SLIMSTORE: A Cloud-based Deduplication System for Multi-version Backups

Zihao Zhang[1], Huiqi Hu[1], Zhihui Xue[2], Changcheng Chen[2], Yang Yu[1], Cuiyun Fu[2], Xuan Zhou[1], and Feifei Li[2]
{zach_zhang, yuyang}@stu.ecnu.edu.cn, {hqhu, xzhou}@dase.ecnu.edu.cn,
{zhihui.xzh, tianyu, cuiyun.fcy, lifeifei}@alibaba-inc.com
[1]East China Normal University [2]Alibaba Group

Abstract—**Cloud backup is becoming the preferred way for users to support disaster recovery. In addition to its convenience, users are deeply concerned about reducing storage costs in the face of large-scale backup data. Data deduplication is an effective method for backup storage. However, current deduplicate methods lack the utilization of cloud resources to provide scalable backup service for cloud backup users, and cannot meet the biased preference for different backup versions. For new backup versions, users want higher deduplicate and restore speed to reduce the waiting time; conversely reducing storage costs is more necessary for older backup versions.**

In this paper, we present SLIMSTORE, with a cloud-based deduplication architecture that disassembles the system into a storage layer and a computing layer to support elastic utilization of cloud resources. We propose two types of processing nodes with different design focuses to meet the needs of cloud-based backup. The L-node exploits locality and similarity and with two history-aware strategies to provide fast online deduplication service. L-node also optimizes online restoration to realize high restore efficiency for new backup versions. Meanwhile, the G-node provides exact deduplication for the old versions offline, and helps the restore performance of the new versions by optimizing their physical storage. We compare SLIMSTORE with some state-of-art deduplicate and restore methods and an open-source system. Experimental results show that SLIMSTORE can achieve fast deduplication, efficient restoration, and effective space reduction. Furthermore, SLIMSTORE achieves scalable deduplication and restoration capabilities.

I. INTRODUCTION

Enterprises used to store backup data with low cost storage such as disks and tape libraries. In recent years, as data scale has increased dramatically and cloud storage has developed, more and more users have chosen to move backup data to the cloud because of its quick and convenient disaster recovery capability. Cloud storage not only means elastic storage capacity, but its flexible pricing based on backup size also saves huge expenses by early storage device investment for users. Therefore, the market for cloud backup services has attracted more and more attention. But how to manage the growing backup data and reduce storage costs is a challenge.

Backup data is usually cold and not accessed frequently, so it is acceptable to adopt storage with low cost and large capacity but slower access speed. *Object Storage Service (OSS)* is a kind of cloud storage that can store and access massive amounts of data from anywhere in the world such as Alibaba's OSS [1] and Amazon's S3 [2]. Due to its extremely low price and large storage capacity, OSS is very suitable for storing infrequently accessed data such as backup data. Although OSS provides storage at a very low price, we still need to explore other ways to further reduce storage costs. The user's backup requirements are long-term and continuous, they tend to upload the latest status of files to the cloud on a regular basis. For instance, database users update the latest snapshots of data every once in a while for rapid disaster recovery. This results in multiple consecutive backup versions stored on the cloud, and incremental modifications cause a lot of duplication between versions. Data deduplication technology can eliminate these duplicate data to reduce the amount of data storage. Therefore, we build a cloud-based backup system, benefiting from OSS's low-cost storage and the reduction in data volume that comes with deduplication.

Data deduplication is a well recognized approach to support large-scale backup storage systems. We observe that three main indicators can measure the deduplication system: deduplication speed, restore speed, and deduplication ratio. It is difficult to perform the best in all three indicators, so most of the existing work only focuses on one of them. Some research works like DDFS [3], SiLO [4], and Sparse Indexing [5] make a trade-off between deduplication speed and deduplication ratio. HAR [6], CBR [7], and Capping [8] rewrite the fragment chunks to gain a better physical locality for better restore performance, but at the expense of some deduplication ratio. In the industry, because enterprises store backup data in their limited local storage, the traditional choices are usually backup methods that maximize the deduplication ratio [9].

Cloud storage makes storage expansion transparent to users, which results in some changes in the trade-off between three indicators. Since the backup data source and cloud storage are physically separated, and accessing OSS is much slower than local disk, so online deduplication and restoration efficiency have become more concerned to reduce the processing time. And we observe that users have biased preferences for the indicators in the old and new backup versions. As for the older versions, their data value decreases over time, so it is hoped that their storage costs can be lower, and the restore process can be tolerated by consuming more time because they are less frequently to restore than new versions. Therefore, the design goals of a cloud-based deduplication system are: for the new backup version, it can quickly deduplicate and restore; for old versions, it can achieve exact deduplication to further reduce storage costs, allowing for a certain amount of restore efficiency sacrifice.

With these two goals in mind, SLIMSTORE is designed to build a cloud-based deduplication system, which provides online deduplicate and restore services. For large-scale, full-volume backup data uploaded by a user at intervals, it eliminates duplicates between versions and supports restoration for any version. SLIMSTORE can perform fast deduplication and restoration for new backup versions while ensuring the effectiveness of deduplication to reduce storage costs. SLIMSTORE separates storage and computation by storing backup data on OSS to gain uncapped storage expansion, and using elastic computing resources to achieve scalable deduplication and restoration. To make a good compromise between deduplication speed and deduplication ratio, SLIMSTORE divides the deduplication into two phases. Firstly, SLIMSTORE exploits the similarity and locality to provide fast online deduplication for the new backup, which reduces the performance loss caused by high latency OSS access. To fully borrow information from previous versions, two history-aware strategies are proposed that use historical information to further accelerate the online deduplication. Besides this, SLIMSTORE performs offline reverse deduplication to accurately identify the missed duplicates to achieve exact deduplication. To keep the restore efficiency of new versions, reverse reduplication removes any duplicate data in old versions instead of that in new versions.

For restoration, the system must combat the fragmentation on its physical storage, especially for the new versions of data. SLIMSTORE also optimizes restoration at two levels. When restoring online, it takes an effective cache with full restore information and a LAW-based (look-ahead window) prefetching method to achieve the highest time efficiency. Meanwhile, SLIMSTORE compacts sparse containers that have few useful data for the new versions in the backend, which can gain a better locality of data layout, thus reducing the OSS bandwidth consumption caused by fragmentation. Both offline actions, reverse deduplication and spare container compaction, reduce the storage overhead of older versions by transferring part of its data to new versions. At the same time, this restructuring will not lose or be more conducive to the restore performance of new versions. Our contributions are summarized as follows:

- We propose a cloud-based deduplication system architecture to realize the design goals. SLIMSTORE separates computing and storage, and makes both of them support elastic scaling. It further decomposes the functionality of the computing layer into high-performance online deduplication and restore services, as well as offline space optimization under the premise of further reducing the storage costs of old versions and ensuring restore efficiency of new version data.
- We propose a hybrid deduplication mechanism. It exploits the similarity between versions to provide fast online deduplication, including two history-aware approaches to improve its efficiency. Meanwhile, offline reverse deduplication is proposed to realize exact deduplication.
- We also holistically optimize restore performance. An effective restore cache is developed to improve the efficiency of online restoration, and sparse container compaction is executed in the backend to further eliminate the degradation of restore performance over time.
- We conduct extensive experiments. Experimental results demonstrate that SLIMSTORE promotes the restoration of new versions to be as fast as old versions. It also outperforms the comparing method by $1.72\times$ in deduplication efficiency. Besides, SLIMSTORE can also achieve scalable deduplication and restoration.

The paper is organized as follows: Section II concludes some related works. Section III descirbes the system architecture. In Section IV, V, and Section VI, we introduce deduplication, restoration, and space management of SLIMSTORE in detail. Experimental results are presented in Section VII.

II. RELEATED WORKS

Many existing works [3]–[5], [10]–[12] adopt chunk-level (e.g., 4KB) deduplication to identify and eliminate duplicates. Xia et.al. [13] divides the chunk-level deduplication workflow into four key stages, namely, chunking, fingerprinting, indexing, and storage management. The storage management includes data restore, garbage collection, etc.

Chunking divides backup data into small chunks so that it can identify more duplicates. Fixed-size chunking is simple but it has a low deduplication ratio due to the boundary-shift problem [14]. Content-Defined Chunking (CDC) can eliminate the impact of boundary-shift, which is the dominating chunking method to achieve a high deduplication ratio. Rabin-based CDC [14] is widely adopted, but the computation of Rabin hash is time-consuming. Gear [15] and FastCDC [16] use a simple hash to reduce the computation cost, which achieves nearly the same deduplication ratio as the Rabin-based CDC, but significantly speed up the CDC process. After chunking, each chunk is calculated a fingerprint by a *cryptographically secure hash signature*(e.g., SHA-1, SHA-256). Two chunks are identified as duplicates if they have the same fingerprints.

The fingerprint index is the key component of deduplication systems to help identify duplicates. In a large-scale deduplication system, it is considered as the bottleneck because its size is overgrowing with the explosive growth of backup data so that it cannot be resident in memory. Many work pays attention to avoiding the bottleneck of fingerprint-lookup on disk. DDFS [3], ChunkStash [12], and Sampled Index [17] use physical locality to accelerate deduplication. When a duplicate chunk is found, they read the entire container(which stores a bunch of chunks) into the cache to find more duplicates. DDFS [3] and ChunkStash [12] store all fingerprints in the index, which can achieve exact deduplication. Sparse Indexing [5], SiLO [4], and Extreme Binning [18] are index methods that use logical locality for deduplication. Sparse Indexing improves memory utilization by sampling representative fingerprints in memory and using champions that share the most representative fingerprints to identify deduplicates. SiLO and Extreme Binning adopt similarity detection to reduce the RAM overhead for indexing, by exploiting the similarity to achieve single on-disk index access for a file or segment(a group of chunks), and use logical locality to enhance deduplication efficiency. DeFrame [19] explores the tradeoffs among deduplication ratio, index's RAM overhead, and restore performance, which provides a good consideration for the design of deduplication index.

After eliminating duplicates, it is desirable to efficiently restore data when a backup version is accessed. An inevitable fact is that restore speed will be hurt because the backup data is physically scattered to many containers, which is known as fragmentation and the restore performance severely declines over time. Existing solutions to this problem is to rewrite the fragmented duplicate chunks [6]–[8], [20]–[23]. Capping [8], CBR [7], and LBW [22] identify fragments within a small range, such as a segment or small part of the buffered backup stream, and rewrite them during deduplication. HAR [6], [23] accurately identifies sparse containers by counting the utilization of each container in the view of the entire backup, and saves sparse containers as historical information to rewrites duplicate chunks in sparse containers when backup the next version. Another research direction to optimize restoration is to design efficient caching policies. Kaczmarczyk et al. [7], Nam et al. [20] used container-based caching. Optimal restore cache [6] uses a look-ahead window(LAW) to collect chunks' sequence to achieve Belady's optimal replacement policy [6]. Some other studies directly store chunks to achieve higher hit ratio [8], [24]. Lillibridge et al. [8] propose a forward assembly area (FAA) to assemble the restored data, it directly copies chunk from container-read buffer to their position in FAA without caching anything. ALACC [24] combines FAA and chunk-based cache to reduce cache management overhead and achieve better restore performance.

Garbage collection (GC) can effectively manage space in deduplication-based backup storage. The state-of-art GC approaches can be generally classified into two categories, namely, reference count and mark-and-sweep. Reference count records the referenced times for each chunk and reclaims chunks with the counter value are zero [11], [25], but it is complicit and suffers from low relability [6], [17]. Mark-and-sweep consists of two stages. Mark stage traverses all chunks and marks the referenced chunks. In sweep stage, the unreferenced chunks will be reclaimed.

Data deduplication is also adopted in some commercial software(e.g., HYDRAstor [26], NetBackup [27], Avamar [28]) to provide enterprise-level deduplication solutions. Open-source deduplication projects, such as Restic [29] is the most popular one with more than 11K stars on GitHub, which is designed for deduplicating on top of the local file system, thereby cannot provide deduplicate service for the cloud. Therefore, in this paper, we focus on building a cloud-based deduplication system named SLIMSTORE, which separate storage and computation to make full use of elastic recourses of cloud to achieve scalable deduplication and restoration. SLIMSTORE exploits the similarity and locality like SiLO [4] and Sparse Indexing [5] for fast online deduplication, and outperforms them through two history-aware strategies. SLIMSTORE also performs global reverse deduplication to accurately eliminate duplicates. To promote the restore performance, SLIMSTORE develops a restore cache with full restore information and LAW-based prefetching, which can realize higher restore efficiency than existing restore caches [6], [17], [24]. SLIMSTORE also performs sparse container compaction offline to adjust the data layout for better restore performance of new versions.

III. ARCHITECTURE

A. Design Features

Separated storage and computation. Many conventional deduplication systems store data and perform deduplicate or restore jobs on the same machine [3], [17] , which limits the number of jobs that a node can carry. Besides, it is superfluous to upgrade compute and storage simultaneously when any one of their resources is insufficient. Decoupling computation and storage is inherent in the cloud scenario. The system should obtain storage of any capacity by storing data on cloud storage, and flexibly allocates computing resources to handle dynamic backup or restore workloads. The system is more cost-efficient because of its elastic expansion capabilities.

Multi-version backups. Our service scenario is that users have continuous backup requirements for full-volume data. The changes between versions are incremental, which means that there are many duplicates between adjacent versions. The system is designed to make the best of this pattern. It mainly eliminates duplicates between versions and generates a recipe for each backup version. The recipe of each file indicates the sequence of chunks in the file. By exploiting the recipe content, the information of the historical version can be used to identify duplicates. When restoring a backup, the system can also restore the original files through the file recipes.

Fast online deduplication and restoration. Our system attempts to provide scalable and fast online deduplicate and restore services for users. Thus we develop the process node named L-node. L-node does not save any state, so it can be quickly deployed and execute backup and restore jobs, which allows the system to dynamically allocate multiple L-nodes to cater to different users' workloads. Therefore, different jobs can be executed in parallel on different L-nodes.

Deduplication systems often suffer from high performance costs because of the frequent access to the fingerprint index. In cloud environment, these operations are extremely onerous since the index is placed on OSS. Thus L-node turns into a lighter method by avoiding frequent access to data on the cloud. L-node detects a historical version or similar file for each backup file, by fetching the recipe and exploiting the similarity and locality in the detected file, duplicates can be identified fast, thus avoiding a lot of OSS access. Besides, historical information can also facilitate the deduplication.

As for restore performance, considering the fact that chunks of backup are physically scattered after deduplication, especially for new backup versions, which results in the restore performance degrades over time. However, new backup versions are more likely to be accessed, and users expect fast speed to restore them. We believe improve the physical locality for newly generated versions by sacrificing their deduplication ratios (e.g., achieving fast restoration by not deduplicating the file) is not a desirable solution. Because the user's restore needs for those versions are unpredictable, and a version does not need to be restored in many cases. Therefore, it is not wise to overstore all versions of data. SLIMSTORE improves the restore performance of new versions through two efforts. First,

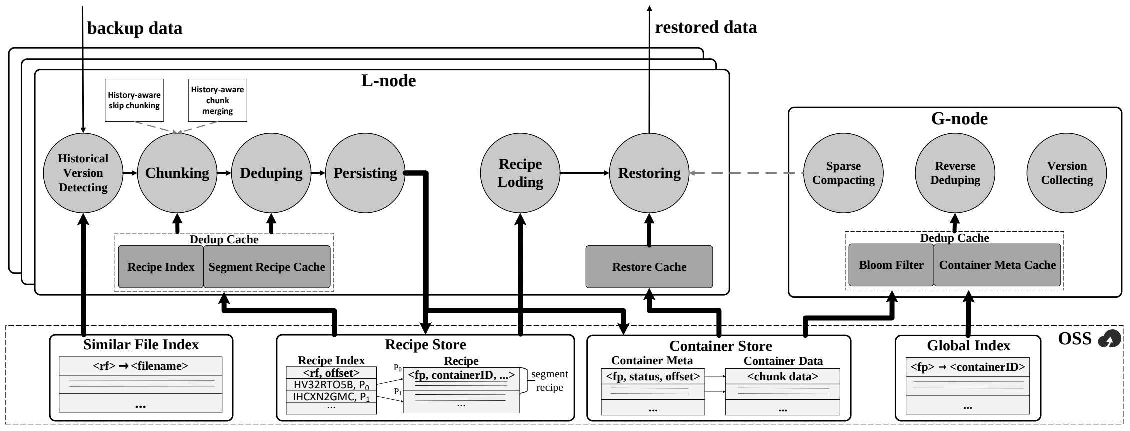


Fig. 1: System architecture of SLIMSTORE.

we design a restore cache with full restore information and LAW-based prefetching to provide efficient online restoration. On the backend, we compact sparse storage for the new version without sacrificing deduplication ratios and additional storage costs, as described in the next paragraph.

Offline storage space optimization. We further propose the G-node for two purposes. The first is to achieve a high deduplication ratio and save the storage cost. Because fast deduplication on L-node may ignore some duplicates, G-node augments the deduplication ratio by further filtering the results with a global fingerprint index, thus achieving exact deduplication of all backup files. And for another purpose of protecting the restore efficiency of the new version from impairing, reverse deduplication is performed to accurately eliminate duplicates in old versions. To make the system more effective for new version data, G-node also optimizes the restore efficiency for new versions by adjusting their physical storage structure with sparse container compaction, it will adjust the data layout by transfer part of data in old versions to new versions to promote the locality of the latter, which improve the restore performance of new versions and reduce the storage cost of old versions. All the actions on G-node are performed offline, independent of online deduplicate and restore performance.

B. System Components

Fig.1 provides the architecture of SLIMSTORE. The system is separated into a storage layer and a computing layer.

1) Storage Layer: The storage layer resides on OSS, it stores backup data, metadata, and indexes.

Container Store. Accessing a chunk from storage at once is not cost-effective for I/O, especially from remote OSS. A common solution is to treat the container as the basic storage and access unit of backup data [13]. While duplicate chunks are eliminated, the remaining non-duplicate chunks will be aggregated into fixed-size containers and persisted on OSS. To access a chunk, the container store also retains the metadata of each container, which keeps each chunk's status and offset, and the proportion of stale chunks (the usage is shown in Section V) in the container. The container based storage gives rise to the *physical locality*. Since a container is a collection of physical chunks that may have a close position in the backup file, once a chunk is accessed, other chunks in the container are also likely to be accessed. Therefore, accessing a container each time can reduce the number of access to OSS.

Recipe Store. Recipe is the data structure that describes the logical sequence of chunks of a backup file. A recipe consists of chunk records, and each chunk record is stored as a quadruple $\langle fp, containerID, size, duplicateTimes \rangle$, which represents the fingerprint of the chunk, the container ID that stores the chunk, the chunk size, and the number of times that the chunk was confirmed as duplicate in historical versions of the file (seen usage in Section IV). There is a *logical locality* embedded in the recipe. Due to the incremental changes of the backup files, the chunk sequences of two backup versions are similar. Thus we exploit the structure called *segment* to make use of the property for deduplication. In the backup file, a number of consecutive chunks constitute a segment. Their corresponding chunk records in the recipe then constitute the *segment recipe*. Based on this, we can speculate that there are many similar segments between two close versions of backup. To quickly match the similar segments and locate its segment recipe, a recipe index is constructed for the recipe of each file. In the recipe index, we extract several representative fingerprints for each segment as samples and map them to the offset of their segment recipe in the recipe.

Similar File Index. Similar index stores the representative fingerprints of each file, which is used to find similar files. Accord to Broder's theorem [30], the similarity of the full set is highly dependent on the similarity of two randomly sampled subsets. A file can be considered as a set of fingerprints, so if two files share some representative fingerprints, they are considered similar.

Global Index. Global index maintains the information of all chunks of a user, it saves the mapping from the fingerprint of chunk to the container where it is stored. Global index is stored in Rocks-OSS, which is a RocksDB that is adapted to suit the OSS. Global index will be used for G-node to accurately identify duplicates in the global scope.

2) Computing Layer: The computing layer is composed of Alibaba cloud elastic compute services (ECS) with two types of nodes: L-node and G-node.

L-node. L-node services online jobs including the backup and restore jobs. When the backup command reaches the L-node, it starts to receive the input file stream and deduplicate it. The L-node first detects a historical version or a similar file via the similar file index. Then it fetches the recipe index of the detected file from the recipe store. With the help of recipe index, the recipes of similar segments are fetched, and L-node exploits the logical locality in them to remove duplicates. Two optimizations named *history-aware skip chunking* and *chunk merging* are further proposed to improve its efficiency (see details in Section IV). As for the restore job, L-node first loads the recipe of the target file, and then reads chunks from container and splices them together based on the sequence of chunk records in the recipe (see details in Section V).

Noting that L-node does not save any state, all the information required in backup and restore is loaded during the job execution, and both the fetching of the recipe index and segment recipes are lightweight. Thus L-node can expand elastically according to the running workload.

G-node. G-node runs offline. G-node is responsible for managing the storage space. Because L-nodes use similar segment detecting and logical locality to identify duplicates, there may be some duplicate data that has not been identified. In order to further identify duplication and save storage space, G-node uses reverse deduplication to filter containers generated by L-node, find and eliminate duplicate chunks (Section VI-A), to achieve exact deduplication. At the same time, G-node will also compact the sparse container identified to ensure that the backup file has a better physical locality for the newer versions, which improves the restore efficiency (Section V-B). Besides, G-node collects the deleted old version to reduce the space occupied by the old version (Section VI-B).

IV. Deduplication on L-Node

We first introduce the process of online deduplication, then two techniques are further proposed to enhance it.

A. Deduplication Workflow

An input file stream will be deduplicated in three steps.

Step 1. Detecting a historical version or similar file. For each input backup file, the latest historical version will be searched first by file path and file name. Examining the file name is simple and effective. However, it doesn't always match because sometimes users change their file names. In that case, the input file will be chunked and sampled, and use the sampling fingerprints to look for a potential historical version or similar file by querying the similar file index. We use the straightforward random sampling method adopted in many deduplication works [5], [19], which selects the fingerprints that $mod\ \mathcal{R} = 0$ in a segment, where $\mathcal{R}$ is an adjustable parameter to control the sampling ratio. It is impractical to process the entire input file that failed to match by name because it is difficult to save all chunks of a large file in memory. Therefore, the common solution for large files is to only sample the header chunks [18]. If the historical version or similar file is detected, L-node will fetch the recipe index of the detected file. For those files without historical versions and similar files, all chunks will be treated as non-duplicate.

Step 2. Prefetching similar segment and deduplicating. After fetching the recipe index of the historical version or similar file, the input file will be chunked and sampled. The sampling method is the same as introduced in Step 1. For each sampled chunk, it looks up the recipe index to find the similar segment. If a chunk with the same fingerprint exists, it prefetches the corresponding segment recipe and adds it to the dedup cache. Once a sampled chunk is matched, other chunks near it will also appear in this segment with a high probability because of the logical locality. By using this feature, a range of duplicate chunks in the vicinity can be filtered efficiently. During the process, the metadata of a chunk including its fingerprint, size, container ID, and duplicated times is generated.

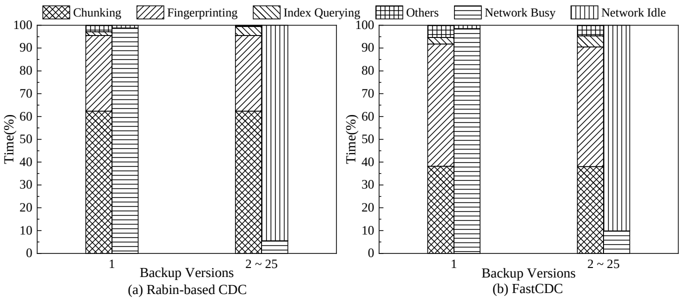


Fig. 2: CPU and network time breakdown of CDC.

Step 3. Segmenting and persisting. Based on the sequence of chunks in the input file, a number of consecutive chunks will be packed into a segment. Once a segment is processed, those non-duplicate will be stored in the new container. When the capacity of a container reaches the upper limit, it will be directly persisted into the container store on OSS. The metadata of all the chunks in the segment will form the segment recipe, which is appended to the recipe in the recipe store after containers are persisted. Meanwhile, the fingerprints of the sampled chunks and the offset of the segment recipe will be preserved and eventually made into the recipe index.

B. History-aware Skip Chunking

Content-defined chunking (CDC) is the dominating chunking method for deduplication due to its high deduplication ratio, but it is compute-intensive and time-consuming. Essentially, the CDC algorithm needs to scan the file byte-by-byte by scrolling a fixed-size sliding window. Each time the window advances one byte, the method needs to compute the hash value of the data in the window, and inspect whether the position is a cut point when the hash value meets certain conditions. These operations for each byte shift are expensive, especially for the classic Rabin-based CDC [14] due to the complexity of Rabin hash. Some other algorithms such as FastCDC [16] use a simpler hash function, but running the byte-by-byte checking mechanism is still inefficient.

In Fig 2, we divide CPU time into four parts, chunking, fingerprinting, index querying, and others. We also monitor network usage to determine system bottlenecks. For the first backup version, the network will be the bottleneck because almost all data needs to be transmitted to OSS. For subsequent versions, a large amount of deduplicates are removed, resulting in less data upload, which makes the replacement of network to CPU becoming the new performance bottleneck. According to the break down of CPU time, chunking and fingerprinting

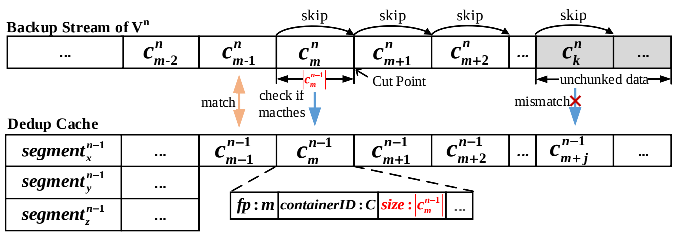


Fig. 3: History-aware skip chunking.

consume the major CPU resources. Rabin-based CDC occupies about 60% of the CPU time. As for FastCDC, despite the optimization of computing overhead, it still accounts for almost 40% of the CPU consumption. Because the fingerprinting algorithm must be sufficiently secure to avoid hash collisions, the CPU overhead of fingerprinting is inevitable. We will focus on reducing the overhead of chunking.

Considering the incremental modification between backup versions, many consecutive duplicate chunks exist between two versions. Therefore, we can speculate that if a chunk is duplicated with a chunk of the previous version, the next chunk is likely to be recognized as a duplicate. By using the historical information in the recipe of the previous version, we can try to skip some bytes to the next promising cut point, thus avoiding the CPU consumption of byte-by-byte checking if the cut condition is met after skipping. We name this CDC acceleration method as *history-aware skip chunking*. Once a chunk is identified as duplicate, we look up the size of the next chunk in the dedup cache, and skip to the cut point based on the size. If skip chunking succeeds (i.e., the new chunk is duplicate), continue to skip to the next cut point, otherwise, turning off skip chunking and continues chunking by the CDC algorithm until the next time a chunk is identified as duplicate.

Fig 3 describes the process of history-aware skip chunking. The n-th version V^n of a file is being backed up, where chunk c^n_{m-1} matches the chunk in $segment^{n-1}_x$ of V^{n-1} and identified as a duplicate. At this time, the size of the next chunk $|c^{n-1}_m|$ can be obtained through the information stored in $segment^{n-1}_x$, then the current version directly skip $|c^{n-1}_m|$ bytes to cut the next chunk. If the position after skipping meets the cut condition, the skip chunking is successful, thus avoiding the CPU consumption of scrolling the sliding window, thereby greatly accelerating the chunking. In addition, when the skip chunking is successful, the fingerprint of the new chunk c^n_m can be directly compared with chunk c^{n-1}_m of the V^{n-1} to verify whether it is duplicated, thereby avoiding searches in the dedupe cache to determine duplicates, so the deduplication speed is further accelerated. The experimental results in Setcion VII-B show that history-aware skip chunking can reduce the CPU consumption of CDC to 2%, which significantly improved deduplication performance.

C. History-aware Chunk Merging

Chunk size has a direct impact on the deduplication speed and deduplication ratio. Adopting a small chunk size can find more duplicates, but it also means that there are more chunks, which increases the overhead of chunking and querying indexes, thereby declining deduplication speed. As for

Algorithm 1: SuperChunking

```
Input: new chunk $c^n$, $c^n$'s start postion $p_0$, $c^n$'s end
       postion $p_1$, the duplicate chunk $c^{n-1}$
/* start SuperChunking when $c^n$ is
   duplicate with $c^{n-1}$                              */
Function SuperChunking()
    if $c^{n-1}$ is the first chunk of superchunk $sc^{n-1}$ then
        $|sc^{n-1}| \leftarrow size\ of\ sc^{n-1}$;
        $sc^n \leftarrow new\ SuperChunk()$;
        $sc^n.data \leftarrow file[p_0, p_0 + |sc^{n-1}|]$;
        $sc^n.fp \leftarrow SHA\text{-}1(sc^n.data)$;
        if $sc^n.fp == sc^{n-1}.fp$ then
            $sc^n.isDuplicate \leftarrow true$;
        else
            $c^n.isDuplicate \leftarrow true$;
            $start\ CDC\ from\ p_1$;
```

large chunk size, it leads to a reverse effect. Because the characteristics of user backup data are varied, it is difficult to determine an appropriate uniform chunk size to ensure high performance and high deduplication ratio at the same time. For example, for a range of data stream with a high duplication ratio, it means that it contains many consecutive duplicate data, so a large chunk size can be adopted to speed up deduplication. But for data with a low duplication ratio, data changes may occur many times, so it is suitable to use a small chunk size to find more duplicates. Therefore, we intend to propose a method which can tune the chunk size according to the characteristics of data, which improves the deduplication efficiency without losing the deduplication ratio.

The duplication ratio for an initial backup file is unknown. To ensure a high deduplication ratio, a small chunk size, such as 4KB, can be used for the first deduplication. But to achieve adaptive chunk size, we need to track the deduplication ratio of the file. Therefore, for each chunk, we use the attribute *duplicateTimes* to record the historical duplicate times in recipe, every time the chunk is identified as a duplicate, *duplicateTimes* increased by one; as for unique chunk, it is set to zero. In the deduplication process of the subsequent version, consecutive chunks whose *duplicateTimes* reaching the threshold are merged into a large chunk, which can speed up the deduplication of the future backup. Because the merged chunks are duplicated in a long period, which means that the probability of this range of data being modified is low, therefore, it is reasonable to use a large chunk size to accelerate the deduplication. The large chunk after merging is named as *superchunk*, and the merging strategy based on historical duplicate times is called *history-aware chunk merging*.

Because the superchunk is merged by a number of chunks, it cannot be obtained by the CDC algorithm. We propose a method to use superchunk. Considering that if two superchunks are duplicates, the first chunk they contain must also be duplicates, so the first chunk can be used to match the superchunk. The meta-information of superchunk stored in recipe has an additional attribute *firstChunk* to record the fingerprint of the first chunk it contains. If a chunk is duplicate with the *firstChunk* of a superchunk, it will start to verify whether there is a superchunk. Algorithm 1 describes the process of SuperChunking. When a new chunk c^n is duplicate with a chunk c^{n-1} of version V^{n-1}, first check whether c^{n-1} is the first chunk of a superchunk sc^{n-1} (line 2), if so, the size of sc^{n-1} is obtained as $|sc^{n-1}|$ (line 3), then skip $|sc^{n-1}|$ bytes to cut a new superchunk sc^n and calculate its fingerprint (line 4-6), then verify whether the fingerprints of the sc^n and sc^{n-1} are the same (line 7). If so, SuperChunking is successful, a duplicate superchunk is identified (line 8); otherwise, it is failed to cut a superchunk, then go back to the current cut point p_1, which is also the end position of c^n, and continue to do chunking by CDC algorithm (line 10-11).

According to our analysis, the overhead of superchunking is narrowed to the overhead of cutting the first chunk by the CDC algorithm, which significantly improves chunking efficiency. Meanwhile, because the total number of chunks is reduced by several times after chunk merging, this means that the overhead of persisting and prefetching recipes is also reduced by several times, which further accelerates deduplication. As for data with low duplication, they are still chunked with a small chunk size. Therefore, by adopting history-aware chunk merging, the chunk size is variable according to the characteristics of data, which can make a good compromise between deduplication speed and deduplication ratio.

V. Restore

As mentioned in Section III-A, restore performance suffers from the read amplification caused by fragmentation, which wastes a lot of OSS bandwidth. Meanwhile, restore job spends much time waiting for reading data from OSS. Therefore, we focus on optimizing restore performance in terms of low OSS bandwidth consumption and high time efficiency.

A. Restore Cache

LAW-based prefetching. The restore job needs to read all data from OSS, but the high access latency of OSS causes very low restoration efficiency. Although restore cache can reduce the frequency of OSS access by caching chunks for future access when reading a container from OSS, it still needs to wait when the container is accessed for the first time. Compared to the local file system, a remote I/O blocking on restoration can significantly reduce its efficiency. The best way is to avoid this I/O blocking completely by prefetching. If all the chunks being processed happen to be in the memory of L-node, the restore job can achieve the highest time efficiency. We introduce a look-ahead window (LAW) to guarantee this by exploiting the chunk sequence in recipe. Backend threads are executed to prefetch containers in the window (which will be accessed soon), so that chunks are loaded into memory before restoring them. For example, when restoring chunk G in Fig 4, the containers to be accessed in the window are C_8 and C_4 to restore chunk U, V, J, and K. By adopting LAW-based prefetching, backend threads have already read these two containers and fill the chunks to be used into the restore cache. Then when restoring these chunks, they can all be found in

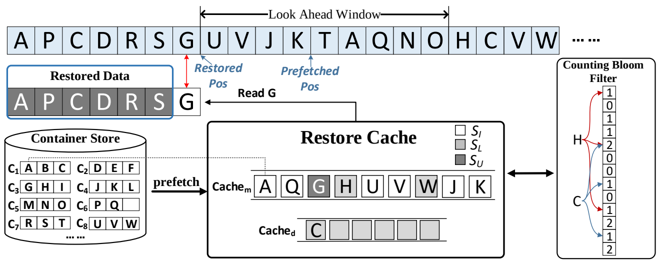


Fig. 4: Restore cache with full vision.

the cache without waiting for reading from OSS, thus greatly speeds up the restore pipeline.

To ensure the highest time efficiency, the *prefetched pos* must be ahead of *restored pos*, which means that the prefetch speed must exceed restore speed. Fortunately, OSS can support multi-channel parallel read that achieves scalable performance improvements, despite its single-channel read throughput is relatively low. Thus SlimStore enables multithreading for prefetching. Results (see SectionVII-C) show LAW-based prefetching is extremely helpful. In our implementation, when prefetch thread number reaches 6, prefetch speed always exceeds restore speed, which means that all the required chunks are in memory, SlimStore does not spend time waiting for reading from OSS, so it achieves the highest time efficiency.

Full vision replacement policy. Due to the fragmentation issues, the conventional cache replacement algorithm like LRU has poor performance. For example, considering the LRU cache size can hold up to 3 containers, when restoring the data stream in Fig 4, container C_6 is read to restore chunk P, but chunks in C_6 are too scattered, a cache miss will occur when restoring chunk Q, because between P and Q, six different containers are filled into the cache, causing repeated reading of C_6. We call this kind of container as *large-span container*. Another fragment that may cause repeated reading is like chunk A, which appears multiple times in the data stream. When the second chunk A is restored, container C_1 has been evicted, results in C_1 needs to be read again. This phenomenon is called *self-reference chunk* [6]. The read amplification caused by repeated reading wastes OSS bandwidth. Existing works [6], [24] use a look-ahead window(LAW) to preserve these two kinds of fragments that in the window in the cache, such as chunk Q and A, which reduce the impact of them.

However, the limited size of LAW will not prevent fragments that out of LAW from being evicted, such as chunk H and C, because they are not in the vision of LAW. To address this problem, we design a restore cache with a full vision replacement policy, which is based on the full information of chunk sequence in recipe, to protect chunks that will be accessed in the future (includes the large-span and self-reference fragments that out of LAW) from being evicted, and make sure all containers only be read once, which completely avoiding repeated reading from OSS.

Fig 4 shows our restore cache. We established a counting bloom filter (CBF) for each file to record the chunks it contains, which is efficient to test whether a chunk is included in the restoring file. CBF can also count the referenced times of each chunk, and once a chunk is restored, its count decrement accordingly. By only evicting the chunks with a zero count, the chunk that out of LAW can also be preserved. There are three statuses for chunks: chunks appear in LAW are marked as S_I (e.g., chunk U and V), which indicates that they will be used soon; chunks only exist in CBF are marked as S_L, means that they will be accessed in the future (e.g., chunk H, C, and W); others that not appear in LAW and CBF are useless chunks and marked as S_U, like chunk G has been restored and does not appear in the future. To avoid useless chunks occupy cache space, when a container is read, only useful chunk (with status of S_I or S_L) is placed in the cache. When replacement occurs, the chunk with a status of S_U is swapped out. In extreme cases, cache may all be occupied by useful chunks, evicting them will cause repeated reading from OSS to restore them later. Therefore, the restore cache is designed as a two-layer, $Cache_m$ is kept in memory, and $Cache_d$ resides on the disk of L-node. When all chunks in $Cache_m$ are useful, the chunk with status S_L will be swapped into $Cache_d$ because it will not be used soon. Before it is accessed, it will be swapped back to $Cache_m$, thus avoiding the OSS access overhead caused by directly evicting some useful chunks.

B. Sparse Container Compaction

We observe that three kinds of fragments may cause read amplification. Both *large-span container* and *self-reference chunk* can be handled by the full vision cache as described in Section V-A. There is another fragment needs to be noticed, which is named as *sparse container*. Since the duplicate chunks in multiple backup versions are eliminated, the chunks of a new version are scattered among many containers, so *sparse container* may only has few chunks that are useful for the new version. For example, in order to restore chunk D in Fig 4, container C_2 must be read. However, there is only one chunk in C_2 is useful, resulting in a large number of invalid reads. The read amplification caused by *sparse container* wastes a lot of bandwidth read from OSS.

To eliminate the impact of sparse containers, SLIMSTORE compacts useful chunks in sparse containers to gain a better physical locality for new versions. We measure the container utilization as $\frac{number\ of\ useful\ chunks\ in\ the\ container}{total\ chunk\ number\ of\ the\ container}$. During the deduplication, the utilization of each container referenced by the backup file of the current version is calculated, and the container whose utilization is lower than the threshold (e.g., 30%) is recorded as the sparse container. After the current backup is finished, G-node starts the *sparse container compaction*(*SCC*) phase, merges chunks of sparse containers that are referenced by the backup file into new containers, and updates the file recipe to the new state. After compaction is completed, the restore job based on the new recipe will eliminate the impact of sparse containers. The benefit of SCC is directly applied to the current version, instead of taking effect in the next version like HAR [6]. Besides, the compacted chunks that in sparse containers will be deleted after compaction, which means that SCC transfers some data of old versions to be stored in the new version, so the storage cost of old versions degrades over time, which meets our design goal that spends less money for old backup data.

VI. SPACE MANAGEMENT ON G-NODE

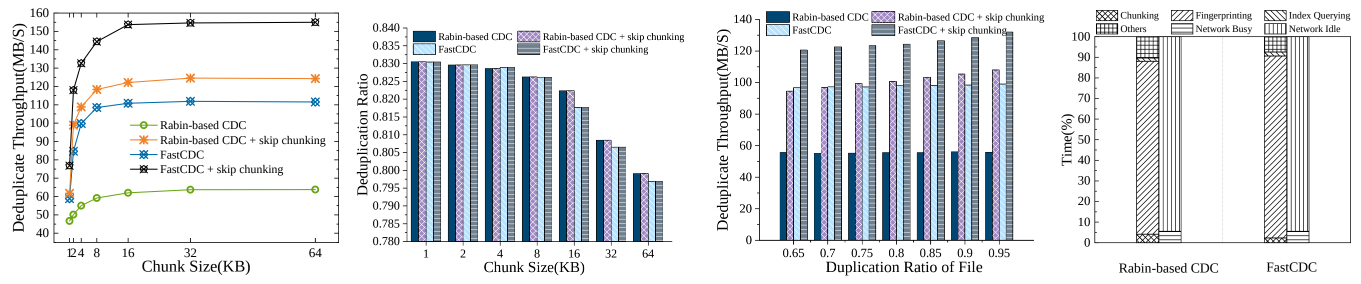
G-node works on the backend to make the storage more space-efficient by tuning the physical storage of containers. Meanwhile, the adjustment needs to be more conducive to the restoration of new versions. While the SCC technique mention in Section V-B caters to this principle, G-node further provides a global reverse deduplication technique. G-node also collects deleted old versions to reclaim the occupied space.

A. Global Reverse Deduplication

Fast deduplication on L-node can not achieve exact deduplication, so some deduplicates still exist between versions, accurately identify and remove them can maximize the deduplication ratio and reduce storage costs. Considering that eliminating duplicate chunks that have already been stored in containers may destroy the layout of the container, which means that the deleted chunks need to redirect to other containers, thus exacerbates fragmentation. Therefore, choose which copy of the duplicate chunk to delete is important, because it will degrade the restore performance of the version that references the deleted chunk. With the design goals in mind, SLIMSTORE needs fast restoration for new versions and low storage costs for old versions, so *reverse deduplication* is adopted. By preserving the data layout of the new version and deleting the duplicate chunk in containers of the old version, reverse deduplication reduces the data volume of old versions without sacrificing the restore performance of the new version.

The global index described in Section III-B is used to accurately identify duplicates. During the backup, L-node records all newly generated containers, and G-node initiates a backend job to filter all chunks in new containers to find if there is a duplicate stored in a container of the old version. If so, reverse deduplication starts to delete the duplicate chunk in old container, and update the location of the chunk in global index to the new container. So as to speed up filtering, a global bloom filter is used to quickly filter out unique chunks. Besides, when two chunks are identified as duplicates, based on the physical locality of the container, there may be other duplicates in two containers. Therefore, caching the meta of the old container can also reduce the access number of Rocks-OSS to accelerate global deduplication. Because the overhead of reading and updating the entire container each time to delete a chunk is unbearable, so global deduplication only marks the duplicate chunk as deleted in the meta of old container. When the number of deleted chunks in a container exceeds the threshold (such as 20%), the container is read out and invalid chunks will be removed, and then rewritten to OSS.

Global deduplication has no impact on the deduplicate and restore jobs on L-node because it is executed offline and the recipes of the latest version are kept unchanged. However, reverse deduplication deletes duplicate chunks of old versions, which may cause extra query of the global index to find the deleted chunks when restoring old versions. But we think it is worth it, because the space occupied by old versions is reclaimed and the restore performance of the new version which is more likely to be restored is also guaranteed.



(a) Throughput: vary chunk size (b) Deduplication ratio: vary chunk size (c) Throughput: vary file characteristics (d) Time breakdowm

Fig. 5: Performance of history-aware skip chunking.

TABLE I: **The characteristics of dataset.**

Dataset name	S-DB	R-Data
Total size (TB)	2.44	1.53
# of versions	25	13
# of files	500	7440
Average duplication ratio	0.84	0.92
Self-reference	20%	0.1%

B. *Version Collection*

In the backup system, timely reclaiming invalid versions can manage storage space more effectively. After a backup version becomes invalid, the container only referenced by it is recognized as a *garbage container*, which can be collected by the system. In SLIMSTORE, there are two categories of garbage containers. One is the container that referenced in version *N* but not referenced by version *N+1* and other similar files. This kind of container is invisible to subsequent versions and can be collected when deleting version *N*. During deduplication, by comparing containers in two backup versions, this type of container can be quickly identified. The other garbage containers come from the sparse container. Because sparse containers identified by version *N* will not appear in subsequent versions, so these containers are converted to garbage and associated with version *N*. Essentially, the *Mark* phase in garbage collection is performed when deduplicating each version. In this way, when the version is reclaimed, only the *Sweep* phase is performed by deleting the garbage containers associated with this version, which significantly accelerates the speed of version collection and will not stop the system from executing other jobs.

VII. EVALUATION

A. *The Experimental Setup*

We deployed SLIMSTORE on a cluster of seven cloud elastic compute services (ECS), each one is equipped with a 2.50GHz Intel Xeon processor with 16 cores and 64GB memory. We use six ECS as L-nodes and one ECS as G-node. And the cloud storage we adopt is Alibaba's OSS [1].

We implemented SiLO [4] and Sparse indexing [5] as our competitor to evaluate the performance of fast online deduplication on L-node, both of these methods use the similarity and logical locality to identify duplicates. We also implement HAR+OPT cache [6] and ALACC [24] to demonstrate the effectiveness of our optimization on restore process. HAR is the rewriting method that can most accurately identify sparse containers, and OPT cache is a LAW-based container cache. ALACC is the most state-of-art restore cache that combines FAA and chunk-based cache to improve restore performance. To comprehensively evaluate our design, we also compare SLIMSTORE with Restic [29], which is the most popular open-source deduplication system on GitHub. By using OSSFS (a tool that can operate OSS like the local file system) [31], we replace Restic's storage with OSS.

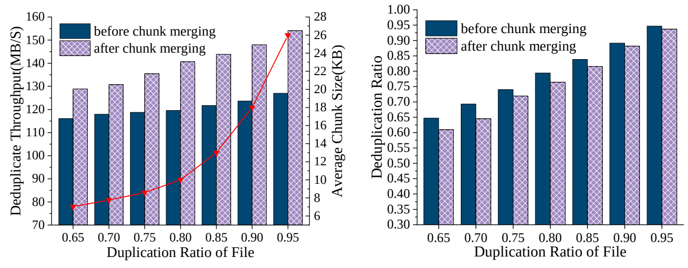
We use two datasets for evaluation as shown in Table I. S-DB is a set of database files, and each table is simulated by the insert, update, and delete operations. By adjusting parameters, we can control the percentage of the modified data, thereby varying the duplication ratio of each table file between versions from 0.65 to 0.95, and the average duplication ratio between versions is 0.84. R-Data is a real backup dataset of an enterprise. It contains 13 backup versions, and a total of 1.53TB of data.

B. *Deduplicate Performance*

We evaluate the deduplicate performance on L-node with S-DB to demonstrate the effect of fast online deduplication. Deduplicate throughput shows the speed of deduplication, which represents the amount of data processed per second; the deduplication ratio represents the effectiveness of deduplication, it is measured in terms of the percentage of duplicates deleted after deduplication, i.e., $\frac{the\ size\ of\ duplicate\ data\ deleted}{total\ size\ before\ deduplication}$.

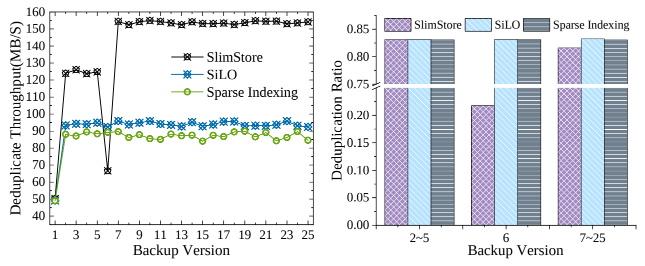
Fig 5 shows the effect of history-aware skip chunking. We observe that skip chunking significantly improves deduplicate throughput in Fig 5(a). To be more precise, Rabin-based CDC has a 2× performance improvement after adopting skip chunking, and the throughput of FastCDC also increased by 1.5 times. Base on the observation in Fig 2, CPU is the bottleneck when deduplicating, history-aware skip chunking can eliminate the computational overhead of byte-by-byte verification, so it saves a lot of CPU time and accelerates deduplication. The CPU time breakdown in Fig 5(d) demonstrates our analysis. When skip chunking is adopted, the CPU cost of CDC is reduced to about 2%. Fig 5(b) shows that skip chunking has no damage on the deduplication ratio, it achieves the same deduplication ratio as Rabin-based CDC and FastCDC. Fig 5(c) using files with different duplication ratio in S-DB to evaluate the effect of skip chunking. We notice that the increase in performance is related to the duplication ratio, file with a higher duplication ratio achieves a larger performance improvement because it has more consecutive duplicated chunks, so skip chunking is more likely to succeed.

Besides, in Fig 5(a) and 5(b), it can be seen that deduplication throughput increases as chunk size grows, and become sta-



(a) Throughput: vary file characteristics (b) Deduplication ratio: vary file characteristics

Fig. 6: Performance of history-aware chunk merging.



(a) Deduplicate throughput (b) Deduplication ratio

Fig. 7: Comparison of fast online deduplication of SLIMSTORE, SiLO, and Sparse indexing.

ble after 32 KB. In contrast, the deduplication ratio degrades, and the downward trend becomes sharper when chunk size is larger than 16 KB. Therefore, history-aware chunk merging is proposed to dynamically tune to an appropriate chunk size, which can achieve a compromise between deduplication speed and deduplication ratio.

We vary the file duplication ratio to verify the performance of history-aware chunk merging, and the initial chunk size is 4KB. The result is shown in Fig 6. It is obvious that history-aware chunk mering can improve the deduplicate throughput. For the file with a duplication ratio of 0.95, the improvement is more than 20%, from 125MB/s to 155MB/s, at the expense of only 0.9% deduplication ratio. But for the file with a lower duplication ratio, chunk merging has a lower benefit and a higher deduplication ratio loss. The reason for the difference is that files with a high duplication ratio will merge more superchunks, thereby causing the average chunk size is large, which is demonstrated by the red line in Fig 6(a), and the large chunks after merging accelerate the deduplication process. In addition, a high duplication ratio means that the data is less likely to be modified, so the deduplication ratio loss caused by the superchunk change is also small for the file with a high duplication ratio as shown in Fig 6(b).

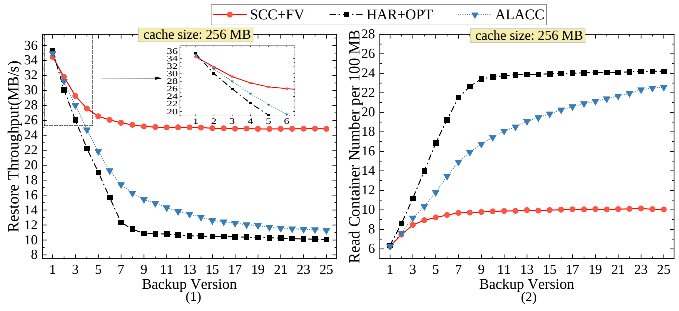
We also perform the evaluation to see the overall deduplication performance of SLIMSTORE and compare it with SiLO and Sparse Indexing. We set the default chunk size is 4KB for all three methods, and the merge threshold is repeatTimes exceeds 5 for SLIMSTORE. Fig 7(a) shows that the stateless deduplication and history-aware skip chunking inspire the throughput of SLIMSTORE, which is 1.32× than SiLO and 1.39× than Sparse Indexing before version 6, meanwhile, three methods achieve almost the same deduplication ratio. When deduplicating version 6, history-aware chunk merging is triggered, so many superchunks are generated and need to store on OSS, which causes performance degradation. But after version 6, benefit from the performance improvement brought by chunk merging, SLIMSTORE outperforms SiLO and Sparse Indexing by 1.63× and 1.72× respectively. Because large chunk size may cause the loss of deduplication ratio, so SLIMSTORE loses about 1.5% of the deduplication ratio compared to other two methods.

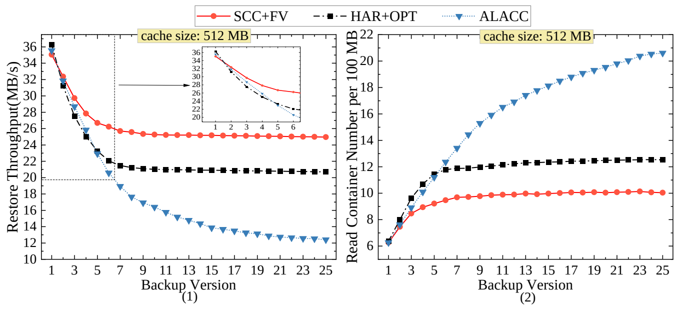
C. Restore Performance

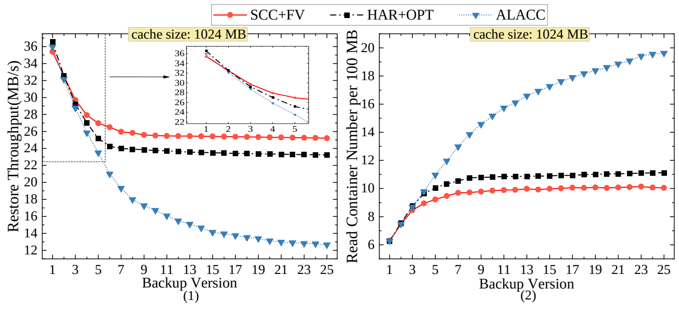
We optimize restoration with two goals of high time-efficiency and low OSS bandwidth consumption. Therefore, we use the restore throughput and read container number per 100 MB to demonstrate our optimization. We backed up 25 versions of S-DB continuously and then restored them under different cache sizes, and we disable LAW-based prefetching to evaluate the effect of full vision cache(FV) and sparse container compaction(SCC). Because before version 5, sparse container is rare, the restore performance depends on the ability of the restore cache to solve large-span containers and self-reference chunks. The partial enlarged views in Fig 8 show the impact of three restore caches. We observe that our FV cache always performs the best under different cache sizes. In contrast, because the unit of OPT cache is container, many useless chunks occupy the precious cache space, thereby causing a low hit ratio, so the read amplification of OPT cache is serious, which causes the worst performance as shown in Fig 8(a). FV cache and ALACC adopted chunk-based cache, so they perform better than OPT cache when cache is small, but due to the limited vision of look-ahead window(LAW) in ALACC, it can not solve the problem of fragments that exceeds LAW. With full restore information, FV cache can address these fragments to makes sure all containers only be read once, so FV cache outperforms ALACC.

When cache size reaches 1024 MB, the large cache can preserve more useful chunks, so the impact of fragments like large-span container and self-reference chunk is reduced. Therefore, the main performance loss comes from sparse containers. With SCC, the read container number per 100 MB is stabilizing after version 7, as shown in (2) of Fig 8(c), which avoids unlimited read amplification, thereby protecting the restore performance from declining over time. Because ALACC has no optimization on sparse containers, so the read amplification continues to increase, resulting in a lot of OSS bandwidth consumption and impair restore performance. HAR has the same effect on restore performance stabilizing, but it rewrites chunks in sparse containers in the next version, which causes the restore performance is still suffering from some sparse containers. And because of the disadvantage of OPT cache in dealing with fragments that exceed LAW, the restore throughput of HAR+OPT cache is still 10% lower than SCC+FV cache. Therefore, SCC and FV cache perform best in combating fragmentation compared with existing methods, and the restore throughput almost reaches the upper limit of the single-channel OSS read bandwidth.

Fig 8(d) shows the performance of LAW-based prefetching, which eliminates the waiting time for reading form OSS by prefetching the continaers to be accessed soon. The results


(a) Restore performance with 256 MB cache


(b) Restore performance with 512 MB cache


(c) Restore performance with 1024 MB cache

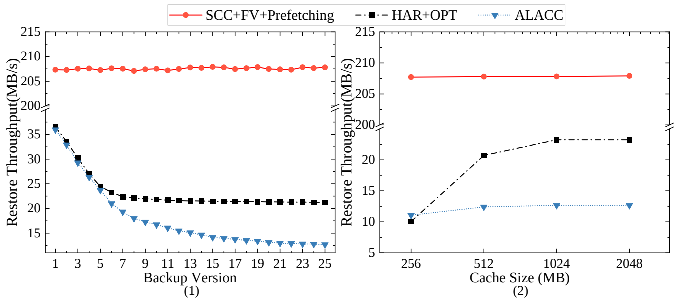

(d) Restore performance with LAW-based prefetching

Fig. 8: Comparison of restore performance.

TABLE II: **Vary prefetching thread number**

Prefetching Thread Number	0	1	2	4	6	8	10
Restore Throughput (MB/s)	36	38	75	154	207	208	208

show that excellent performance is achieved when SCC + FV enables LAW-based prefetching, which is 9.75× and 16.35× of HAR+OPT and ALACC respectively. Besides, LAW-based prefetching achieves the same restore throughput of new and old versions, thereby avoiding restore performance degrades for new versions. We further explore the relationship between prefetch speed and restore speed by varying the prefetching thread number in Table II. After the number of threads reaches 6, restore performance stabilizes, which means that all chunks can be obtained directly in memory, so the highest time-efficiency of restore job is achieved.

D. Space Cost

G-node can effectively manage space by strategies like sparse container compaction, global reverse deduplication, and version collection. Fig 9 demonstrates the effect of space management after backing up 25 versions of S-DB. We use L-dedupe to represent deduplication on L-node and G-dedupe as global reverse deduplication. In (a), when deduplication is not applied, a total of 2.44 TB data are stored. L-dedupe can eliminate duplicate data, thus resulting in a 4.8× reduction in space consumption, occupying only 516.6 GB of storage space, which proves the effectiveness of fast deduplication on L-node. G-dedupe can achieve exact deduplication, so it further reduces the occupied space by 2.4% to 504.2 GB. To measure the effect of version collection, we only preserve the last 10 versions. We can observe that after version 10, the growth of space usage has slowed down, it is because many old containers become garbage and are reclaimed. Therefore, version collection can significantly reduce the space occupation of old versions and improve space efficiency.

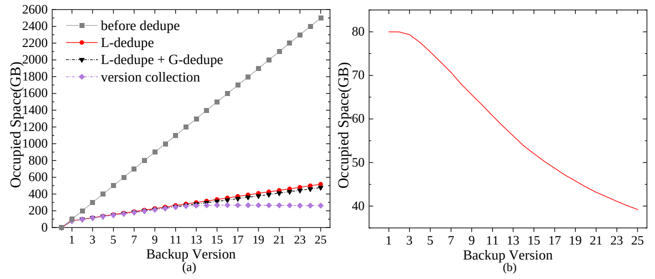


Fig. 9: Effect of space management

Fig 9(b) shows the space occupied by version 0 as time goes by, and the version collection is not enabled. It can be seen that the occupied space is gradually decreasing. Because sparse container compaction will merge some duplicate chunks into new containers to gain a better physical locality, which means that some data are moved into new versions, thus reducing the occupied space of old versions. Moreover, global reverse deduplication also eliminates duplicate chunks in old versions to reduce their data volume. Therefore, the occupied space of old versions is decreasing over time, which meets user needs that spend less money to store old backup data.

E. Comparison with Open-source Deduplication System

We compare the performance of SLIMSTORE with Restic by adopting the real backup workload R-Data. We use concurrent jobs to deduplicate and restore the files of R-Data, with each job processing one file. Because Restic recommends using large chunks with 1MB, so we increase the chunk size of SLIMSTORE, which ranging from 256KB to 2MB by adopting history-aware chunk merging. In Fig 10(a), SLIMSTORE achieves a linear increased throughput as the number of concurrent backup jobs increases. We notice that one job of SLIMSTORE outperforms Restic by 25%. Moreover, when the number of concurrent backup jobs exceeds 13, multiple L-

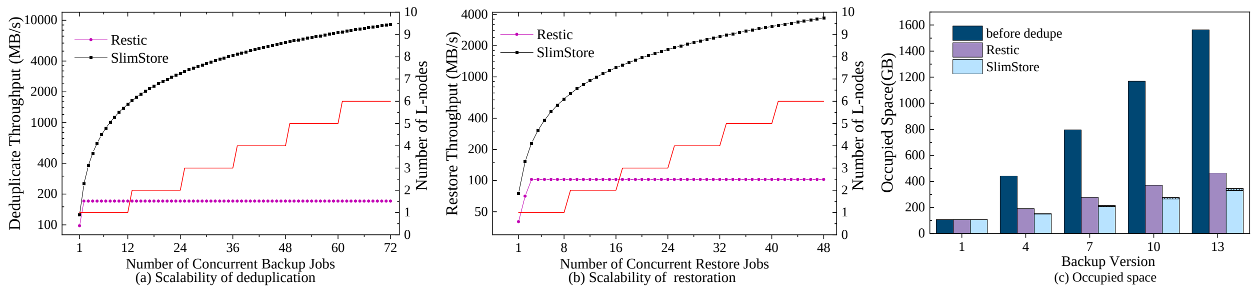


Fig. 10: Comparsion of SLIMSTORE and Restic

nodes can be allocated for parallel deduplication as the red line in Fig 10(a) shown. Therefore, SLIMSTORE achieves scalable deduplication according to the actual backup workload, and reaches 9102 MB/s when 72 deduplicate jobs are executed in parallel. However, because multiple Restic deduplication jobs need to access the same fingerprint index to identify duplicates, so Restic cannot carry out multiple backup jobs concurrently, which limits its deduplication throughput to 170 MB/s. Restore performance shows the same trend. We set the prefetching thread number as two for SLIMSTORE. Due to network bandwidth limitations, each L-node can execute up to eight restore jobs at the same time. Fig 10(b) shows SLIMSTORE achieves linear scalable restore throughput, which reaches 3676 MB/s when six L-nodes execute concurrently. As for Restic, limited by the fingerprint index access to get the data locations, it only gets a maximum restore throughput of 102 MB/s. Fig 10(c) shows the occupied space. Because SLIMSTORE can adjust the chunk size according to the actual data characteristics, ranging from 256KB to 2MB, so it achieves a higher deduplication ratio, and saves about 20% of the space than Restic. Besides, the shaded part of SLIMSTORE shows the effect of global reverse deduplication, which can further reduce the space occupation by 4.6%.

VIII. CONCLUSION

This paper presents SLIMSTORE, a cloud-based deduplication system that provides online deduplicate and restore services for large-scale multi-version backups. It perform fast deduplication and restoration for new backup versions while ensuring the effectiveness of deduplication to reduce storage costs. Several techniques are proposed to improve its efficiency. In isolation, each of these technique is fairly simple. The novelty comes from designing and combining these ideas into an effective and coherent system that meets design goals. Experimental results demonstrate that SLIMSTORE achieves very high-speed deduplication and restoration, and can effectively eliminate duplicate data to reduce the storage costs.

REFERENCES

[1] "Alibaba oss," https://www.alibabacloud.com/product/oss/.

[2] "Amazon s3," https://aws.amazon.com/s3/.

[3] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST*, 2008, pp. 269–282.

[4] W. Xia *et al.*, "Silo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *ATC*, 2011.

[5] M. Lillibridge, K. Eshghi *et al.*, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *FAST*, 2009, pp. 111–123.

[6] M. Fu, D. Feng, Y. Hua *et al.*, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *ATC*, 2014, pp. 181–192.

[7] M. Kaczmarczyk, M. Barczynski *et al.*, "Reducing impact of data fragmentation caused by in-line deduplication," in *SYSTOR*, 2012, p. 11.

[8] M. Lillibridge, K. Eshghi *et al.*, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *FAST*, 2013.

[9] *Understanding data deduplication ratios*, Storage Networking Industry Association, 2008.

[10] C. Dubnicki, L. Gryz *et al.*, "Hydrastor: A scalable secondary storage," in *FAST*, 2009, pp. 197–210.

[11] J. Wei, H. Jiang *et al.*, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *MSST*, 2010.

[12] B. K. Debnath, S. Sengupta, and J. Li, "Chunkstash: Speeding up inline storage deduplication using flash memory," in *ATC*, 2010.

[13] W. Xia, H. Jiang *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[14] A. Muthitacharoen, B. Chen *et al.*, "A low-bandwidth network file system," in *SOSP*, 2001, pp. 174–187.

[15] W. Xia, H. Jiang *et al.*, "Ddelta: A deduplication-inspired fast delta compression approach," *Perform. Evaluation*, 2014.

[16] W. Xia, Y. Zhou *et al.*, "Fastcdc: a fast and efficient content-defined chunking approach for data deduplication," in *ATC*, 2016, pp. 101–114.

[17] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *ATC*, 2011.

[18] D. Bhagwat, K. Eshghi *et al.*, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *MASCOTS*, 2009.

[19] M. Fu, D. Feng *et al.*, "Design tradeoffs for data deduplication performance in backup workloads," in *FAST*, 2015, pp. 331–344.

[20] Y. Nam, G. Lu *et al.*, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *HPCC*, 2011, pp. 581–586.

[21] Y. Nam, D. Park *et al.*, "Assuring demanded read performance of data deduplication storage with backup datasets," in *MASCOTS*, 2012.

[22] Z. Cao, S. Liu *et al.*, "Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance," in *FAST*, 2019, pp. 129–142.

[23] M. Fu, D. Feng *et al.*, "Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 855–868, 2016.

[24] Z. Cao, H. Wen *et al.*, "ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching," in *FAST*, 2018, pp. 309–324.

[25] D. N. Simha, M. Lu *et al.*, "A scalable deduplication and garbage collection engine for incremental backup," in *SYSTOR*, 2013.

[26] "Hydrastor," https://www.necam.com/hydrastor/.

[27] "Netbackup," https://www.veritas.com/protection/netbackup-appliances.

[28] "Avamar," https://www.delltechnologies.com/en-us/data-protection/data-protection-suite/avamar-data-protection-software.htm.

[29] "Restic," https://restic.net/.

[30] A. Z. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of SEQUENCES*, 1997.

[31] "Alibaba ossfs," https://github.com/aliyun/ossfs.