating speed (b) DCR & DCE of Odess+ (c) DCR & DCE of Odess

Fig. 22. Impact of sampling rate in Odess and Odess+.

Figure 22(b) shows the impact of Odess+'s sampling rate on DCR and DCE on the TAR dataset. Usually, DCR is affected with the same trend as DCE by the sampling rate, but this is not absolute. DCR is affected by both DCE and SCR, resulting in DCR sometimes slightly lower than the expected value, like the point with sampling rate 1/32 in Figure 22(b). But overall, a more frequent sampling rate (e.g., 1/64) will lead to a higher DCE and DCR, which means a higher resemblance detection

efficiency, and a less frequent sampling rate (e.g., 1/512) does the opposite. These results are also consistent with Figure 7.

4.3.3 ***Impact of the Number of Parallel windows in Odess+.*** This experiment runs on the TAR dataset in Table 2, the chunk size is configured as 8KB, and the sampling rate is set to 1/128.

Table 3. The impact of configuring different number of parallel windows in Odess+ on feature generation speed and data reduction effiency.

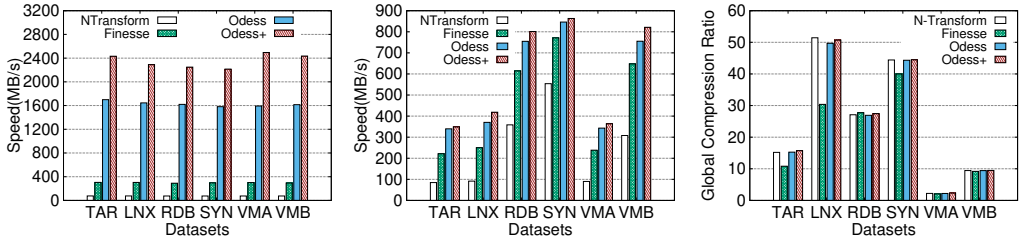
Configuration	Hash Value Size	DCE	DCR	SCR	Speed
2 Windows in Parallel	64 Bit	0.989	7.364	7.936	1532 MB/s
4 Windows in Parallel	32 Bit	0.982	7.845	9.163	2129 MB/s
8 Windows in Parallel	16 Bit	0.846	5.911	58.896	2890 MB/s

Table 3 compares the feature generating speed and the delta compression ratio (DCR) of Odess+ under different parallel window number configurations. The higher the number of windows for parallel calculation, the faster the feature generation speed will be. However, limited by the maximum vector length of 128 bits, the method with 8 parallel windows has to use 16-bit hash values for feature computation. As mentioned in Section 3.4, the subwindow of the approach with 8 parallel windows is 8 bytes (the subwindow size is the same as the number of parallel windows). So we must map the content of each subwindow to a 16-bit hash value in the calculation, which leads to the information extraction efficiency dropping significantly, making DCR lower than the approach with 4 parallel windows. After considering all of the factors, we feel that the configuration with 4 parallel sliding windows achieves more reasonable results for Odess+.

4.3.4 ***Performance of the Whole System on Six Datasets.*** We first study the feature generating speed in Figure 23(a) for N-Transform, Finesse, Odess, and Odess+ on six datasets. Odess and Odess+ use a sampling rate of 1/128 based on previous experiments. Results show that N-Transform is the slowest, Odess+ runs about 29.6× ∼ 33.5× faster than N-Transform (average is 31.4×), about 7.5× ∼ 8.3× faster than Finesse (average is 7.9×) and about 1.39× ∼ 1.57× faster than Odess (average is 1.45×). Summarizing our results, Odess+ demonstrates a large improvement in the feature generation speed.

Next, we study the influence of resemblance detection approaches on the throughput of the data reduction system in Figure 23(b). The system with Odess+ shows a significant advantage, achieving about 1.11× ∼ 1.67× faster throughput than the system with Finesse (average is 1.41×), and about 1.55× ∼ 4.57× faster than the system with N-Transform (average is 3.20×). Note that only non-deduplicated chunks will be processed by a resemblance detection method (as shown in Figure 15), and from Table 2 we see that the deduplication ratios of RDB, SYN, and VMB are much higher than other datasets. Therefore, the throughput advantage of Odess+ is smaller in RDB, SYN, and VMB datasets. Besides, the content-based sampling method can greatly reduce the time overhead of resemblance detection, which makes the performance bottleneck of resemblance detection less obvious. Therefore, the improvement of system throughput of Odess+ is limited compared to Odess. In Section 4.3.1, we learn that Finesse identifies fewer similar chunks than N-Transform and Odess+, and the system with Finesse also performs less delta encoding. But even if the system with Finesse has less computational overheads in delta encoding, the system with Odess+ still has higher throughput.

We also study the Global Compression Ratio (defined as $\frac{total\ size\ before\ compression}{total\ size\ after\ compression}$) in Figure 23(c). The system with Odess+ provides greater data reduction efficiency than Finesse, achieving about 1.23× on average, which is almost identical to the results for N-Transform. The results are consistent with those shown in Section 4.3.1.



(a) Features generating speed (b) System throughput (c) Global Compression Ratio

Fig. 23. Comparison between four approaches on resemblance detection and delta compression prototype systems with different datasets.

From experiments in this subsection, we learn that compared with Finesse, a prototype data reduction system with Odess+ improves throughput about 1.41× and increases the compression ratio about 1.22×; compared with Odess, a prototype system with Odess+ improves throughput about 1.06× and the compression ratio is almost unchanged; and compared with N-Transform, a prototype system with Odess+ improves throughput about 3.20× while achieving nearly the same compression ratio.

5 CONCLUSION AND FUTURE WORK

As the most popular resemblance detection approach, N-Transform introduces serious computational overhead in the feature-generating process of delta compression. In this paper, we propose Odess, a fast resemblance detection approach, which combines a fast Subwindow-based Parallel Rolling hash method using SIMD (called SWPR) and a novel Content-Defined Sampling method. The evaluation results show that, Odess's approach is about 31.4× faster than N-Transform in feature generation for resemblance detection while achieving comparable detection accuracy. We apply Odess in a prototype data reduction system of post-deduplication delta compression, and our experiments show that, compared with N-Transform, the prototype system with Odess is about 3.20× faster with nearly the same compression efficiency.

In the future, we will focus on improving the grouping method that is combined with our fast Content-Defined Sampling. The N-Transform approach can not specify a similarity threshold when detecting similar candidates, and chunks with lower similarity are hard to detect.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their insightful comments. This research was partly supported by the National Natural Science Foundation of China under Grant no. 61972441, no. 61972112 and no. 61832004; the Guangdong Basic and Applied Basic Research Foundation under Grant no. 2021B1515020088; Shenzhen Science and Technology Innovation Program under Grant no. RCYX20210609104510007, no. JCYJ20210324131203009, no. JCYJ20200109113427092, and no. GXWD20201230155427003-20200821172511002; Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005); and the HITSZ-J&A Joint Laboratory of Digital Design and Intelligent Fabrication under Grant no. HITSZ-J&A-2021A01. Xiangyu Zou(xiangyu.zou@hotmail.com) is the corresponding author.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Hossein Amiri and Asadollah Shahbahrami. 2020. SIMD Programming Using Intel Vector Extensions. *J. Parallel and Distrib. Comput.* 135 (2020), 83–100.

[2] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. 2009. The Design of a Similarity Based Deduplication System. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR'09)*. Association for Computing Machinery, Haifa, Israe, 1–14.

[3] Tony Asaro and Heidi Biggar. 2007. *Data De-Duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations*. Technical Report. The Enterprise Strategy Group, Newton, MA, USA. 2–15 pages.

[4] Andrei Z. Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of SEQUENCES 1997*. IEEE, Salerno, Italy, 21–29.

[5] Andrei Z. Broder. 1998. Identifying and Filtering Near-Duplicate Documents. *Lecture Notes in Computer Science* 1848 (1998), 1–10.

[6] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 2000. Min-Wise Independent Permutations. *J. Comput. System Sci.* 60, 3 (2000), 630–659.

[7] Fred Douglis and Arun Iyengar. 2003. Application-Specific Delta-Encoding via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'03)*. USENIX Association, San Antonio, TX, USA, 113–126.

[8] George Forman, Kave Eshghi, and Stephane Chiocchetti. 2005. Finding Similar Files in Large Document Repositories. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. Association for Computing Machinery, New York, NY, USA, 394–400.

[9] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. 2015. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (USENIX FAST'15)*. USENIX Association, Santa Clara, CA, USA, 331–344.

[10] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. 2008. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of 2008 USENIX Symposium on Operating Systems Design and Implementations (OSDI'08)*. Association for Computing Machinery, New York, NY, USA, 85–93.

[11] Arne Holst. 2021. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. *Statista, June* 1 (2021), 9 pages.

[12] David A Huffman. 1952. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[13] Paul Jaccard. 1901. Etude de la Distribution Florale Dans une Portion des Alpes et du Jura. *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (1901), 547–579.

[14] Navendu Jain, Michael Dahlin, and Renu Tewari. 2005. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proceedings of 4th USENIX Conference on File and Storage Technologies (FAST'05)*. USENIX Association, San Francisco, CA, USA, 21–35.

[15] Purushottam Kulkarni, Fred Douglis, Jason D LaVoie, and John M Tracey. 2004. Redundancy Elimination within Large Collections of Files. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'04)*. USENIX Association, Boston, MA, USA, 59–72.

[16] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (USENIX FAST'09)*. USENIX Association, San Francisco, CA, USA, 111–123.

[17] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. 2014. Migratory Compression: Coarse-Grained Data Reordering to Improve Compressibility. In *Proceedings of 12th USENIX Conference on File and Storage Technologies (USENIX FAST'14)*. USENIX Association, Santa Clara, CA, USA, 256–273.

[18] Josh MacDonald. 2000. *File System Support for Delta Compression*. Master's thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley.

[19] Dirk Meister, Jürgen Kaiser, and André Brinkmann. 2013. Block Locality Caching for Data Deduplication. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. Association for Computing Machinery, New York, NY, USA, 15:1–15:12.

[20] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-Bandwidth Network File System. *ACM SIGOPS Operating Systems Review* 35 (2001), 174–187.

[21] Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based Deduplication Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*. Association for Computing Machinery, New York, NY, USA, 220–232.

[22] Fan Ni, Xing Lin, and Song Jiang. 2019. 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*. Association for Computing Machinery, New York, NY, USA, 86–96.

[23] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. 2007. Exploiting Similarity for Multi-Source Downloads Using File Handprints. In *Proceedings of 4th USENIX Symposium on Network System Design and Implementation (NSDI'07)*. USENIX Association, Cambridge, MA, USA, 15–28.

[24] Sean Quinlan and Sean Dorward. 2002. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (USENIX FAST'02)*. USENIX Association, Monterey, CA, USA, 89–101.

[25] Michael O Rabin. 1981. *Fingerprinting by Random Polynomials*. Technical Report. Center for Research in Computing Technology, Harvard University.

[26] David Reinsel, John Gantz, and John Rydning. 2018. The Digitization of the World from Edge to Core. *IDC White Paper* 16 (2018), 28 pages.

[27] Phlip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. 2012. WAN-Optimized Replication of Backup Datasets Using Stream-Informed Delta Compression. *ACM Transactions on Storage* 8, 4 (2012), 1–26.

[28] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. 2012. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality. In *Proceedings of 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'12)*. USENIX Association, Boston, MA, USA, 1–6.

[29] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (USENIX FAST'12)*. USENIX Association, San Jose, CA, USA, 4.

[30] Avani Wildani, Ethan L Miller, and Ohad Rodeh. 2013. HANDS: A Heuristically Arranged Non-Backup in-Line Deduplication System. In *Proceedings of 39th IEEE International Conference on Data Engineering (IEEE ICDE'13)*. IEEE, Brisbane, QLD, Australia, 446–457.

[31] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710.

[32] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A Similarity-Locality Based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'11)*. USENIX Association, Portland, OR, USA, 26–30.

[33] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. 2016. DARE: A Deduplication-Aware Resemblance Detection and Elimination Scheme for Data Reduction with Low Overheads. *IEEE Trans. Comput.* 65, 6 (2016), 1692–1705.

[34] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. 2014. Ddelta: A Deduplication-inspired Fast Delta Compression Approach. *Performance Evaluation* 79 (2014), 258–272.

[35] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, Denver, CO, USA, 101–114.

[36] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R Ganger. 2017. Online Deduplication for Databases. In *Proceedings of 2017 ACM Conference on Management of Data (ACM SIGMOD'17)*. Association for Computing Machinery, New York, NY, USA, 1355–1368.

[37] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, Jin Li, and Gregory R Ganger. 2015. Reducing Replication Bandwidth for Distributed Document Databases. In *Proceedings of 2015 ACM Symposium on Cloud Computing (SoCC'15)*. Association for Computing Machinery, New York, NY, USA, 222–235.

[38] Lawrence L You, Kristal T Pollack, and Darrell DE Long. 2005. Deep Store: An Archival Storage System Architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, Tokyo, Japan, 804–815.

[39] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: Fine-Grained Feature Locality based Fast Resemblance Detection for Post-Deduplication Delta Compression. In *Proceedings of 17th USENIX Conference on File and Storage Technologies (USENIX FAST'19)*. USENIX Association, Boston, MA, USA, 121–128.

[40] Feng Zhou, Li Zhuang, Ben Y Zhao, Ling Huang, Anthony D Joseph, and John Kubiatowicz. 2003. Approximate Object Location and Spam Filtering on Peer-to-peer Systems. In *Proceedings of 2003 International Middleware Conference (Middleware'03)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[41] Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (USENIX FAST'08)*. USENIX Association, San Jose, California, USA, 269–282.

[42] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

[43] Jacob Ziv and Abraham Lempel. 1978. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.

[44] Daniel Zwillinger and Stephen Kokoska. 1999. *CRC Standard Probability and Statistics Tables and Formulae*. Crc Press, New York, NY, USA. 31–32 pages.