From Hyper-dimensional Structures to Linear Structures: Maintaining Deduplicated Data's Locality

XIANGYU ZOU and JINGSONG YUAN, Harbin Institute of Technology, Shenzhen, China
PHILIP SHILANE, Dell Technologies, USA
WEN XIA, Harbin Institute of Technology, Shenzhen, China and Wuhan National Laboratory for Optoelectronics, HUST, China
HAIJUN ZHANG and XUAN WANG, Harbin Institute of Technology, Shenzhen, China

Data deduplication is widely used to reduce the size of backup workloads, but it has the known disadvantage of causing poor data locality, also referred to as the fragmentation problem. This results from the gap between the hyper-dimensional structure of deduplicated data and the sequential nature of many storage devices, and this leads to poor restore and garbage collection (GC) performance. Current research has considered writing duplicates to maintain locality (e.g., rewriting) or caching data in memory or SSD, but fragmentation continues to lower restore and GC performance.

Investigating the locality issue, we design a method to flatten the hyper-dimensional structured deduplicated data to a one-dimensional format, which is based on classification of each chunk's lifecycle, and this creates our proposed data layout. Furthermore, we present a novel management-friendly deduplication framework, called MFDedup, that applies our data layout and maintains locality as much as possible. Specifically, we use two key techniques in MFDedup: Neighbor-duplicate-focus indexing (NDF) and Across-version-aware Reorganization scheme (AVAR). NDF performs duplicate detection against a previous backup, then AVAR rearranges chunks with an offline and iterative algorithm into a compact, sequential layout, which nearly eliminates random I/O during file restores after deduplication.

Evaluation results with five backup datasets demonstrate that, compared with state-of-the-art techniques, MFDedup achieves deduplication ratios that are 1.12× to 2.19× higher and restore throughputs that are 1.92× to 10.02× faster due to the improved data layout. While the rearranging stage introduces overheads, it is more than offset by a nearly-zero overhead GC process. Moreover, the NDF index only requires indices for two backup versions, while the traditional index grows with the number of versions retained.

CCS Concepts: • **Information systems** → **Deduplication**; **Data layout;**

This work was partly supported by the National Natural Science Foundation of China under Grants No. 61972441, No. 61972112, and No. 61832004; the Guangdong Basic and Applied Basic Research Foundation under Grants No. 2021A1515012634 and No. 2021B1515020088; the Shenzhen Science and Technology Program under Grants No. JCYJ20210324131203009, No. JCYJ20190806143405318, and No. JCYJ20200109113427092; the HITSZ-J&A Joint Laboratory of Digital Design and Intelligent Fabrication under Grant No. HITSZ-J&A-2021A01; and the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2018WNLOKF008. 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.

Authors' addresses: X. Zou, J. Yuan, H. Zhang, and X. Wang, Harbin Institute of Technology, Shenzhen, China; emails: xiangyu.zou@hotmail.com, js.yuann@gmail.com, hjzhang@hit.edu.cn, wangxuan@cs.hitsz.edu.cn; P. Shilane, Dell Technologies, 131 Pheasant Lane, Newtown, PA, 18940, USA; email: philip.shilane@dell.com; W. Xia (corresponding author), HIT Campus of University Town of Shenzhen, Shenzhen, 518055, China; email: xiawen@hit.edu.cn.

1553-3077/2022/08-ART25
https://doi.org/10.1145/3507921

Additional Key Words and Phrases: Fragmentation, restore, garbage collection

ACM Reference format:
Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. 2022. From Hyper-dimensional Structures to Linear Structures: Maintaining Deduplicated Data's Locality. *ACM Trans. Storage* 18, 3, Article 25 (August 2022), 28 pages.
https://doi.org/10.1145/3507921

1 INTRODUCTION

Deduplication is an important data reduction technique in modern commercial backup systems, because it usually achieves a high deduplication ratio (the logical size divided by the post deduplication size), which was found to often be in the range of 10×–30× [6], thus greatly reducing storage costs. The basic technique of deduplication is to replace redundant chunks of data with references to identical chunks that have already been stored [51]. While deduplication has been applied to numerous storage and networking topics [6, 17, 27, 37, 58], our research focuses on hard-drive-based deduplication for backup storage, because it remains one of the most significant use cases.

Maintaining locality for deduplicated data remains challenging. For standard (non-deduplicated) storage systems, maintaining locality in storage devices is easy and natural, because data in a standard storage system is organized sequentially and storage devices also store data in sequential order. For deduplicated systems, the situation is different. As illustrated in Figure 1, Chunk B is the common chunk for all three files, and if we wish to maintain the locality of these files after deduplication, we must ensure that Chunk B's surrounding chunks (i.e., Chunk A, Chunk C, Chunk F, Chunk G, and Chunk H) are still adjacent to Chunk B in the deduplicated layout. Because chunks in a sequential (one-dimensional) structure have only two adjacent positions, but Chunk B has five adjacent chunks (i.e., Chunk A, Chunk C, Chunk F, Chunk G, and Chunk H), the deduplicated data cannot be stored in a sequential structure but must be stored in a more complex structure to accommodate the additional adjacent positions. It is a three-dimensional structure in this basic example (Figure 1), but in more general cases, it will be a hyper-dimensional structure.

However, a hyper-dimensional structure is a mismatch to the sequential nature of most storage media, and it must be flattened to a linear structure. In past deduplication systems, the issue of the locality of deduplicated data was not considered in the framework of dimensionality, and instead there was a focus on maintaining sequential locality as much as possible. This strategy causes chunks from a backup that are logically consecutive to refer to previously written chunks scattered across the disks, which generates poor locality. As an example, consider backup *version* 1 that has few or no duplicates, so its chunks are stored sequentially. Then, *version* 2 may be highly redundant with the first backup with small modifications throughout the backup, so its recipe has references to many chunks of the first version intermixed with references to newly written chunks. Later, *version N* tends to have even worse locality as it refers to chunks written by many previous backup versions, so restoring a backup version involves random seeks back and forth across the disks, and read amplification is caused, since an accessed container may have both needed and unneeded chunks.

However, deduplication systems usually group the deduplicated chunks into a large unit called a **container** (often 1 MB in size or larger) for compression and maximizing write performance to arrays of disks, and containers are usually the basic unit of I/O (i.e., called Container-based I/O). Poor locality and Container-based I/O together lead to serious read amplification, which sacrifices restore performance. In an extreme case, a single 8 KB chunk is needed out of a multi-megabyte

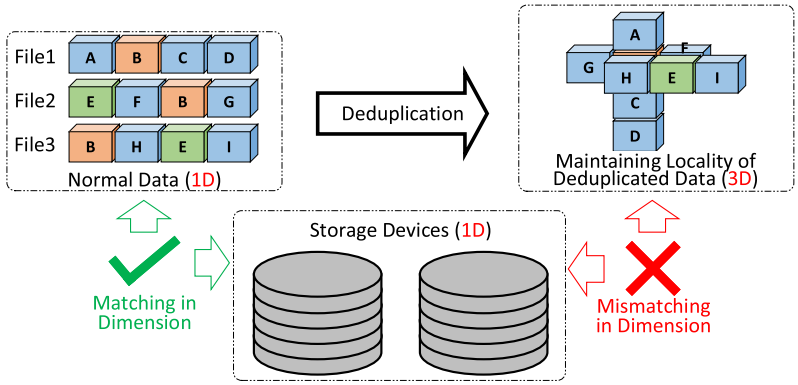


Fig. 1. Maintaining locality of deduplicated data in storage devices is challenging.

container, which may be read in its entirety. Deduplication systems then need a complicated **garbage collection (GC)** process to determine whether chunks remain referenced as files are written and deleted. GC will determine which chunks are live, which containers are efficient to clean, and how to copy live chunks forward into new containers.

To alleviate the fragmentation problem and improve restore performance, many techniques have been proposed that write some duplicates (called rewriting approaches) according to their "fragmentation degree" to maintain a level of data locality [20, 28, 30, 39, 40, 46]. Alternatively, there have been proposals to use memory or SSDs to cache the fragmented chunks or frequently referenced chunks [1, 32], which also helps achieve a higher restore speed, though with increased hardware costs. However, these approaches do not focus on the gap between the hyper-dimensional structure of deduplicated data and sequential format of storage devices, so fragmentation inevitably becomes worse with high generation backup versions.

According to our experimental observations on five backup datasets (see Figures 11 and 13 in Section 5), even using the state-of-the-art rewriting techniques of Capping [30] and HAR [21], the restore speed drops to about 1/8–1/3 of the sequential read speed of storage devices while the actual deduplication ratio drops about 20%–40% due to "rewriting." The performance of GC is also impacted by data locality in traditional data layouts. As older backup versions are deleted, some chunks become unreferenced by any version and can be removed from containers to reclaim space. Generally, GC includes two stages: selecting which containers have unreferenced chunks and migrating referenced chunks into new containers so selected containers can be freed. Several approaches [23, 48] have explored ways to quickly select containers for the first stage. When locality is poor, containers will have a mix of referenced and unreferenced chunks, so the migration stage is time-consuming, because many chunks must be read and rewritten.

Directly considering how to flatten the hyper-structure of deduplicated data is difficult, and in this article, we consider how to maintain locality in the flattened structure and discuss that from both the micro and macro perspectives.

From the micro view, we first notice the result of read amplification is related to chunks' lifecycles, and a lifecycle-based classification, which classifies chunks into categories according to their lifecycles, could eliminate read amplification. However, we observe that naive lifecycle-based classification is infeasible, because the number of categories is too large. In our observations of deduplicating backups, we find that almost all the duplicate chunks in a backup version B_{i+1} are derived from its previous version B_i (studied in Section 3.3), which means most chunks have a "successive" lifecycle across backup versions, and few chunks have a "skip" lifecycle that references previous

backup versions. This observation suggests that avoiding deduplicating skip duplicate chunks (i.e., splitting skip lifecycles into several successive ones) has little impact on the deduplication ratio, which provides an opportunity to reduce the number of categories from $2^n - 1$ to $n(n+1)/2$ for n backups and makes the lifecycle-based classification feasible.

From the macro view, we studied which categories will be required when restoring a backup version. This motivated our method to group categories and finally led to the proposed data layout, the flattened hyper-dimensional structure. We also notice a derived relationship between the new version and the old version of the proposed data layout, which inspired us to design a method for maintaining the correctness of our data layout.

Note that our proposed data layout is significantly different from a traditional deduplication layout that writes out chunks in the order written. Traditional deduplication mainly focuses on the write path of deduplication (which we call **write-friendly**) and rarely manages the location and placement of chunks. In contrast, our approach tries to redesign the data layout of deduplicated chunks to maintain locality as much as possible and thus achieves dramatically faster restore and GC performance, which we call **management-friendly**. The implementation of our layout uses an offline method of iteratively arranging chunks for each incoming backup version, and we find the costs are acceptable, especially compared with the huge overheads of restore and GC in previous write-friendly approaches.

Finally, we propose a novel Management-Friendly Deduplication framework, called MFDedup,built on two new techniques: **Neighbor-duplicate-focus (NDF)** indexing and **Across-version-aware Reorganization (AVAR)** scheme. Together, they generate and maintain our proposed data layout, which eliminates the fragmentation problem. Specifically, the contributions of this article are threefold:

- We designed NDF to only detect duplicates of a backup version (B_{i+1}) with its previous version (B_i), which utilizes our observation about deduplication patterns and supports our proposed data layout. NDF significantly reduces the memory footprint for the fingerprint index while achieving a near-exact deduplication ratio.
- After deduplicating the new version B_{i+1} using NDF, AVAR arranges the unique chunks of B_{i+1} into our proposed data layout by classifying and grouping chunks according to their lifecycles. As a benefit of our layout, GC becomes a simple operation of immediately deleting the oldest categories as the oldest versions are deleted.
- Evaluation results with five backup datasets suggest that, compared with previous approaches, MFDedup achieves a significantly higher deduplication ratio (1.12× to 2.19× higher) and restore throughput (1.92× to 10.02× higher). Restore throughput is increased and fully utilizes the bandwidth of the storage devices. Meanwhile, the NDF index has a fixed and limited overhead compared with the traditional global index, and the GC in MFDedup has nearly zero-overhead.

The rest of the article is organized as follows. Section 2 describes background and related work about deduplication and fragmentation. In Section 3, we introduce our motivation and theories of our approach. Section 4 presents the design and implementation of MFDedup in detail. Section 5 discusses evaluation results. Finally, we conclude our work in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 Background of Data Deduplication

Data deduplication is a widely used data reduction approach for storage systems [8, 17, 35, 42, 43, 49, 57, 58]. In general, a typical data deduplication system splits the input data stream (e.g., backup

files, database snapshots, virtual machine images, etc.) into multiple data "chunks" (e.g., 8 KB size) that are each uniquely identified with a cryptographically secure hash signature (e.g., SHA-1), also called a fingerprint [37, 43]. Deduplication systems then deduplicate data chunks according to their fingerprints and store only one physical copy to achieve the goal of saving storage space. The file is represented with a recipe structure used to read back chunks when a file is restored.

Backup storage and locality. *Backup storage* often leverages data deduplication due to the highly redundant nature of the data. In backup storage systems, workloads usually are a series of backups versions (i.e., successive snapshots of the primary data), and the size of backups can be greatly reduced to about 1/10–1/30 of their original size, reducing hardware costs. *Locality* in backup workloads means that the chunks of a backup stream will appear in approximately the same order in each full backup with a high probability, which is widely exploited for improving deduplication performance, such as for fingerprint indexing, restoring, and so on, by utilizing the high sequential I/O speed of HDDs (i.e., Hard Disk Drives).

Container-based I/O. Many deduplication-based storage systems usually include compression techniques, and all chunks in a container are compressed together as the basic unit for compression. Alternatively, consecutive chunks may be compressed together in a compression region, and multiple compression regions may be stored to a container. Thus, write I/Os are usually based on containers, and read I/Os are either for containers or compression regions. Usually, containers are immutable and have a fixed size (e.g., 4 MB). Containers offers several benefits: ① Writing in large units achieves the maximum sequential throughput of hard drives and is compatible with striping across multiple drives in a RAID configuration. Hard drives remain significantly cheaper than SSDs and other media, and cost is an important consideration for backup storage. ② The locality of data in containers is frequently leveraged to improve the efficiency of identifying duplicates as well as for restoring backups to clients [58].

Fragmentation Problem. Fragmentation in deduplication systems is related to container-based I/O and the seek latency of HDDs. Due to deduplication, different backups will share chunks, and these shared chunks are distributed across containers. In other words, *spatial locality* of each backup will be destroyed after deduplication. Due to container-based I/O, when we restore a backup, even if only a few shared chunks in a container are required, we have to read the whole container from HDDs, and this problem is a form of *read amplification* (defined as $\frac{Total\ Size\ of\ Loaded\ Containers}{Size\ of\ Actually\ Restored\ Data}$ during restores). Even if a system supports compression regions within a container, a full compression region must be read and decompressed to supply a needed chunk. In addition, since the required containers for each backup are randomly distributed across the HDDs, *seeking* to these required containers on HDDs is also time-consuming. Moreover, the read amplification and seek issues become worse as the number of backups increases.

2.2 Deduplication Techniques

A typical deduplication system usually consists of several techniques, including chunking, fingerprint indexing, restore optimizations, garbage collection, and so on.

Chunking Techniques. Content-defined Chunking (CDC) [19, 37, 41, 53, 55] is a widely used chunking approach to split the backup stream into variable-sized chunks according to the content, which can handle the "boundary-shift" problem existing in Fix-sized Chunking [43].

Fingerprint Index Techniques. Checking the fingerprint index (i.e., detecting duplicates) is a critical step in the workflow of deduplication. Fingerprint indices grow as a fraction of backup storage system capacity, so keeping them in memory is expensive and impractical while storing the index in HDDs will cause a deduplication system bottleneck for indexing. Several approaches [5, 7, 23, 31, 34, 36, 52, 58] have been proposed, and most leverage spatial or temporal locality by using

the fingerprint index to load many fingerprints from disk that were written at the same time or consecutively in a file.

Restore Optimization Techniques. As introduced in Section 2.1, fragmentation leads to read amplification and many disk seeks when restoring a backup. Among the restore optimization approaches, there are two main approaches to review that can be used separately or together: "*rewriting*" and "*caching*." Rewriting trades-off deduplication space savings to improve locality by selectively writing duplicates [10, 11, 20, 28, 30, 39, 40, 46]. Rewriting lowers the deduplication ratio, and results show read amplification remains 2×–4× after rewriting according to some previous works [20, 30]. The caching approach uses SSDs or memory to cache chunks that are frequently referenced or believed to be needed in the near future [1, 32], but the cache hit ratio still depends on locality, and read amplification is not addressed.

Among rewriting approaches, Capping [30] follows a simple policy. When deduplicating against a previously-written container, record how many chunks in the container are referenced for the current backup. For containers with low reuse, it writes duplicate chunks to improve the locality of the current backup version. HAR [21], utilizing the similarity of backup streams, identifies sparse containers according to historical information, and writes duplicate chunks instead of referencing sparse containers. In contrast, our approach writes minimal duplicate chunks and creates a data layout without any fragmentation or read amplification.

Garbage Collection Techniques. Customers usually configure a retention policy for backup files using their backup software, which often involves retaining weeks or months of backups and deleting backups older than the retention policy. GC then removes unreferenced chunks from the system [15, 20, 23, 48]. There are generally two kinds of GC in deduplicated systems. The first one is traditional Mark-Sweep [15, 23, 48]: it walks the backups and marks the chunks referenced from those backups, and then the unreferenced chunks are swept away. In practice, this often requires copying live chunks from a partially-unreferenced container and forming new containers. Although numerous optimizations have been proposed [15, 20, 23, 48], copying live chunks and writing new containers is I/O intensive [21].

The second approach is the **Container-marker Algorithm (CMA)** [21]. CMA maintains a container manifest, a mapping from container to backup that references it. As backups are deleted, whole containers can be deleted as they become unreferenced, which is a coarse-grained GC approach that maintains containers if any chunks are still referenced and thus causes wasted space. In contrast, we focus on fine-grained GC, but have an approach with dramatically lower overheads than previous techniques.

应用于主存储和辅助（备份/存档）存储

Inline and offline deduplication. Deduplication has been applied in both primary storage and secondary (backup/archival) storage, and there is previous work studying these two use cases. Primary storage [2, 12, 18] prioritizes low latency, whereas secondary storage works [13, 14, 14, 16, 23, 31, 33, 43, 44, 52, 54, 58] prioritize high throughput. Additionally, there is a hybrid solution, such as RevDedup [29], which deduplicates current backups inline and eliminates redundancy between older backups offline. RevDedup shifts fragmentation to older backups by adjusting their references to newer backups to retain the locality of newer backups that are more likely to be restored. As a result, restoring older backups will be slower with this approach.

3 MOTIVATION AND THEORIES

3.1 Behind the Fragmentation

Fragmentation is not a new topic in storage technologies. Disk defragmentation has been a common process for many years, which arranges and migrates blocks in file systems to reconstruct files' locality for better performance. After that, files are stored sequentially with the best locality.

Similarly, as discussed in Section 2.1, fragmentation also causes serious performance bottlenecks in deduplication systems. This leads us to focus on the question of how to design a defragmentation process for deduplicated storage?

As the example in Figure 1 shows, however we arrange the deduplicated chunks, there is always fragmentation for some files. Thus, it is easy to realize that a simple defragmentation process will not work in deduplication systems. This is because the causes for fragmentation in standard file systems and deduplication systems are different. For file systems, the locality in files matches the structure of storage devices. Specifically, fragmented files can be avoided by storing their content linearly, and storage devices also store data linearly. In other words, the fragmentation in standard file systems comes from misordering file data, and the structure of files and storage devices are compatible, though redirection related to snapshots can affect locality. But for deduplication systems, the fragmentation not only comes from misordering file data but is also produced by sharing chunks. Simple defragmentation could address misordering but cannot handle chunk sharing.

In fact, sharing chunks deeply changes the locality of deduplicated data. As Figure 1 shows, if we want to keep the locality of three deduplicated files, then they are curled to a three-dimensional structure. And for more complex examples, the dimension of the structure will be larger. Thus, we acquire **our first** observation that for deduplicated files, its locality needs to be achieved in a hyper-dimensional structure.

To store deduplicated data in linear (1D) storage devices, the hyper-dimensional structure must be flattened, and data locality is destined to be broken. Therefore, these problems motivate the question **how can we build a mapping to flatten the hyper-dimensional structure of deduplicated data to a linear format and maintain as much locality as possible.**

3.2 Puzzle Pieces: A Micro Perspective of Flattened Structures

Generally, it is challenging to consider how to flat the hyper-dimensional structure as a whole, because it is difficult to imagine what a hyper-dimensional structure looks like and how it could be converted to a HDD-friendly format. Thus, in this subsection, we discuss the flattened structure from a micro perspective, think about local properties, and try to generate "Puzzle Pieces" that are locally flat but together represent the high-dimensional structure.

For typical deduplication systems, deduplicated chunks are written and organized in the order they are seen. Figure 2 shows an example of the traditional data layout after deduplication of three backup versions. Deduplicated chunks from three backup versions are stored in five containers using the traditional data layout. It is easy to notice that the lifecycles of chunks in *Container* 1 are different, since they are referenced by multiple backups, and the oldest backups will likely be deleted before newer backups. *Chunk* 1 and *Chunk* 6 are shared by all *Backup*s, but *Chunk* 2 is only referenced by *Backup* 1. When we want to restore *Backup* 2 and *Backup* 3, *Chunk* 1 and *Chunk* 6 are required, but *Chunk* 2 is also in *Container* 1. Thus, *Chunk* 2 breaks the locality of *Backup* 2 and *Backup* 3, and when accessing *Container* 1 to restore these two backups, *Chunk* 2 is also read from disk though it is unneeded, which causes **Read Amplification**. This example gives us a hint that chunks with identical lifecycles should be arranged sequentially.

To implement this idea, we classify chunks into categories with their lifecycles, like "a data layout inspired by lifecycle classification" in Figure 2. **Categories** could be considered as variable-sized containers in this data layout. Chunks with identical lifecycles are arranged sequentially in categories, and whatever category to accessed, the locality is always ensured. Thus, **these categories could be "Puzzle Pieces" for our flattened structure.**

However, we should consider the number of categories created after classification. Each category maps to a kind of lifecycle, which is referenced by several backup versions. Since a set with n

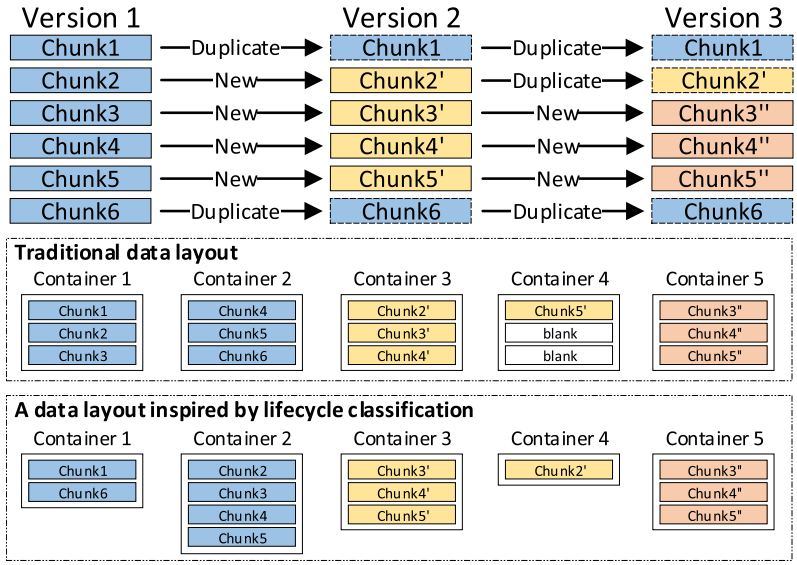


Fig. 2. Examples of two data layouts for deduplicated data.

elements has $2^n - 1$ non-empty subsets, there could be $2^n - 1$ kinds of lifecycles if there are n backup versions. $2^n - 1$ categories is an impractically large number for typical backup retention policies. For example, when $n = 30$, there will be over 1 billion categories, which is unacceptable for deployed systems.

3.3 Reducing the Number of the Puzzle Pieces

In this subsection, we will present our key observation about the relationship of backups through an analysis of five large backup datasets. These relationships can be exploited to greatly decrease the number of categories (i.e., containers) needed for the flattened structure.

In backup storage systems, workloads usually consist of a series of backup images, which are all generated from the original data on a primary storage system (i.e., laptop, server, database, etc.). Therefore, the duplicate chunks of each backup are not randomly distributed but are derived from the chunks of the last backup, as we will show with experiments, which is consistent with the typical consecutive pattern of duplicates that is leveraged by many systems [31, 52, 58] for high deduplication performance.

To better illustrate our observation, we denote four kinds of chunks in a backup version B_i as follows:

- *Internal duplicate chunks*, whose referenced chunks are also in B_i.
- *Adjacent duplicate chunks*, whose referenced chunks are not in B_i but in the last version B_{i-1}.
- *Skip duplicate chunks*, whose referenced chunks are neither in B_i nor in B_{i-1}.
- *Unique chunks*: the non-duplicate chunks.

Figure 3 studies the distribution of the four kinds of chunks in five backup datasets, which includes real-world and synthetic datasets that are described in Section 5.1. Exact deduplication was used in this analysis.

We find that Internal, Adjacent, and Unique chunks all have consecutive lifecycles, while Skip chunks have skip lifecycles. From Figure 3, we can acquire **our second** observation that most

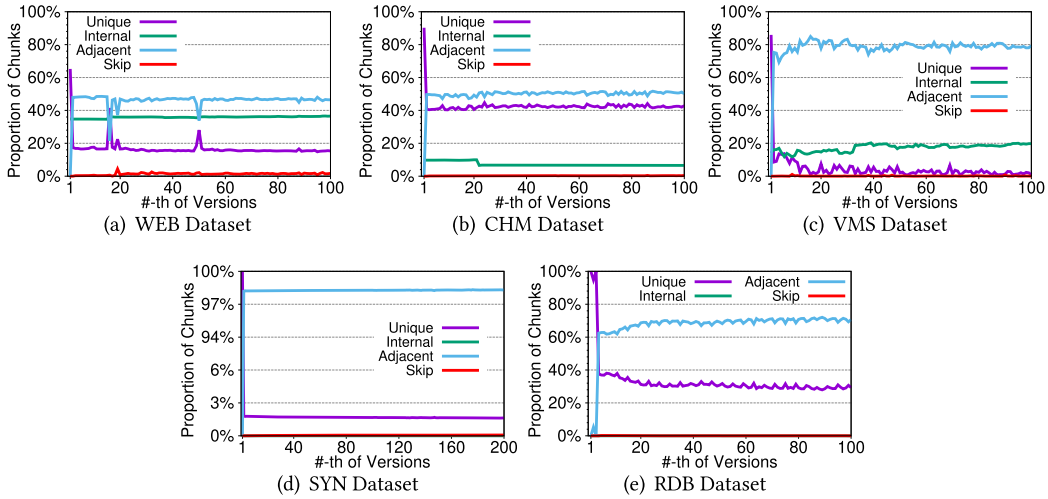


Fig. 3. Distribution of four kinds of chunks in five backup datasets. *Skip* duplicate chunks are the least common.

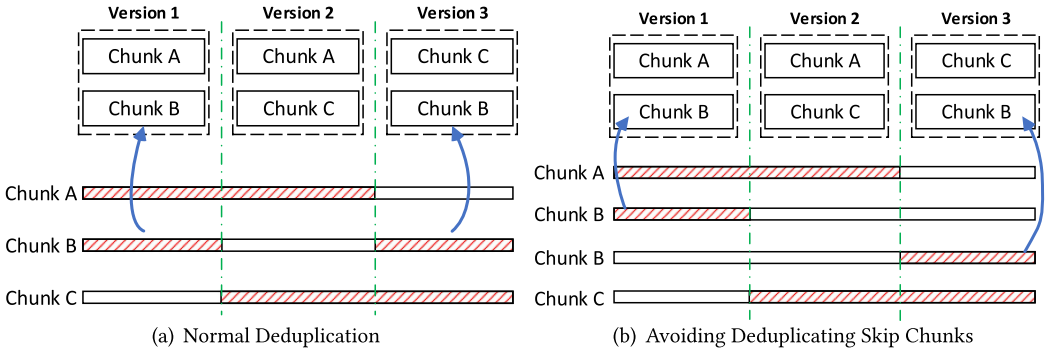
duplicate chunks for a backup version are from the previous version (*Adjacent duplicate chunks*) and within the current backup itself (*Internal duplicate chunks*). Specifically, Adjacent and Internal duplicate chunks account for more than 99.5% of all duplicate chunks in most datasets, and *Skip* duplicate chunks only consist of a small fraction (i.e., less than 0.5% in most datasets). This observation suggests an approach of **avoiding deduplicating Skip chunks**, which means splitting a skip lifecycle into several successive lifecycles and treating them as Unique chunks, would cause an insignificant reduction in the deduplication ratio.

However, avoiding deduplicating Skip chunks gives an opportunity to enforce a consecutive lifecycle for each physical chunk. For example, Figure 4 gives an example to demonstrate physical chunks' lifecycle in different deduplication strategies. In Figure 4(a), *Chunk B* is referenced by *Version* 1 and *Version* 3, thus, it is a Skip duplicate chunk and its lifecycle (across *Version* 1 and *Version* 3) is not consecutive. If we avoid deduplicating Skip chunks, as Figure 4(b) shows, then there will be two physical Chunks B, and one of them is referenced by *Version* 1 while another is referenced by *Version* 3.

In this way, all physical chunks have consecutive lifecycles, and the kinds of lifecycles (or classified categories in Section 3.2) could be hugely reduced. Taking three backup versions for example, because each chunk must be referenced by consecutive versions, their lifecycles are always $\{B_i, \ldots, B_{i+k}\}$, where $i \geq 1$ and $i + k \leq 3$. Thus, when i = 1, k={0, 1, 2}; i = 2, k={0, 1}; i = 3, k={0}, which means there are at most six categories (kinds of lifecycles). In other words, when k=0, there are $\binom{3}{1}$ categories; when $k \neq 0$, there are $\binom{3}{2}$ categories (choosing the start and the end lifecycles for successive versions). Hence, in this example, the number of categories is reduced from seven to six.

Generally, if we have n backup versions and apply the above strategy, then the upper limit of the number of categories (containers) will be $\binom{n}{1} + \binom{n}{2} = n(n+1)/2$, which is much less than $2^n - 1$. Continuing an earlier example with 30 backup versions, there will be 465 containers with about 2150 chunks on average, which is acceptable for most deduplication systems.

In summary, we learn that avoiding deduplicating Skip chunks makes the classification feasible for lifecycle-based data layout with an insignificant reduction of deduplication ratio.



(a) Normal Deduplication (b) Avoiding Deduplicating Skip Chunks

Fig. 4. Example of physical chunks' lifecycle in different deduplication strategies.

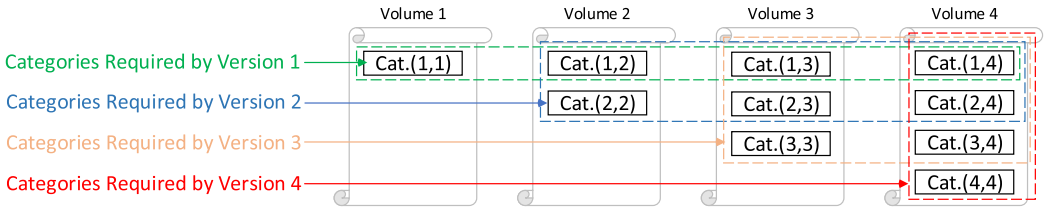


Fig. 5. Construction of the flattened structure with "Puzzle Pieces" (Categories).

3.4 Blueprint: Construction of the Flattened Structure from a Macro View

After steps discussed in Sections 3.2 and 3.3, "Puzzle Pieces" (or categories) are prepared and acquired. In this step, we consider how "Puzzle Pieces" will be used in restoring workloads, which inspires us to construct the whole flattened structure with "Puzzle Pieces."

First, we present how to name categories. Each category includes one or more chunks, and these chunks all have the same lifecycle, which is different from those of chunks in other categories. Thus, we name categories with its chunks' lifecycle. For example, if a category includes all chunks with a lifecycle from $Version\ n$ to $Version\ m$, then we denote this category as Cat.(n,m).

In a restore process, we can learn which categories are necessary to access from their names. For example, if there are n backup versions stored, and we want to restore a backup version B_k, then all categories whose lifecycles include B_k are required. Specifically, Cat.(3,k+2) is required, because its lifecycle is from B_3 to B_{k+2}, which includes B_k. Thus, all required categories for B_k could be represented as

$$\begin{aligned} Required\ Cat. &= \{Cat.(i,j)\}, where\ 1 \le i \le k \le j \le n \\ &= \cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j). \end{aligned} \tag{1}$$

According to Equation (1), we learn that, when restoring a certain backup, Cat.(1,n), Cat.(2,n), . . ., Cat.(k,n) are always required together, which gives us a hint that we should place them sequentially in the flattened storage structure.

Therefore, we construct the flattened structure like Figure 5 shows, in which Cat.(1,n)–Cat.(n,n) are stored sequentially in a volume. In this flattened structure, for whichever backup version that is restored, the required categories are always in always in consecutive order, which enables storage devices to achieve their maximum sequential throughput.

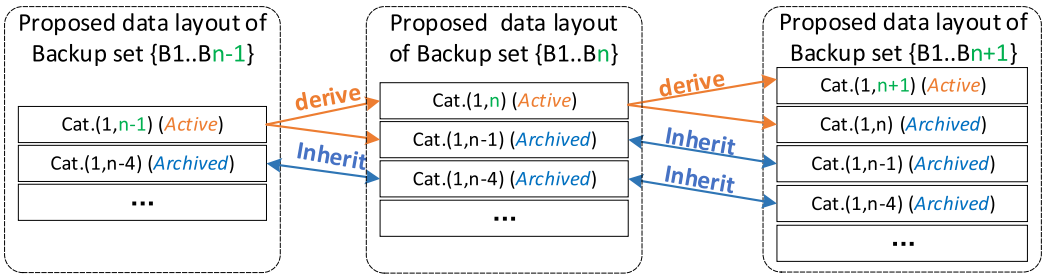


Fig. 6. An example of the flattened structure's evolution on three backup versions. Some categories are inherited from the previous version, and some derive new categories.

3.5 Evolution of the Flattened Structure

In the previous sections, we design a flattened structure (or data layout) that **promises no read amplification and limited random seek operations in restoring any backup version**. However, there still exists a problem: how to achieve this data layout. Because reorganizing all data to generate a specific layout is very time consuming, we study the evolution of the proposed data layout and try to design a method to iteratively update the data layout.

Figure 6 shows a general example of the proposed data layout with three backup versions and the evolution of the categories. As we mentioned in Section 3.3, chunks in our proposed data layout always have a consecutive lifecycle. Thus, for a backup set $\{B_1..B_{n-1}\}$, Cat.(1,n-4) will not be referenced by B_n or later backup versions, because its lifecycle is from B_1 to B_{n-4} and does not include B_{n-1} (the last one in $\{B_1..B_{n-1}\}$). As a result, these kinds of categories are always carried forward as a fixed part of our proposed data layout, and we call them **archived**.

However, Cat.(1,n-1) is referenced by the last backup version B_{n-1} in the set, thus, chunks in Cat.(1,n-1) could be referenced by the next version B_n. Therefore, Cat.(1,n-1) will be split into two categories when backing up B_n: a subset of chunks in Cat.(1,n-1) are referenced by B_n, and then, they should be migrated to Cat.(1,n), since their lifecycles are extended; another subset of chunks in Cat.(1,n-1) are not referenced by B_n and they should remain in Cat.(1,n-1). Since categories like Cat.(1,n-1) will be split, we call them **active**.

Therefore, in our proposed data layout, classified categories have two states in the evolution of our data layout: **Active** and **Archived**. Archived means the categories are immutable, while Active means the categories will be further split after future backups. Thus, when n backup version are stored, there will be $n(n+1)/2$ categories and most categories (i.e., $n(n-1)/2$) are in the Archived state. Thus, in the evolution of our data layout, we only need to consider updates on Active categories like the example in Figure 6, and Archived categories will not be modified.

We have designed the flattened structure in Sections 3.2, 3.3, and 3.4 and explored how to generate the flattened structure with acceptable costs in Section 3.5. We have covered the theoretical motivations for our work and will next present the design and implementation details.

4 DESIGN AND IMPLEMENTATION

4.1 MFDedup Overview

Based on discussions in Section 3, implementing our proposed data layout in a deduplicated backup storage systems is feasible by leveraging the following two key design principles: ① *All chunks are classified into categories (like containers) according to their lifecycles.* ② *Skip duplicate chunks are treated as unique chunks to reduce the number of lifecycles.*

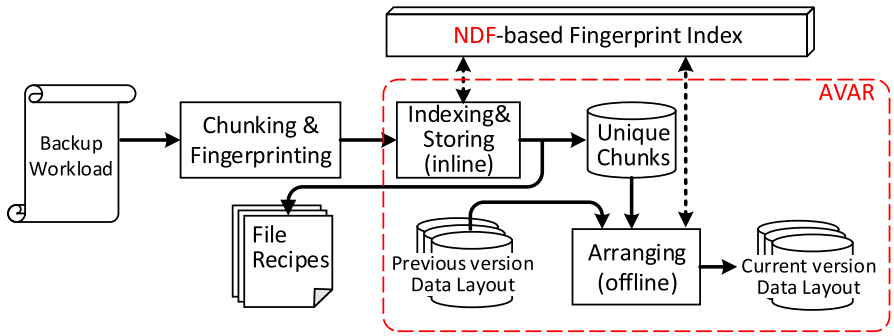


Fig. 7. An overview of MFDedup framework.

We follow the two design principles outlined above and propose MFDedup, a management-friendly deduplication framework. MFdedup will identify three kinds of chunks: Internal duplicate chunks, Adjacent duplicate chunks, and Unique chunks. Internal duplicate chunks match identical chunks within the same backup version; Adjacent duplicate chunks cannot find their identical matches within the same backup but do find a match in the previous backup; the remaining chunks are Unique Chunks. Briefly, MFDedup inline deduplicates backups against themselves and their previous backups to eliminate Internal and Adjacent duplicate chunks, and iteratively reorganizes chunks in an offline algorithm to achieve the data layout, which is achieved by implementing two techniques:

也就是说，新版本和上一版本之间的index是在inline完成的，通过index更新data layout是在offline完成的

- **Neighbor-Duplicate-Focus indexing (NDF).** MFDedup eliminates duplicates inside single backups and between consecutive backup versions. Hence, we only need to build and access two local fingerprint indexes, which consist of the latest backup's fingerprints and its previous backup's fingerprints. Therefore, the memory cost of the fingerprint index could be lower compared with a traditional global fingerprint index, as detailed in Section 4.2.
- **Across-Version-Aware Reorganization (AVAR).** After using NDF to detect and eliminate duplicate chunks inside the new backup version and between consecutive backups, MFDedup arranges the remaining chunks of the last backup version in an offline process. Specifically, these chunks are classified and grouped to iteratively update the flattened structure according to their lifecycles, as detailed in Section 4.3.

The overall workflow of the MFDedup framework is shown in Figure 7, which includes three key stages: *Chunking & Fingerprinting*, *Indexing & Storing*, *and Arranging*. Chunking & Fingerprinting refers to splitting the backup stream into chunks using Content-Defined Chunking [37, 51] and then calculating a fingerprint (i.e., SHA1 digest) for each chunk. Indexing & Storing detects duplicate and unique chunks from the previous backup version by using a NDF-based fingerprint index and then stores unique chunks and a **Recipe** for each backup. Arranging is an offline process, which iteratively updates the proposed data layout version by version, with the support of a NDF-based fingerprint index. Note that the Recipe records the chunk-fingerprint sequence of a backup version, which is used to recover the backup version after deduplication.

In general, MFDedup applies inline deduplication using NDF, which removes duplicates within single backups and between neighboring backup versions. MFDedup then starts an offline Arranging process using AVAR, which keeps the proposed data layout to maintain locality of backup workloads and thus eliminate fragmentation. However, MFDedup is self-organized and has significant differences with previous deduplication architectures. The exact physical position and the

无需精确的物理地址，也无需对unique chunk保留引用
reference counts of each unique chunk, which are usually used for restore and GC in traditional deduplication systems, are not needed for MFDedup. For example, in restore, the required categories for each version are calculable in our data layout. Thus, the cost of metadata is also greatly reduced.

4.2 Neighbor-duplicate-focus Indexing

In this section, we will introduce the Neighbor-duplicate-focus indexing (NDF) technique in MFDedup, which is based on our observation (Section 3.3) that most duplicate chunks exist between neighboring versions in a backup storage system (i.e., duplicate chunks of backup version B_i are nearly all from its previous version B_{i-1}). Therefore, MFDedup chooses to treat *Skip* duplicate chunks as unique chunks instead of deduplicating them. In other words, MFDedup only identifies duplicate chunks in backup version B_i that are identical to chunks either in B_i (**within the same version**) or B_{i-1} (**the previous version**). We refer to this as a **NDF-based fingerprint index**.

In the NDF implementation, we maintain an independent fingerprint index table for each backup version. Besides using NDF in the Indexing & Storing stage of MFDedup for duplicate detection, NDF is also used in the *Arranging* stage for classification as detailed in next subsection. After a fingerprint index table is used in the above two stages for the latest two backup versions, it can be released. Therefore, we only need to maintain two fingerprint indices, which could be kept in memory if they are small (they are typically much smaller than the traditional index that stores all versions) or loaded using previous locality-based approaches [31, 58]. Assuming a fingerprint index entry takes 20 bytes (i.e., the size of SHA1 digest), the size of a backup version is 10 GB, and the expected chunk size is 8 KB, the total memory cost of the NDF-based fingerprint index will be $2 \times 10\text{GB}/8\text{KB} \times 20\text{B} = 50\text{MB}$ (about 0.4882% of the size of a backup version).

Indexing overhead of NDF is related to the data size of the two most recent backup versions, which is considerably smaller than traditional deduplication systems that have a global fingerprint index. Meanwhile, NDF is able to achieve a near-exact deduplication ratio while supporting the proposed data layout in MFDedup (detailed in Section 4.3). MFDedup使用了NDF，并且将index table保留在了RAM。以前的技术因为index过多而用了cache。

这段没太看懂 Although the current design of MFDedup has the index in RAM, previous techniques for prefetching and caching sequences of fingerprints (designed for a large fingerprint index) [5, 7, 23, 31, 34, 36, 52, 58] could also be used in MFDedup. We expect the sequential locality to also exist in MFDedup for two reasons. First, sequential locality exists inside categories, although it is destroyed across categories by Arranging. Second, recipes also keep the sequential locality of each backup.

4.3 Across-version-aware Reorganization

In this subsection, we will introduce AVAR in MFDedup, which is designed to eliminate fragmentation and generate the proposed data layout in combination with the NDF technique.

There are two stages in AVAR, which are the *Deduplicating* stage (i.e., *Indexing & Storing* in Section 4.1) and the *Arranging* stage, and both utilize the NDF-based fingerprint index. Figure 8 gives an example of AVAR on three backup versions, running with the two stages alternating (except the first backup version): The *Deduplicating* stage identifies unique chunks of a backup version B_i. Then, according to the evolution rules we discussed in Section 3.5, the *Arranging* stage updates the proposed data layout for backup version sets $\{B_1 \ldots B_i\}$, by reorganizing the previous data layout of $\{B_1 \ldots B_{i-1}\}$ with the unique chunks of B_i. Note that there is naturally a flattened structure when the first backup version is stored, so the *Arranging* stage is not required after the Deduplicating stage. More details about the two stages of AVAR are elaborated below.

***Deduplicating* Stage.** In this stage, we detect duplicate chunks using the NDF-based fingerprint index and then store unique chunks as well as Recipes. In the remainder of this section, we ignore

1. 发现重复快
2. 存储unique chunk到recipe中

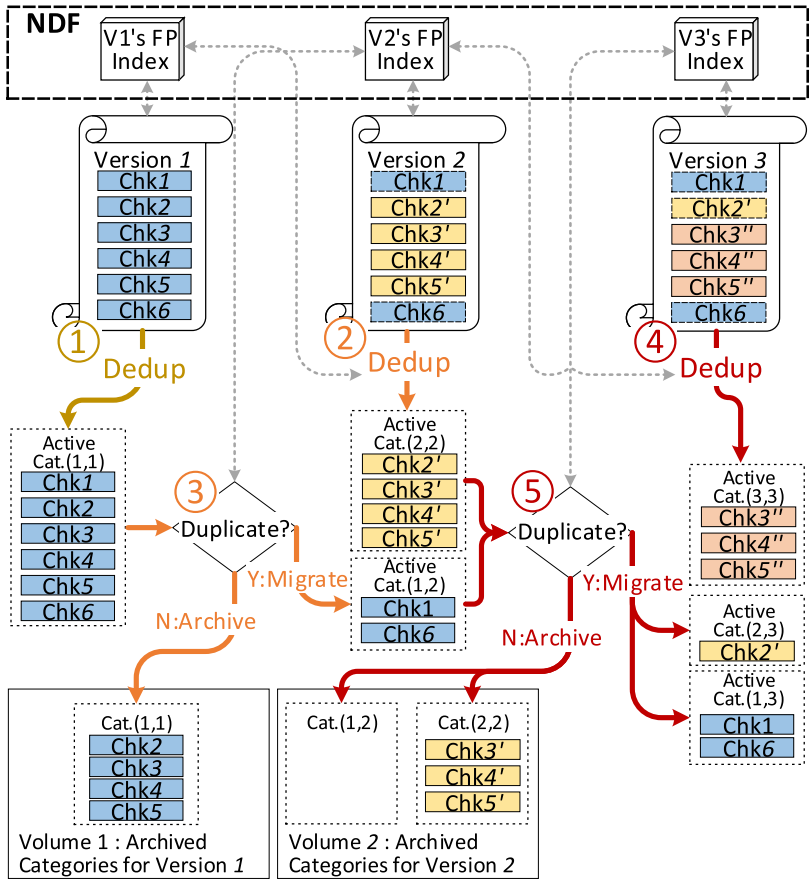


Fig. 8. An example of the AVAR workflow on three backup versions, which is presented by a solid line in five steps: ① *Deduplicating* Version 1 → ② *Deduplicating* Version 2 → ③ *Arranging* Version 1 → ④ *Deduplicating* Version 3 → ⑤ *Arranging* Version 2. Gray dashed lines refer to fingerprint indexing operations.

Recipes and focus on how data chunks are managed for the flattened structure. Therefore, as shown in Steps ①, ②, and ④ of Figure 8, the Deduplicating Stage is responsible for storing unique chunks of the latest new backup version B_i into a new active ***Category***, which is currently only referenced by B_i. Note that we have mentioned "active" and "archived" category in Section 3.5, and chunks in the active Category may be referenced by future backup versions and thus will be processed by the Arranging stage later.

***Arranging* Stage.** According to the first principle of MFDedup, classification methods used for generating the proposed data layout are based on the chunks' lifecycles, which means the old proposed data layout expires when a new version arrives and is processed by the *Deduplicating* stage. This is because some chunks' lifecycles have changed. Therefore, the Arranging stage is responsible for iteratively updating the existing data layout with new unique chunks of the incoming version (after the Deduplicating stage).

Note that Cat.(x,y) includes all chunks whose lifecycles are from Backup Version x to Backup Version y as introduced in Section 3.4. Like the example of *Step* ⑤ shown in Figure 8, Cat.(1,2) and Cat.(2,2) are the only two existing (old) active categories after backing up Version 3, and we check each chunk of them with the fingerprint index of backup version 3. Duplicate chunks, existing

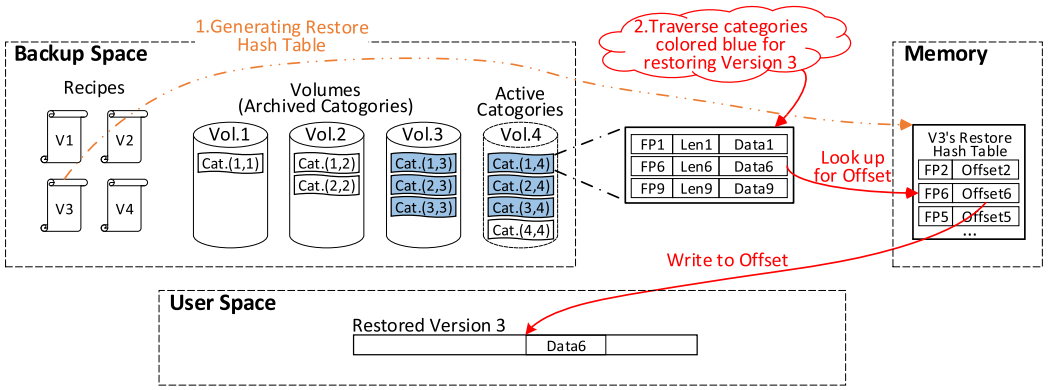


Fig. 9. An example of restore on the flattened structure with four backup versions.

in Version 3, are migrated to new active Cat.(1,3) and Cat.(2,3), and other chunks are arranged in archived Cat.(1,2) and Cat.(2,2). After migrating and archiving, old active Cat.(1,2) and Cat.(2,2) are no longer required and thus deleted.

Grouping. After Arranging existing Active categories, the new Archived categories are grouped into a **Volume** by the order of their name (e.g., in the order of Cat.(1,3), Cat.(2,3), Cat.(3,3)), as discussed in Section 3.4.

4.4 Restore

Restore benefits from our proposed data layout in MFDedup and the workflow is greatly simplified as described in this subsection.

When restoring a backup version in MFDedup, we only need to read the required categories on the proposed data layout, which is referenced by the to-be-restored version. Meanwhile, recording physical locations of chunks in a recipe is not required in MFDedup, because the needed categories can be simply calculated.

When restoring, required categories are given in Equation (1): when there are n backup versions and we want to restore B_k, required categories are $\cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j)$. As shown in Figure 5, whatever backup version is restored, the needed chunks are always sequentially located in several volumes, and reading these necessary chunks in order will never cause any read amplification. However, the number of referenced volumes is limited to never be bigger than the number of retained backups, and the number of random seeks is also small and predictable. These features mean that restoring older and newer backups will always be free from read amplification and achieve a high performance.

In the example of Figure 9, there are four stored backups, and restoring Version 3 requires the blue-colored categories. Note that according to our grouping approach, $\cup_{i=1}^{j} Cat.(i,j)$ are always sequentially grouped in the same Volume. Thus, loading $\cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j)$ requires n sequential reads at most.

A recipe is required to restore a backup version, which is used to build a **restore hash table**, whose format is shown in Figure 9. The entry of the hash table is a pair <fingerprint, offset>, which records a chunk's fingerprint and its offset in the to-be-restored file.

The workflow of restore is shown in Figure 9, after getting the required categories, the chunks are restored one by one according to the Recipe for $Version$ 3. Therefore, MFDedup only needs to seek to the required volumes and then sequentially read the required (consecutive) categories

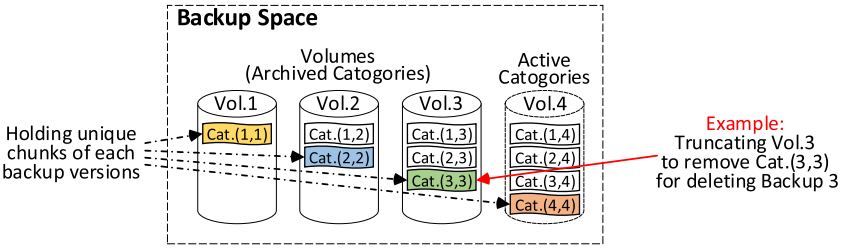


Fig. 10. An example of deleting Backup 3 on the flattened structure with four backup versions.

in those volumes, which achieves a high restore performance (i.e., few seeks and large sequential I/Os). The writes to the newly restored file may be out of order, which we believe is acceptable with the use of SSDs in primary storage. This is discussed further in Section 4.7.

这个不大懂，是指针对新版本备份写？还是写入新备份？

4.5 Deletion and Garbage Collection

As a result of our proposed data layout, deletion and garbage collection are simple, and space can be immediately reclaimed in MFDedup. In deduplication systems, deleting a backup version means reclaiming its unique chunks (those not referenced by other backups). FIFO-based deletion in MFDedup simply deletes and reclaims the earliest volumes, because they consist of unique chunks of the earliest backup versions. A volume consists of categories with chunks referenced by backups up to that volume number, so when all backups up to the volume number have been deleted, it is safe to remove the volume, since it is unreferenced. For example, in Figure 10, we can reclaim space of *Version* 1 by directly deleting *Volume* 1, since *Volume* 1 only includes Cat.(1,1), which hold all unique chunks of *Version* 1.

说实话我还是不大明白，连续存储到底指的是category连续还是category指向的数据块也连续。。。

MFDedup also supports deleting other backup versions besides the earliest ones. From the description of the Deduplicating stage, we see that the unique chunks of each backup version are always stored in the last category of each volume (see Figure 10). Thus, we can also delete any backup version by resizing the corresponding volume using "*truncate()*" (i.e., deleting the last category in this file). For example, if we want to delete *Version* 3 in Figure 9, then we can remove Cat.(3,3) by just truncating *Volume* 3. A previous work [15] mentioned that the CMA approach [21] only supports FIFO deletion, while we support any deletion pattern.

666

In this way, we no longer apply traditional GC techniques in MFDedup, such as mark-and-sweep or reference-count management, since the chunk-reference relationship is naturally designed into our classification-based data layout. This is a dramatic reduction in system resources (CPU cycles, RAM, I/O) and coding complexity, since GC is one of the most resource-intensive modules of deduplicated storage, though some of that work has been shifted into the Arranging phase as discussed in Section 4.7.

它这意思是说，每次删除一个备份只删除那个备份特有块，也即Cat n,n ，也就是说比如备份2被删了，那么Cat 1,3 依然整体存在而不会被拆分为Cat 1,1 + Cat 3,3

4.6 Space Management

Deduplication allows backup versions to share common data chunks to reduce space requirements but brings another challenge: how to estimate the space usage of each version?

In the existing approaches, chunks of different backup versions are mixed in containers, and it is hard to acquire the total size of a specific version's unique chunks or shared chunks. Thus, when a deduplication system becomes full, it will be difficult for administrators to decide which files to delete to release storage space or how much space will be released. However, administrators also struggle to estimate how much more can be written to a deduplication system, since it is difficult to estimate the incremental space needed for the next backup. It is difficult to answer

these two issues [45] for previous deduplication systems, though sketching approaches have been considered [24].

MFDedup offers a new, simple solution to these existing problems. First, MFDedup could tell administrators how much space will be freed after deleting backup versions. Because categories in volumes are already archived, and their sizes are fixed, MFDedup stores categories' information in the header of volumes. More specifically, the header of each volume will record how many categories are stored in this volume and the total size of each category. With these records and the naming style we defined in Section 3.4, the amount of space uniquely associated with each backup file can be quickly calculated. As the example in Figure 10 shows, if we want to know how much space will be freed after deleting *Version* 2 and *Version* 3, then we first find out all referenced categories: Cat.(2,2), Cat.(2,3), and Cat.(3,3) in this example. Then, from the header of *Volume* 2 and *Volume* 3, we can acquire the sizes of referenced categories, and the to-be-freed space could be totaled.

Second, MFDedup could help administrators estimate how much more can be written to the deduplication system. According to the arranging strategy, chunks that are not referenced by the next backup version will be archived to volumes, which means that each volume represents the difference between neighboring versions. For example, *Volume* 2 includes all chunks that belong to *Version* 2 but are not referenced by *Version* 3. Therefore, we could observe the trend of volumes' sizes to predict the next backup version's unique content size, assuming the change rate remains consistent.

4.7 Discussion and Limitations

[Scenario]. As introduced in Section 3.3, to make our data layout feasible, we reduce the number of categories by only allowing consecutive lifecycles. This means that MFDedup cannot handle the case that the backups fork into several branches. As a result, MFdedup is best suited for backups of virtual machines and disks, which rarely have numerous branches.

[Arranging versus Garbage Collection]. As introduced in Section 10, MFDedup supports simple, immediate deletion instead of complex, background deletion with garbage collection (used in other approaches), but MFDedup adds an Arranging process. Though both Arranging and Garbage Collection are offline processes, their goals are distinct: Garbage Collection seeks to remove unnecessary chunks from disks, whereas Arranging focuses on ensuring chunks' placement in categories according to their lifecycles, avoiding a fragmented layout. A benefit of Arranging is that GC is replaced with simple deletion. Arranging and GC are designed for different purposes, and we experimentally evaluate the overheads of each approach.

[Memory Overhead in the Restore Workflow]. As discussed in Section 4.4, the restore workflow in MFDedup requires a restore hash table, which records fingerprints and offsets for the to-be-restored backup. The size of this table is related to the size of a single backup. For example, for a 1TB size backup, the size of its restore hash table would be 3.5GB. Some previous works [4, 49] suggest that the majority of backups were 50–500 GB in Data Domain and Symantec production systems, and thus, MFDedup can be directly applied in these scenarios with a reasonable memory overhead. Moreover, for much larger backups, the restore hash table could be maintained in an on-disk (possibly SSD) key/value store to reduce memory cost.

[Out-of-order Restore]. Unlike the implementation of the traditional deduplication framework, restore in MFDedup is out-of-order, which means the writing order of restored chunks does not absolutely follow their logical order in the backup file. While the chunks in volumes are generally in order for a backup, there are logical gaps that are filled by other volumes, which causes random writes to the restored version. Although the sequential locality still exists inside categories,

as discussed in "Fingerprint Prefetching," restore will have better performance if the destination media has good random write performance (e.g., SSDs). Also, previous techniques, like a reassembly buffer [26], could be applied to improve the performance when streaming a restore to HDD devices.

增量备份

这段没看懂

[Incremental Backups]. The description of MFDedup focuses on full backups (i.e., a full snapshot of primary storage), but it also works for incremental backups. Incremental backups may record several modified blocks and their offsets in primary storage (the size of tracked blocks may be much larger than CDC chunks), such as (Block A, Offset A), which is the "difference" compared with the previous backup.

For this case, we can construct a full recipe by copying the previous version's recipe and applying the changes. Specifically, we first build a recipe for the incremental backup Bi_k, in which we copy entries from the previous backup B_{k-1}'s recipe except for chunks in the range of Block A. After that, we reconstruct Block A-1, whose offset is Offset A-1 (starting at the previous chunk anchor point before Block A and extending to the next chunk anchor point beyond Block A). We then recalculate chunk anchor points in Block A-1. Finally, we find duplicate/non-duplicate chunks in Block A-1, store non-duplicate chunks and update Bi_k's recipe. Chunk lifecycles and Arranging are handled normally.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

Evaluation Platform and Configurations. We perform our experiments on a workstation running Ubuntu 18.04 with an Intel Core i7-8700 @ 3.2 GHz CPU, 64 GB memory, Intel D3-S4610 SSDs, and 7,200 RPM HDDs.

In our evaluation, we built a MFDedup prototype system and also built Destor [22] for comparison with several state-of-the-art techniques for restore and GC, including the **History-aware Rewriting (HAR)** algorithm [21], Capping [30], SMR [50], and CMA [20]. MFDedup and Destor use the same configuration in the Chunking & Fingerprinting stage: Chunking uses FastCDC [53] with the minimum, average, and maximum chunk sizes set to 2, 8, and 64 KB; Fingerprinting uses a SHA1 digest generated by the Intel Intelligent Storage Acceleration Library Crypto Version (i.e., ISA-L_crypto [25]).

Experimental methods. To simulate real backup/restore scenarios, we separate the storage space of our workstation into two parts: a backup space using a 7,200 RPM HDD and a user space using an Intel D3-S4610 SSD. Both spaces (drives) have XFS file systems. It is typical in backup environments to use HDDs for cost reasons while primary systems often use SSDs for higher performance.

To evaluate backup/restore performance, datasets are backed up from the user space (SSD) to the backup space (HDD) version by version while the restore runs in the reverse direction. Note that before each backup/restore, we always flush the file system cache using the command: "echo 3 > /proc/sys/vm/drop_caches."

To simulate users' retention (deletion) requirements in backup systems, we retain the most recent 20 versions. Thus Version $n - 20$ is deleted after Version n is backed up, which is the same as the previous work HAR [21] and CMA [20]. For throughput (time cost) of backup, restore, and GC/Arranging in our evaluation, we present the average result of five runs.

Container-based I/O is considered, because many deduplication-based storage systems usually combine deduplication with compression techniques, and all chunks are stored in containers as the basic unit for compression. Because we are focusing on deduplication, and compression techniques are orthogonal to deduplication, we do not include compression in our evaluations.

Table 1. Five Backup Datasets Used in Evaluation

Name	Total Size Before Dedup	Versions	Workload Descriptions
WEB	269 GB	100	Backup snapshots of website: news.sina.com, captured from June to September in 2016.
CHM	279 GB	100	Source code of Chromium project from v82.0.4066 to v85.0.4165
VMS	1.55 TB	100	Backups of an Ubuntu 12.04 Virtual Machine
SYN	1.38 TB	200	Synthetic backups by simulating file create/delete/modify operations [47]
RDB	1.5 TB	100	Generated backups of RocksDB [9]

Evaluation Datasets. Five backup datasets are used for evaluation as shown in Table 1. These datasets represent various typical backup workloads, including website snapshots, an open source software project, virtual machine images, RocksDB snapshots and a synthetic dataset. Deduplication ratios vary from 2.19 to 44.65. The WEB, SYN, and VMS datasets have been studied previously in many deduplication papers [21, 53, 56].

5.2 Actual Deduplication Ratio

As mentioned in Section 2, rewriting and certain GC techniques (e.g., HAR, Capping, SMR, and CMA) consume more storage space in exchange for better restore and GC performance. Similarly, MFDedup ignores Skip Duplicate Chunks to implement the proposed data layout, which also reduces the deduplication ratio. Hence, in this subsection, we evaluate MFDedup and other approaches with the ***Actual Deduplication Ratio*** **(ADR)** defined as $\frac{Total\ Size\ of\ the\ Dataset}{Size\ after\ Running\ an\ Approach}$, which reflects the corresponding reduced deduplication ratio due to these techniques (such as rewriting).

Figure 11 shows ADR of MFDedup, Exact Deduplication, and other approaches, including combinations of rewriting (HAR, Capping and SMR) and GC (Perfect GC and CMA) techniques. Here MFDedup includes its simple GC approach after Arranging. Note that we only retain the latest 20 backup versions in our evaluation, and thus Perfect GC and CMA represent two typical GC techniques using Mark-and-Sweep with utilization thresholds set at 0% and 100%, respectively. Perfect GC reclaims all possible space, while CMA runs faster but leaves unreferenced chunks in containers that are partially referenced, so they show the two extreme impacts of GC.

Generally, Figure 11 shows that MFDedup achieves ADR that is very close to exact deduplication, which is much higher than other rewriting and GC approaches. This is because the space cost of ignoring Skip Duplicate Chunks in MFDedup is quite small, especially compared with the number of rewritten chunks in other approaches.

Figure 11 also shows rewriting techniques cause a decrease in ADR when GC starts after version 21. When the CMA technique (higher GC speed, fewer unreferenced chunks removed) is added, this loss worsens. This is consistent with our discussion in Section 2: rewriting reduces deduplication while GC also can lead to more rewritten chunks. Meanwhile, deletion and GC are naturally supported in our data layout with NDF and AVAR techniques, which has no fragmentation issue and thus no space cost for MFDedup. Overall, MFDedup achieves a 1.12× to 2.19× higher ADR than other approaches due to the proposed data layout.

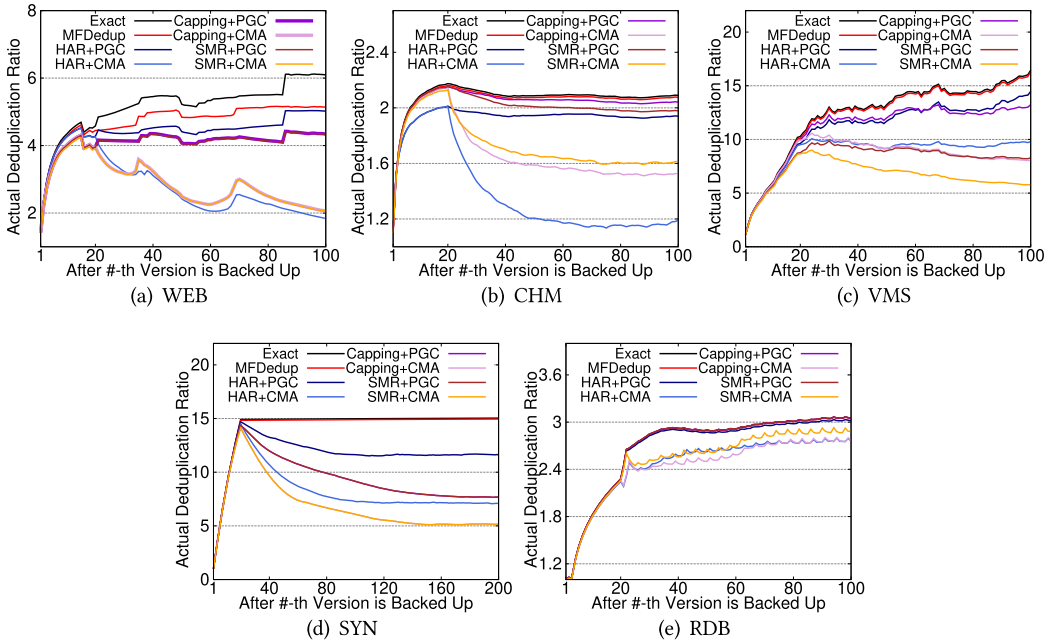


Fig. 11. Actual Deduplication Ratio of MFDedup and five approaches running on five datasets (retaining 20 backups).

5.3 Backup Throughput

In this section, we study the backup throughput of MFDedup compared with rewriting approaches. All of HAR, Capping, and SMR use a full-in-memory global fingerprint index while MFDedup use a NDF-based local fingerprint index. To minimize the performance impact of reading datasets, we back up the datasets from a RAM disk to measure the backup throughput, which is defined as $\frac{The\ size\ before\ deduplication}{Back\ up\ time}$, so logical throughput can be higher than the physical throughput of the storage device. Note that here we only count the time cost of inline steps, so the overhead of Arranging and GC is not considered, since they are offline processes.

Figure 12 shows backup throughput of the four approaches, which have similar results for a given dataset. This highlights that MFDedup does not sacrifice backup throughput to achieve the other benefits we discuss. The performance of the four techniques is similarly limited by the chunking and SHA1 digest calculation. In theory, since MFDedup no longer rewrites duplicate chunks (thus achieving higher Actual Deduplication Ratio in Section 5.2), its storage I/O time when backing up will also be smaller than the traditional design.

Indexing Overhead. During backups, we measured the maximum memory cost for the NDF index, which varied from 6.27 to 46.35 MB (only indexing 2 backup versions). In contrast, traditional deduplication approaches maintain a global fingerprint index for all 20 backup versions and would require 26.81 to 64.45 MB space. Note that the traditional global index grows with the number of retained versions, while NDF only maintains two indices. Moreover, the memory overhead for the NDF indexes could be further reduced by using Bloom filter and prefetching techniques [34, 58].

5.4 Restore Throughput

Previous approaches [21, 30] use **Speed Factor** to measure restore throughput. It is defined as the ratio of useful data restored per container read in deduplication-based backup systems, assuming

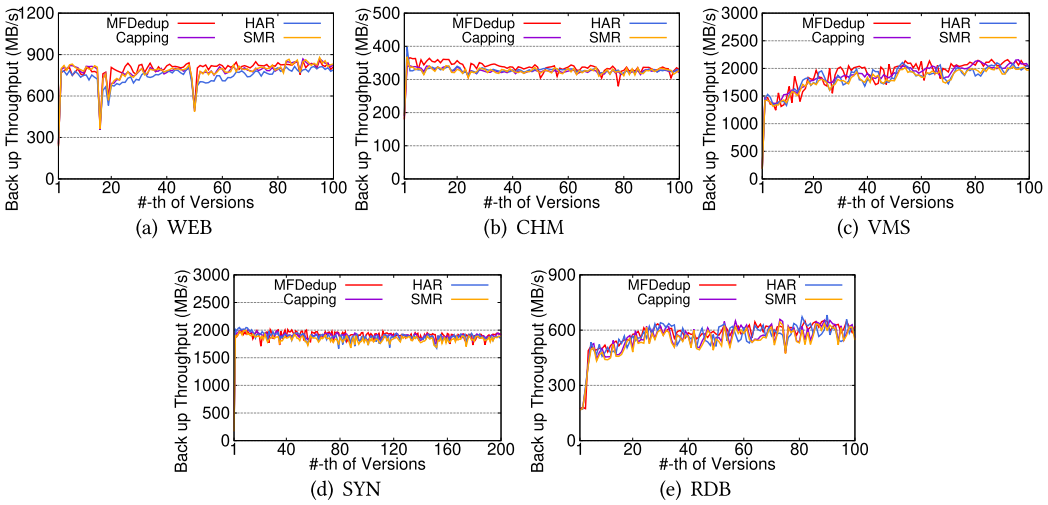


Fig. 12. Backup throughput of MFDedup and two other rewriting approaches running on five datasets, without considering (offline) Arranging.

fix-sized containers as the read I/O unit [30]. Since MFDedup uses variable-sized containers to hold categories as the I/O unit, we define three metrics in this subsection, **Restore Throughput**, **Seek Number**, and **Read Amplification Factor**. Here Seek Number is defined as the number of seek operations required for reading containers/volumes on disk devices while Read Amplification Factor is defined in Section 2.2. Note that for these two new metrics smaller values are better.

Figures 13, 14, and 15 present the restore results of MFDedup, HAR, Capping, and SMR on the three metrics, which demonstrate that Restore Throughput is generally consistent with the other two metrics. HAR, Capping and SMR require container caches (their sizes are configured to be 120 MB in this evaluation) to reduce repeatedly accessing containers for internal duplicate chunks, and MFDedup does not require a container cache. Figure 13 shows that SMR performs better than HAR and Capping for most datasets (i.e., WEB, CHM, SYN, and RDB) for the last several backups, but MFDedup is 10.02× (WEB), 4.19× (CHM), 3.39× (VMS), 3.61× (SYN), and 1.92× (RDB) higher than SMR. This is because MFDedup has eliminated fragmentation by maintaining locality of backup workloads on the proposed data layout, while fragmentation (though alleviated) still exists in HAR, Capping and SMR-based systems and becomes worse with higher versions.

Figure 14 shows the Seek Number on five datasets. MFDedup reduces the Seek Number from thousands for HAR and Capping to 20, which is because it groups several archived categories into one big, sequentially written volume; Capping, HAR and SMR need more seek operations due to their scattered distribution of required chunks.

Furthermore, Figure 15 shows the Read Amplification Factor. MFDedup has the smallest Read Amplification Factor, which is only 48.16% of Capping, 65.04% of HAR, and 44.70% of SMR on average. This is because our data layout has eliminated fragmentation. Meanwhile, HAR, Capping, and SMR will encounter more unneeded chunks in loaded containers when restoring. Read Amplification Factor is less than 1 for MFDedup due to internal deduplication within a backup version (Figure 3), so read chunks can be used multiple times for a restore. Therefore, restore throughput of MFDedup is even higher than the storage media: up to 1.5× of *fread()*, which means MFDedup can completely utilize the performance of storage devices.

这里解释了为什么比fread猛：因为内部重复块读一次就可以用多次

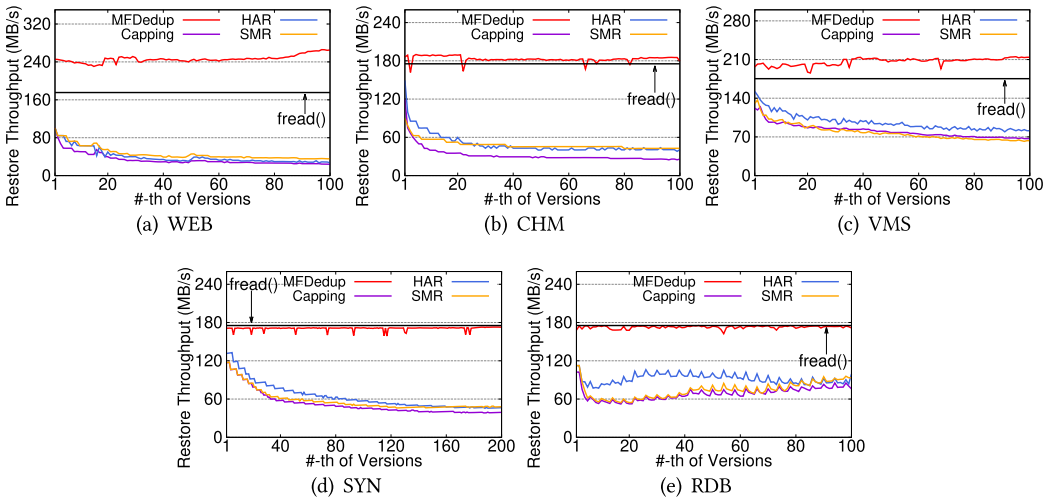


Fig. 13. Restore throughput of MFDedup, HAR, and capping on five backup datasets. *fread()* denotes sequential throughput of the backup device. 跟fread对比这点我感觉挺有意思！很巧妙

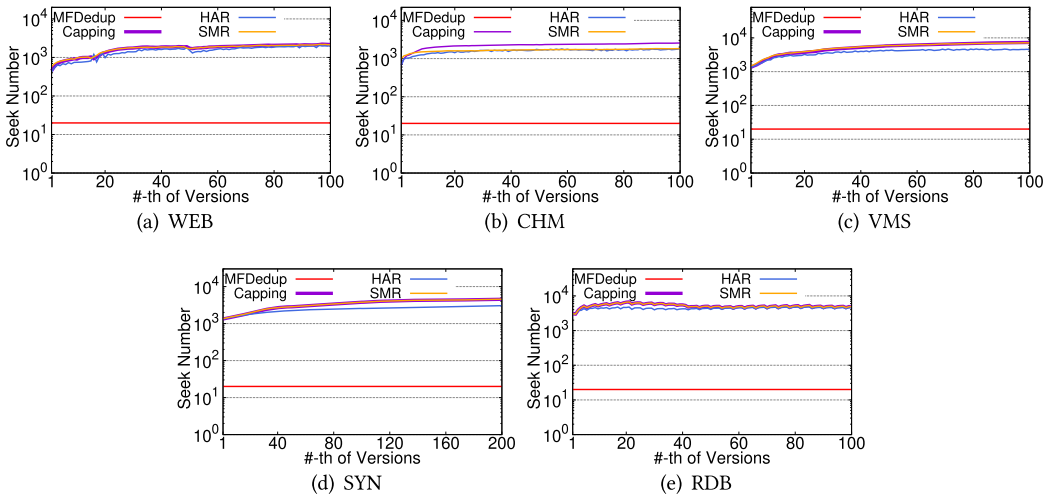


Fig. 14. Seek number of MFDedup, HAR, and capping on restoring five datasets.

Note that these results are also evaluated while retaining 20 backup versions. If we retain more backup versions, then the restore results of HAR, Capping, and SMR will decrease, as is discussed in many previous works [21, 30]. Without fragmentation, MFDedup achieves a consistently high Restore Throughput, even when retaining more backup versions.

5.5 Arranging vs. Traditional GC 我们事实上是将工作转移到了offline，才增快GC

Compared with traditional deduplication approaches, MFDedup has the benefit of nearly zero-overhead GC, but adds the offline Arranging process. In other words, we have transferred background work from GC to Arranging while achieving many benefits: high restore speed, immediate space reclamation, and so on. Therefore, in this subsection, we evaluate the time cost of

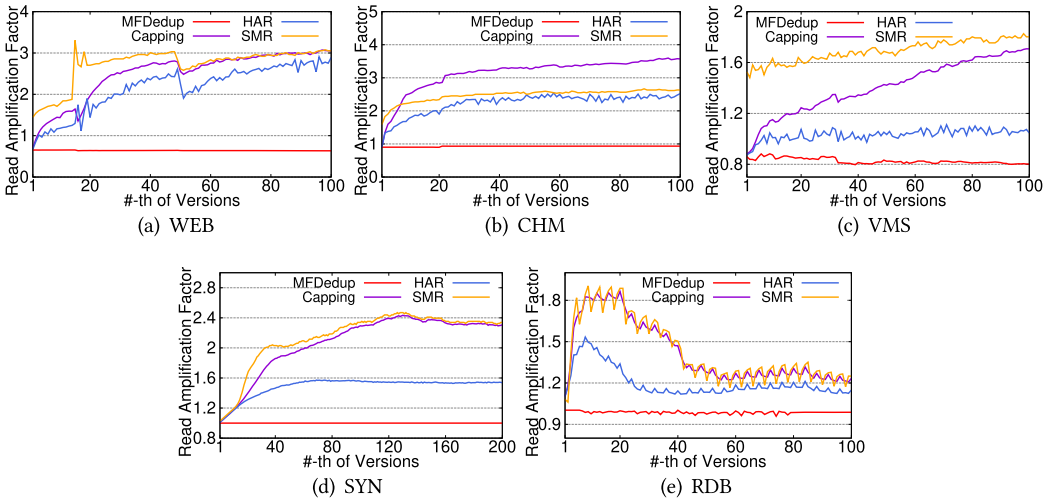


Fig. 15. Read amplification factor of MFDedup, HAR, and capping on restoring five backup datasets.

Arranging in comparison with Perfect GC, which reflects the overhead for updating the proposed data layout in MFDedup.

GC approaches mainly differ in the technique to select the containers and chunks to clean. Once selected though, all the GC techniques involve migrating referenced chunks into new, immutable containers. To simplify our evaluation, we conservatively focus on the cost of reading selected containers and migrating valid chunks into new containers, since that is the common phase. This is a lower bound on the cost of GC, since it neglects the selection phase, which involves enumerating the live files/chunks [15].

The results are shown in Figure 16 comparing Arranging and Perfect GC. Since we are retaining 20 versions, GC does not run for the first 20 versions, though Arranging does. Analyzing the steady-state performance after the 20th version, Arranging's total processing period is only 45% (WEB), 37% (CHM), and 25% (SYN) of GC's total processing time on average. But in VMS and RDB, Arranging takes 9% and 4% longer than GC, because their modification style (always change the same region in each backup in VMS and always append new data after old data in RDB) makes GC very efficient. Generally, Figure 16 suggests that Arranging is usually faster than GC, which would take even more time if the selection phase were included in GC's total. When MFDedup runs its version of GC, the processing time is insignificant, since large Volumes can be deleted at once without any copy-forward. 相比于GC，Arrange还有一个优点就是稳定

Arranging has a consistent processing time across versions, while GC's runtime is more variable, and consistent overheads are easier to plan for in a storage system. Arranging's processing time is consistent, because it is a local process on a recent version, while GC is a global process. As Figure 9 shows, Arranging is always applied in Active categories generated in the same backup version, and it always generates a better locality. However, GC in other techniques suffers from poor locality [23], because the selected containers and chunks are distributed randomly.

Note that other GC approaches will be faster than Perfect GC but at the cost of greatly decreasing Actual Deduplication Ratio as discussed in Section 5.2. In contrast, MFDedup has almost no deduplication ratio loss for GC while supporting immediate deletion and GC, and also achieving nearly perfect restore performance with an acceptable Arranging cost, as shown in Figures 11, 13, and 16.

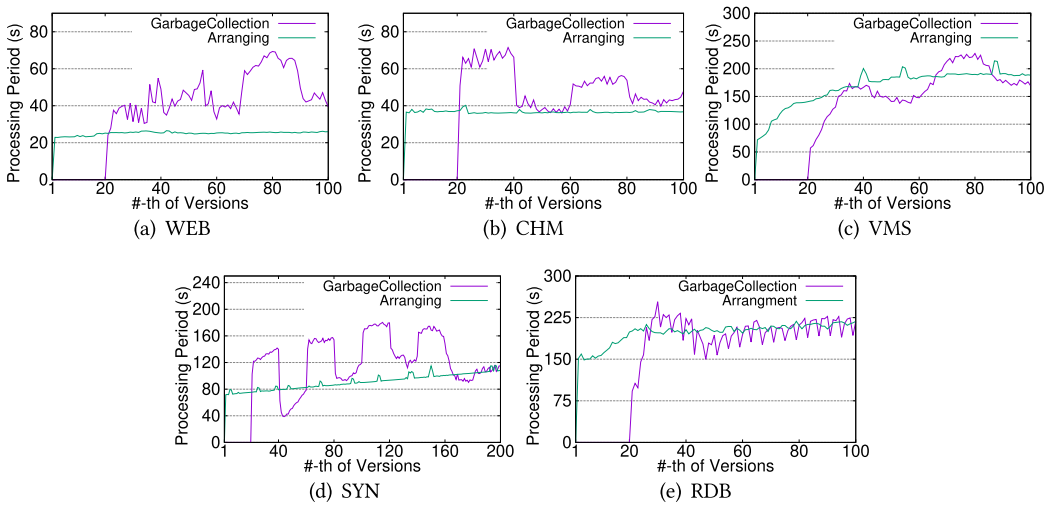


Fig. 16. Time cost comparison between Arranging of MFDedup and GC of traditional deduplication systems.

是的，arrange也是需要数据迁移的

这段话说实话看不大懂

In addition, Arranging is an offline process that traverses active categories of a backup version, migrates duplicate chunks, and archives the remaining chunks. In the article, we always run Arranging after each backup to keep the data layout healthy, but Arranging could act like GC: just running once after several backups. In this case, Arranging falls behind and will cause slight read amplification, as discussed in Section 5.7. Besides, several Arranging tasks, in which duplicate chunks will be migrated several times, could be merged in this situation, which could reduce the total overhead for Arranging, though this has not been evaluated.

5.6 Size Distribution of Volumes/Categories 这个也没看懂

As we discussed in Section 5.5, Arranging is an offline process on active categories (Figure 8), in which chunks are migrated or archived, and it requires additional reserved space.

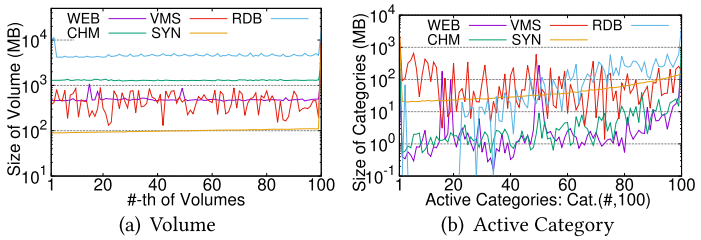
In this section, we demonstrate the data layout of MFDedup with the size of volumes and active categories and study how much reserved space is required. We back up 100 versions without deletion for evaluation. After that, there will be 99 Volumes and 100 active categories, and these active categories compose a logical volume (they will be archived in a volume after the next Arranging).

Figure 17(a) shows the size of Volumes varying from 90 MB to 1.3 GB on our five datasets. The results can tell administrators how much space will be freed by MFDedup by deleting backup versions, and also help administrators estimate how much more can be written to the deduplication system, as discussed in Section 4.6. For previous deduplication systems, it is difficult to answer these issues [45], though sketching approaches have been considered [24, 38].

Figure 17(b) shows the size of active Categories vary over a large range. We learn that the maximum categories hold about 16.99% (WEB), 46.46% (CHM), 18.49% (VMS), 51.87% (SYN), 29.39% (RDB) of the size of the last backup version. This indicates the reserved-space requirement for offline Arranging in MFDedup is much smaller than a full backup. Meanwhile, the reserved space can be further reduced by compressing categories.

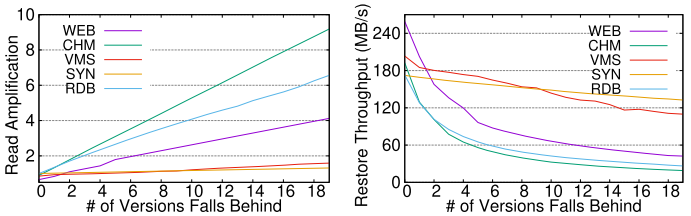
5.7 What if Arranging Falls Behind?

Generally, we expect Arranging will not fall behind, since users usually create full backups daily or less frequently [3, 30], which provides enough time for our offline Arranging. Therefore, Arranging



(a) Volume (b) Active Category

Fig. 17. Size distribution of Volumes and Categories after deduplicating 100 backup versions with MFDedup.



(a) Read Amplification (b) Restore Throughput

Fig. 18. Results of read amplification and restore throughput when Arranging stage falls behind backing up.

falling behind is not a normal case, and might only happen for frequent backups or an overloaded system.

明天要好好想想，怎么做到的“暂停arrange”

In this case, we can skip Arranging temporarily, and apply it in future idle time. The more Arranging falls behind, the more seriously the proposed data layout is damaged, with an increase in read amplification and decrease in restore throughput. In this subsection, we study the impact when Arranging falls behind backups, which provides guidance on how often Arranging should be performed in this case.

According to the results in Figure 13, we learn that read amplification and restore throughput of MFDedup remain nearly the same as the number of backup versions increases. Therefore, to simplify, we only run experiments on the first 20 backup versions of each dataset whose deduplication ratios are 4.58 (WEB), 2.17 (CHM), 10.28 (VMS), 14.91 (SYN), and 3.08 (RDB).

In Figure 18, we let the Arranging stage fall behind by different numbers of backup versions, and we observe how read amplification and restore throughput change when restoring the latest backup. According to Figure 18, we learn that: ① The more Arranging falls behind, the more seriously the flattened layout structure is damaged, with an increase in read amplification and decrease in restore throughput, ② A higher deduplication ratio leads to a smaller read amplification and also smaller reduction in restore throughput. It is because when Arranging falls behind (e.g., the deduplication workflow has processed B_n, but the Arranging has only processed B_{n-k}), we are equivalent to having full backups of $B_1, B_2, \ldots, B_{n-k}$, and “incremental backups” (i.e., unique chunks) of $B_{n-k+1}, B_{n-k+2}, \ldots, B_n$. When restoring B_n, we effectively need to first read the full backup of B_{n-k} and then read these “incremental backups” as patches. Because VMS and SYN have higher deduplication ratios and share more chunks between backups, their “incremental backups” are also smaller and they also have little read amplification in this case.

确实

6 CONCLUSION AND FUTURE WORK

In this article, we propose a management-friendly deduplication framework, MFDedup. Different from traditional “Write Friendly”-style deduplication architectures, MFDedup is designed to

be "Management Friendly" and solves the fragmentation problem in deduplication-based backup systems by introducing a novel deduplication process (NDF) and a locality improvement process (AVAR) to generate our data layout and thus maintain locality of backup workloads.

With the benefits of eliminating the fragmentation problem, MFDedup improves actual deduplication ratios (1.12×–2.19× higher) and restore throughput (1.92×–10.02× higher) beyond previous approaches. MFDedup has an acceptable time cost for the offline "Arranging" to update the proposed data layout, while GC in MFDedup has nearly zero overhead. GC is typically a complex, resource-intensive process in deduplication systems, so eliminating GC is a significant advantage of MFDedup.

Deduplication has been studied for about 20 years, but only a few works pay attention to how we should better place the deduplicated data. While investigating MFDedup, we first discovered the importance of chunks' lifecycle management in deduplication systems, and we designed a special data layout to maximize its benefits and demonstrate its advantages.

As future work, we are considering what insights MFDedup can give to currently popular deduplication systems to improve their lifecycle management. However, chunk-level deduplication usually cannot completely utilize workloads' compressibility because of its processing granularity, and we are considering adding delta compression in MFDedup for further space savings and exploring other ways to reduce the number of categories for general use cases.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their insightful comments.

REFERENCES

[1] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. 2018. Can't we all get along? Redesigning protection storage for modern workloads. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'18)*.

[2] C. Alvarez. 2011. NetApp Deduplication for FAS and V-Series Deployment and Implementation Guide. Technical Report TR-3505, NetApp.

[3] George Amvrosiadis and Medha Bhadkamkar. 2015. Identifying trends in enterprise data protection systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*.

[4] George Amvrosiadis and Medha Bhadkamkar. 2016. Getting back up: Understanding how enterprise data backups fail. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. 479–492.

[5] 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)*.

[6] Tony Asaro and Heidi Biggar. 2007. *Data De-duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations*. The Enterprise Strategy Group, 2–15.

[7] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. 2009. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'09)*.

[8] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. 2000. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (USENIX FAST'20)*.

[10] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. 2019. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (USENIX FAST'19)*.

[11] Zhichao Cao, Hao Wen, Fenggang Wu, and David H. C. Du. 2018. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (USENIX FAST'18)*.

[12] IBM Corporation. 2002. IBM White Paper: IBM Storage Tank—A Distributed Storage System. White paper.

[13] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)*.

[14] Wei Dong, Fred Douglis, Kai Li, R. Hugo Patterson, Sazzala Reddy, and Philip Shilane. 2011. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (USENIX FAST'11)*.

[15] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. 2017. The logic of physical garbage collection in deduplicating storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (USENIX FAST'17)*.

[16] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. 2009. HYDRAstor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (USENIX FAST'09)*.

[17] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. 2012. Primary data deduplication–large scale study and system design. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*.

[18] EMC. 2010. Achieving Storage Efficiency Through EMC Celerra Data Deduplication. White paper.

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

[20] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. 2015. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Trans. Parallel Distrib. Syst.* 27, 3 (2015), 855–868.

[21] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. 2014. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*.

[22] 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)*.

[23] Fanglu Guo and Petros Efstathopoulos. 2011. Building a high-performance deduplication system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'11)*.

[24] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen I. Kat. 2020. Sketching volume capacities in deduplicated storage. *ACM Trans. Stor.* 15, 4 (2020), 24:1–24:23.

[25] Intel. 2016. Intel Intelligent Storage Acceleration Library Crypto Version. https://github.com/intel/isa-l_crypto.

[26] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A reusable networking stack for flow monitoring middleboxes. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*.

[27] Keren Jin and Ethan L. Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR'09)*.

[28] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. 2012. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*.

[29] Yan Kit Li, Min Xu, Chun-Ho Ng, and Patrick P. C. Lee. 2015. Efficient hybrid inline and out-of-line deduplication for backup storage. *ACM Trans. Stor.* 11, 1 (2015), 2:1–2:21.

[30] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (USENIX FAST'13)*.

[31] 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 *Proccedings of the 7th Conference on File and Storage Technologies (USENIX FAST'09)*.

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

[33] T. McClure and B. Garrett. 2009. EMC Centera: Optimizing Archive Efficiency. Technical report.

[34] 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)*.

[35] Dutch T. Meyer and William J. Bolosky. 2012. A study of practical deduplication. *ACM Trans. Stor.* 7, 4 (2012), 1–20.

[36] Jaehong Min, Daeyoung Yoon, and Youjip Won. 2011. Efficient deduplication techniques for modern backup operation. *IEEE Trans. Comput.* 60, 6 (2011), 824–840.

[37] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*.

[38] Aviv Nachman, Gala Yadgar, and Sarai Sheinvald. 2020. GoSeed: Generating an optimal seeding plan for deduplicated storage. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (USENIX FAST'20)*.

[39] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. 2011. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC'11)*.

[40] Young Jin Nam, Dongchul Park, and David H. C. Du. 2012. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'12)*.

[41] 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)*.

[42] Calicrates Policroniades and Ian Pratt. 2004. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'04)*.

[43] Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (USENIX FAST'02)*.

[44] Sean C. Rhea, Russ Cox, and Alex Pesterev. 2008. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'08)*.

[45] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 2016. 99 deduplication problems. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'16)*.

[46] Kiran Srinivasan, Timothy Bisson, Garth R. Goodson, and Kaladhar Voruganti. 2012. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (USENIX FAST'12)*.

[47] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating realistic datasets for deduplication analysis. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*.

[48] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2009. Cumulus: Filesystem backup to the cloud. *ACM Trans. Stor.* 5, 4 (2009), 14:1–14:28.

[49] 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)*.

[50] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. 2019. Improving restore performance in deduplication systems via a cost-efficient rewriting scheme. *IEEE Trans. Parallel Distrib. Syst.* 30, 1 (2019), 119–132.

[51] 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.

[52] 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)*.

[53] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Yucheng Zhang, and Qing Liu. 2016. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*.

[54] Lawrence You, Kristal T. Pollack, and Darrell D. E. Long. 2005. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*.

[55] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. 2015. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'15)*.

[56] 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)*.

[57] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. 2020. DupHunter: Flexible high-performance deduplication for docker registries. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*.

[58] Benjamin Zhu, Kai Li, and R. 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)*.

Received July 2021; revised November 2021; accepted December 2021