SLIMSTORE: Towards an Efficient Cloud-based Deduplication System for Multi-version Backups

Zihao Zhang, Huiqi Hu(✉), Zhihui Xue, Changcheng Chen, Yang Yu,
Cuiyun Fu, Xuan Zhou, Feifei Li, and Aoying Zhou

Abstract—Cloud backup is becoming a preferred way 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 deduplication methods do not suit the cloud environment in providing scalable backup service, and cannot achieve efficient deduplication and restoration because they suffer from the high latency of cloud storage. Moreover, they cannot meet the variation of user's requirement on data layout among different backup versions.

In response, we present SLIMSTORE, with a new cloud-based deduplication architecture that disassembles the system into storage and computing layer to support elastic scaling. To cater to different responsibilities, the computing layer is decomposed into two types of processing nodes. L-nodes fully utilize similarity and locality to enable fast online deduplication, and with history-aware strategies to further enhance deduplication efficiency. L-nodes also achieve the highest online restore efficiency of new backup versions with an effective cache. Meanwhile, G-node optimizes data layout offline, which guarantees the restore efficiency of new versions while reducing overall storage cost. Experimental results show that SLIMSTORE enables fast deduplication, efficient restoration, and effective space reduction.

Index Terms—deduplication system, cloud storage, database backup

1 INTRODUCTION

ENTERPRISES used to store backup data with low-cost storage such as SMR disks and tape libraries. In recent years, as data scale has increased dramatically and cloud storage has been available, more and more users have chosen to move backup data to the cloud because of its reliability and convenient disaster recovery capability. Cloud storage not only means elastic storage capacity but also flexible pricing which can significantly save the cost of early storage device investment. While the market for cloud backup services has been growing quickly, cloud providers strive to reduce the storage cost of backup service, to gain a competitive advantage.

As backup data is usually cold and not accessed frequently, customers prefer storage with low cost and large capacity but slower access speed. *Object Storage Service (OSS)*, such as Alibaba's OSS [1] and Amazon's S3 [2], seems to be an ideal choice, due to its low price and large storage capacity. Although OSS is already cheap, we are still incentivized to explore ways to further reduce storage costs. Based on our observation, customers' needs for backup service are long-term and continuous. They tend to upload the latest versions of data to the cloud regularly. For instance, database users prefer to upload the latest snapshots of a database everyday for rapid disaster recovery. Multiple consecutive backups lead to massive duplication among the backup versions. Data deduplication technology can be employed to eliminate duplication and remarkably reduce the storage cost. This paper introduces SLIMSTORE, a cloud-based deduplication system, which enables backup service on OSS can further reduce storage costs.

Data deduplication has been widely employed in local backup systems. Three indicators can be used to measure the effectiveness of a deduplication system: deduplication speed, restore speed, and deduplication ratio. It is difficult to achieve the optimal in all three indicators. Therefore, most of the existing works only focus on one of them. For instance, DDFS [3], SiLO [4] and Sparse Indexing [5] attempted to find a trade-off between deduplication speed and deduplication ratio, while HAR [6], CBR [7], and Capping [8] tried to achieve better restore performance at the price of declined deduplication ratio. In industry, as most enterprises prioritize storage cost, they usually choose to maximize the deduplication ratio [9].

With the migration of backup service to the cloud, the design goal of a deduplication system has shifted. It appears that a cloud-based deduplication system faces two main challenges. The first challenge is posed by the data access performance of cloud storage. In OSS, as the storage nodes are physically separated from the computing nodes, the data access latency increases, which may slow down deduplication and restoration. To alleviate the negative impact of high latency, we need to exploit data similarity and locality to minimize the interactions between the storage nodes and the computing nodes, thereby providing fast deduplication and restore services.

The other challenge comes from the variation of user requirements for data layout among different backup versions. If deduplication is employed, a newer backup version

- ✉ *represents the corresponding author.*
- *Zihao Zhang, Huiqi Hu, Yang Yu, Xuan Zhou and Aoying Zhou are with the School of Data Science and Engineering, East China Normal University, Shanghai 200062, China. E-mail: {zihaozhang, yuyang}@stu.ecnu.edu.cn, {hqhu, xzhou, ayzhou}@dase.ecnu.edu.cn.*
- *Zhihui Xue, Changcheng Chen, Cuiyun Fu, and Feifei Li are with Alibaba Group, Hangzhou 311100, China. E-mail: {zhihui.xzh, tianyu, cuiyun.fcy, lifeifei}@alibaba-inc.com*

is more likely to be fragmented than older versions, as its duplicated parts are redirected to the old. However, a newer version is more likely to be restored, because users usually choose to restore the latest version. Therefore, users want the newer versions to be less fragmented so that restoration can be faster. Conversely, as the data value of old versions is declining, users' expectations for restore speed decrease, thereby reducing the storage occupancy of older backup versions is more cost-efficient for the system. In order to resolve the contradiction, we need to perform data layout optimization to speed up the restoration of newer versions while reducing the space consumption of older versions. Previous work on local backup deduplication did not fully consider the above challenges.

To elastically scale and make full use of cloud resources, SLIMSTORE separates storage and computation (by storing backup data on OSS) to gain uncapped storage expansion, and use elastic computing resources to achieve scalable deduplication and restoration. SLIMSTORE further decomposes the computing layer into two types of processing nodes with different responsibilities. Namely, L-nodes are responsible for fast online deduplication and restoration, while G-node are responsible for optimizing data layout to meet the user requirements for restore speed.

As L-nodes must combat the high latency of OSS to perform deduplication and restoration at acceptable user experience, a lightweight deduplication method is designed. By making full use of the similarity and locality among versions, L-nodes only access some synopsis of history versions on the cloud to perform deduplication. At the same time, we employ two strategies named *history-aware skip chunking* and *chunking merging*, which exploit the historical information to make the deduplication more CPU-efficient. To accelerate online restoration, we design a *full-vision cache* that accurately preserves useful chunks to reduce the impact of fragmentation. Besides, to hide the data access latency during restoration, L-nodes prefetch data into memory through an approach called *look-ahead perfetching*. By embedding the prefetching strategy into the effective cache, SLIMSTORE can achieve the highest restore efficiency.

G-node plays an important role in optimizing data layout. It performs optimization offline by executing *sparse container compaction* and *global reverse deduplication*. Both techniques can adjust the data layout between the old and new versions, to reduce the overall storage cost, and at the same time improve the data locality and thus the restore performance of new versions.

Our contributions are summarized as follows:

- We propose a scalable cloud-based deduplication system that is suitable for elastic cloud environments. SLIMSTORE separates the computing and storage layers, so that can scale separately.
- We propose holistic designs to optimize the efficiency of online deduplication and restoration by minimizing the impact of the high latency of OSS. Moreover, two optimizations are explored to accelerate deduplication by exploiting historical information.
- We resolve the mismatch between data fragmentation and user requirements for restore speed among different versions by optimizing data layout offline. The optimization can improve the restore speed of newer versions and ensure the space efficiency of the system.
- We implement SLIMSTORE and deploy it on the cloud. We conduct extensive experiments to demonstrate that SLIMSTORE can promote the restore speed of backup data. It also outperforms the best alternative method by 1.72× in deduplication efficiency. Besides, SLIMSTORE can achieve scalable deduplication and restoration.

The paper is organized as follows: Section 2 reviews related works. Section 3 introduces the system architecture. In Section 4, 5, and Section 6, we present the deduplication, restoration and space management methods of SLIMSTORE in detail. The experimental results are presented in Section 7. Finally, we conclude our work in Section 8.

2 RELEATED WORK

In general, many works [3]–[5], [10]–[12] adopt chunk-level (e.g., 4KB) deduplication instead of file-level deduplication because it can identify and eliminates duplicates in finer granularity. Xia et.al. [13] divides the chunk-level deduplication workflow into five stages, namely, chunking, fingerprinting, indexing, further compression, and storage management. The storage management includes data restore, garbage collection, etc. Because further compression [14]–[16] is not a key stage of deduplication, this section will mainly introduce the other four stages. Firstly, the chunking stage divides backup data into small chunks so that it can identify more duplicates. Fixed-size chunking is the simplest chunking method, while its low deduplication ratio is low due to the boundary-shift problem [17]. 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 [17] is the most classical CDC method and is widely adopted, but the computation of Rabin hash is time-consuming, so Gear [18] and FastCDC [19] are proposed that 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 to generate a fingerprint by a *cryptographically secure hash*(e.g., SHA-1, SHA-256), which is the unique identifier of each chunk. Therefore, two chunks can be identified as duplicates if they have the same fingerprint.

The indexing stage will build a fingerprint index to help identify duplicates, which is a key component of deduplication systems. In a large-scale deduplication system, the fingerprint index is considered as the bottleneck because its size is overgrowing with the explosive growth of backup data so that it cannot reside in memory. Many works attempt to avoid the bottleneck of fingerprint-lookup on disk. DDFS [3], ChunkStash [12], and Sampled Index [20] 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 [21] are index methods that use logical locality for deduplication. Sparse

Indexing improves memory utilization by sampling representative fingerprints in memory and using champions that are more similar to the current data stream to identify duplicates. SiLO and Extreme Binning adopt similarity detection to reduce the RAM overhead for indexing, by exploiting 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 [22] explores the tradeoffs among deduplication ratio, RAM overhead, and restore performance, which provides some good insights in designing a fingerprint index.

Restoration is another important issue. An inevitable fact is that the restore speed will be hurt after deduplication because the backup data is physically scattered to many locations, which is known as fragmentation. Since fragmentation is exacerbating with the increase of backup versions, the restore performance severely declines over time. Many solutions tried to rewrite the fragmented duplicate chunks into new containers to improve the locality of data [6]–[8], [23]–[26]. These methods differ in fragmentation identification strategies and the timing of rewriting.Capping [8], CBR [7], and LBW [25] 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], [26] 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, then rewrites duplicate chunks in sparse containers when backing up the next version. Another research direction to optimize restoration is to design an efficient caching policy. Some works like Kaczmarczyk et al. [7], Nam et al. [23] and Optimal Restore Cache [6] adopt container-based cache. Optimal Restore Cache 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 a higher cache hit ratio [8], [27]. Lillibridge et al. [8] propose a forward assembly area (FAA) to assemble the restored data, it directly copies chunks from container-read buffer to their position in FAA without caching anything. ALACC [27] combines FAA and chunk-based cache to reduce cache management overhead and achieve better restore performance.

Garbage collection (GC) [6], [11], [20], [28], [29] can effectively manage space in deduplication-based backup storage. The previous 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], [28], but it is complicated and suffers from low relability [6], [20]. Mark-and-sweep consists of two stages. Mark stage traverses all chunks and marks the referenced chunks. In the sweep stage, the unreferenced chunks will be reclaimed.

Data deduplication is also adopted in some commercial software(e.g., HYDRAstor [30], NetBackup [31], Avamar [32]) to provide enterprise-level deduplication solutions. Among open-source deduplication projects, Restic [33] is the most popular one with more than 13K stars on GitHub, which is designed for deduplicating on top of the local file system, it cannot provide deduplication service for the cloud. In this paper, we focus on building a cloud-based deduplication system, which separates storage and computation to make full use of elastic recourses of the 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 by using two history-aware strategies. To promote the restore performance, SLIMSTORE employs a full vision restore cache and embedded with look-ahead prefetching strategy, which can realize higher restore efficiency than existing restore caches [6], [20], [27]. SLIMSTORE also performs sparse container compaction and global reverse deduplication offline to adjust the data layout for better restore performance of new versions.

3 System Overview

3.1 Design considerations

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 consecutive 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 (see details in Section 3.2). By exploiting the recipe content, the information of the historical version can be used to identify duplicates and accelerate deduplication. When restoring a backup, the system can also restore the original files through the file recipes.

Separation between storage and computation. Many conventional deduplication systems store data and perform deduplicate or restore jobs on the same machine [3], [20] , 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. SLIMSTORE can obtain storage of any capacity by storing data on cloud storage, and flexibly allocates computing resources to handle dynamic backup or restore workloads. Therefore, SLIMSTORE is more cost-efficient because of its elastic expansion capabilities.

Decomposition of computation. SLIMSTORE aims to provide scalable and fast online deduplication and restore services for users, meanwhile, the system also needs some computing resources to optimize the storage to cater to user requirements for data layout. Therefore, SLIMSTORE decomposes the computing layer into two types of processing nodes, named L-node and G-node respectively. L-nodes are working on eliminating the impact of OSS's high latency to provide fast online deduplication and restore services, and by exploiting historical information, L-nodes significantly improve the CPU efficiency of deduplication, which further enhance the system performance. While G-node offline executing storage management operations to adjust data layout for efficient restoration and effective space reduction.

3.2 System Components

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

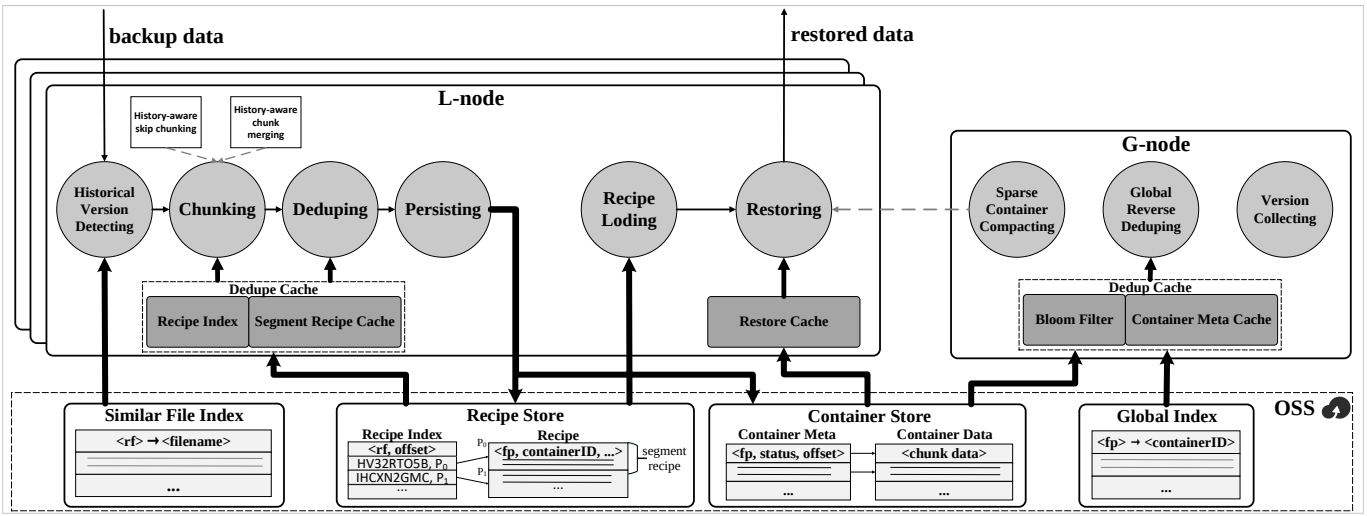


Fig. 1: System architecture of SLIMSTORE.

3.2.1 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. The container-based storage gives rise to the *physical locality*. Since a container is the 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. Besides chunk data, 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 5) in the container.

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 4). 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 consecutive 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.

Similar File Index. Similar index stores the representative fingerprints of each file, which is used to find similar files. Accord to Broder's theorem [34], the similarity of the full set is highly dependent on the similarity of two randomly sampled subsets. As a file can be considered as a set of fingerprints, 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 stored container. 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.

3.2.2 Computing Layer

The computing layer is composed of Alibaba cloud elastic compute services (ECS). The ECSs act as two types of processing nodes named L-node and G-node with different design purposes.

L-node. L-node is responsible for fast online deduplication and restoration. When the backup command reaches the L-node, it starts to receive the input file stream and deduplicate it. Traditional deduplication systems suffer from performance loss due to the frequent access to the fingerprint index, which is extremely onerous in the cloud environment since the index is placed on cloud storage. 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 accesses. Besides, as the historical information can also facilitate deduplication, two optimizations named *history-aware skip chunking* and *chunk merging* are further proposed to improve its efficiency (see details in Section 4).

As for restoration, L-node first loads the recipe of the target file, and then reads chunks from containers and splices them together based on the sequence of chunk that records in the recipe. Because the online restoration needs to combat

the read amplification caused by fragmentations, and also need to resolve the high latency because all the restored data is read from OSS. Therefore, we design an efficient restore cache with full restore information to maximize the hit ratio, and with the look-ahead prefetching strategy to completely hide the high latency of OSS in the backend, the restore cache in SLIMSTORE can achieve the highest time efficiency for online restoration(see details in Section 5.1).

Noting that L-node is stateless, all the information required in deduplication and restoration is loaded during the job execution, and both the fetching of the recipe index and segment recipes are lightweight. Thus it can be quickly deployed and execute deduplicate and restore jobs, which allows the system to dynamically allocate multiple L-nodes to cater to different users' workloads.

G-node. G-node is responsible for managing the storage space offline. The optimization of storage space is under the premise of adjusting data layout to cater to user requirements. Therefore, all the operations on G-node are working on this purpose.

G-node performs sparse container compaction to optimize data layout to be conducive for restoration of new versions. Considering the fact that duplicate chunks of backup are redirected to the data of old versions, the data distribution is physically scattered after deduplication, especially for new backup versions, which results in the restore performance degrades over time. Therefore, G-node performs sparse container compaction to gather the chunks that are still visible to the new version into the new containers. This operation can transfer part of data in old versions to new versions to promote the locality of the latter, which improves the restore performance of new versions(Section 5.2).

Global reverse deduplication is proposed to maximize the deduplication ratio to save storage space. Because fast deduplication on L-node may ignore some duplicates, G-node adopts global reverse deduplication to augment the deduplication ratio by further filtering containers generated by L-node, find and eliminate duplicate chunks to achieve exact deduplication. By removing the duplicates in old versions instead of the new version, global reverse deduplication can protect the data layout of new versions and reduce the overall storage cost (Section 6.1). Besides, G-node collects the deleted old versions to further reduce the space occupied by the old version (Section 6.2).

4 DEDUPLICATION

We first introduce the process of online deduplication, with a detailed explanation of how to use the similarity and locality to realize lightweight deduplication. Then two techniques that exploit historical information are further proposed to accelerate deduplication.

4.1 Deduplication Workflow

To minimize the performance penalty caused by high-latency OSS access, L-nodes try to make full use of the similarity between consecutive backup versions to perform lightweight deduplication. By only access the recipe of the similar segment that is searched in historical versions, and use logical locality to quickly filter duplicates, L-nodes can significantly reduce the OSS accesses to promote deduplication efficiency. 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 similar file by querying the similar file index. We use the straightforward random sampling method adopted in many deduplication works [5], [22], 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 [21]. 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 segments 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 a similar segment. If a chunk with the same fingerprint exists, it prefetches the corresponding segment recipe and adds it to the dedupe 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.

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.

4.2 History-aware Skip Chunking

Despite lightweight deduplication can avoid the impact of OSS's high latency, deduplication performance still suffers from the time-consuming chunking algorithm, we intend to accelerate it by exploiting historical information and data locality. Therefore, history-aware skip chunking is proposed that can significantly improve the CPU efficiency of chunking, which results in a huge performance improvement and reduce the overall executing time of deduplication jobs.

Content-defined chunking (CDC) is the dominating chunking method for deduplication due to its high dedu-

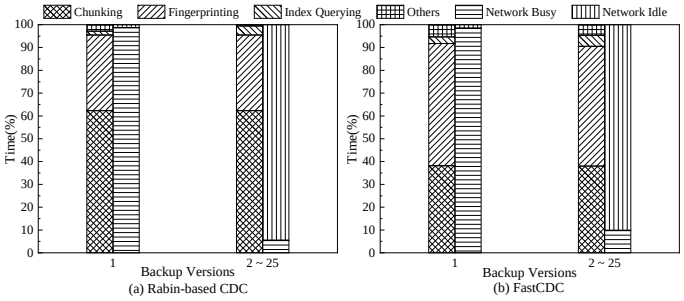


Fig. 2: CPU and network time breakdown of deduplication.

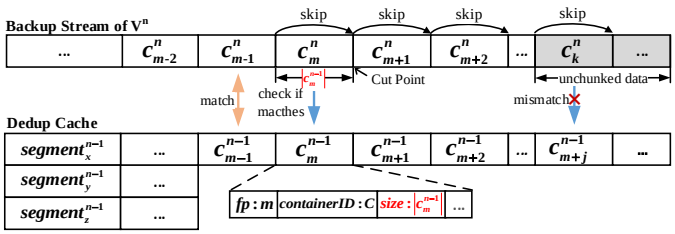


Fig. 3: An example of history-aware skip chunking.

plication 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 [17] due to the complexity of Rabin hash. Some other algorithms such as FastCDC [19] 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 duplicates are removed, resulting in less data upload, which makes the replacement of network to CPU becoming the new performance bottleneck. According to the breakdown of CPU time, chunking and fingerprinting 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 dedupe cache, and skip to the promising cut point based on the size. If skip chunking succeeds (i.e., the promising cut point meets the cut condition) and 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 backing up, where chunk c_{m-1}^n matches the chunk in $segment_x^{n-1}$ of V^{n-1} and identified as a duplicate. At this time, the size of the next chunk $|c_m^{n-1}|$ can be obtained through the information stored in $segment_x^{n-1}$, then the current version directly skip $|c_m^{n-1}|$ 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 skip chunking is successful, the fingerprint of the new chunk c_m^n can be directly compared with chunk c_m^{n-1} of the V^{n-1} to verify whether it is duplicated, thereby avoiding searches in dedupe cache to determine duplicates, the deduplication speed is further accelerated. Experimental results in Section 7.2 show that history-aware skip chunking can reduce the CPU consumption of CDC to 2%, which significantly improved deduplication performance.

4.3 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 more chunks are generated, which increases the overhead of chunking and index querying, thereby declining the deduplication speed. As for 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, a data stream with a high duplication ratio contains many consecutive duplicate data, so a large chunk size can speed up deduplication. But for data with a low duplication ratio, data changes may occur many times, it is suitable to use a small chunk size to find more duplicates. Therefore, we intend to propose a method that can tune the chunk size according to the characteristics of data, which improves the deduplication efficiency without losing the deduplication ratio.

Because 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 deduplication of the first version. 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

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}                    */
1  Function SuperChunking()
2      if c^{n-1} is the first chunk of superchunk sc^{n-1} then
3          |sc^{n-1}| ← size of sc^{n-1};
4          sc^n ← new SuperChunk();
5          sc^n.data ← file[p_0, p_0 + |sc^{n-1}|];
6          sc^n.fp ← SHA-1(sc^n.data);
7          if sc^n.fp == sc^{n-1}.fp then
8              sc^n.isDuplicate ← true;
9          else
10             c^n.isDuplicate ← true;
11             start CDC from p_1;
```

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 duplicated, the first chunk they contain must also be duplicated, 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.

5 RESTORE

Restore is the reverse process of deduplication. To restore the backup file back, the restore algorithm scans the file recipe, and reads the required chunks from OSS, then splices each chunk in sequence. As mentioned in Section 3.2, restore performance suffers from the read amplification caused by fragmentation, and read amplification further increases the waiting time for reading data from OSS due to the high latency. Therefore, we focus on optimizing restore performance in terms of reducing read amplification and hiding high latency of OSS.

5.1 Restore Cache

The restore cache of SLIMSTORE is designed for the above purpose. It consists of two parts: the full vision cache (FVC) and the look-ahead prefetching (LAP) strategy. FVC is the cache with full restore information. The full vision is able to accurately identify useful chunks, which means that it can avoid a large number of invalid reads caused by fragmentation to maximize the hit ratio. While LAP strategy can hide the OSS latency in the backend to realize the highest time efficiency. By embedding LAP into FVC, SLIMSTORE can maximize the restore performance.

Full vision cache. Due to the fragmentation issues, the conventional cache algorithm like LRU has poor performance. For example, considering an LRU container cache which 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 fragmentation 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 significantly increases the I/O waiting time. Existing works [6], [27] use a look-ahead window(LAW) to preserve these two kinds of fragmentation 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 fragmentation 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 fragmentation that outside LAW) from being evicted, and make sure all containers only be read once, which completely avoiding repeated reading.

Fig 4 shows our restore cache. To be noticed, in order to avoid the useless data in a container occupy cache space, we cache data in fine granularity. FVC directly caches chunks instead of containers. We established a counting bloom filter

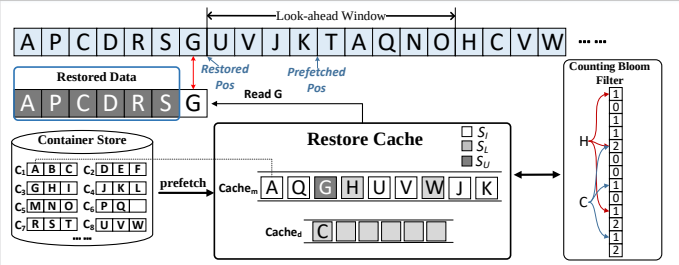


Fig. 4: An example of full vision caching policy.

(CBF) [35] for each file to obtain the full restore information. The CBF record the chunks contained in the restoring file, which is efficient to test whether a chunk is useful for restoration. CBF can also count the referenced times of each chunk, 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. Benefit from the combination of CBF and LAW, FVC can obtain the full restore vision, while also aware of the most recent access patterns. According to the existence of chunks in CBF and LAW, there are three statuses for chunks: chunks that 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, which 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 chunks (with the status of S_I or S_L) are placed in the cache. When replacement occurs, the chunk with a status of S_U is swapped out. In extreme cases, the 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.

Look-ahead prefetching. The restore job needs to read all data from OSS, but the high access latency of OSS causes very low restore efficiency. Although restore cache can reduce the frequency of OSS access by caching chunks for future access, 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 the efficiency. The best way is to avoid 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. Benefit from the LAW in FVC, we can prefetch containers in the backend based on the access sequence in LAW, so that chunks are loaded into memory before restoring them. For example, after restoring chunk G in Fig 4, the containers to be accessed are C_8 and C_4 to restore chunk U, V, J, and K. By adopting look-ahead 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 the cache without waiting for OSS reading, which 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 Section7.3) show look ahead prefetching is extremely helpful. In our implementation, when the 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, it can achieve the highest time efficiency.

5.2 Sparse Container Compaction

We observed that three kinds of fragmentation may cause read amplification. Both *large-span container* and *self-reference chunk* can be handled by the full vision cache as described in Section 5.1. There is another kind of fragmentation 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, *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* seriously impairs the restore performance.

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 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 occupancy of old versions degrades over time.

6 SPACE MANAGEMENT ON G-NODE

G-node adjusts the data layout on the backend to improve the restore performance of new versions, and make the storage more space-efficient at the same time. The SCC technique mention in Section 5.2 caters to the goal of restore performance improvement, G-node further provides global

reverse deduplication and garbage collection techniques, which can reclaim the occupied space of old versions to reduce the overall storage costs of the system.

6.1 Global Reverse Deduplication

Because fast deduplication on L-node can not achieve exact deduplication, some duplicates 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 principle in mind, SLIMSTORE needs fast restoration for new versions and low storage occupancy 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 3.2 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 chunk stored in a container of the old version. If so, reverse deduplication starts to delete the duplicate chunk in the old container, and update the location of the chunk in the 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, global deduplication only marks the duplicate chunk as deleted in the meta of the 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 deduplication 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 queries 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.

6.2 Garbage 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 sparse containers. Because sparse containers identified by version *N* will not appear in subsequent versions, these containers are converted to the garbage and associated with version *N*. To collect the garbage containers, SLIMSTORE adopts the mark-and-sweep method. 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.

In fact, the most common scenario of version deletion is to delete all the old versions before a time point, and only the newer versions are preserved. This can be achieved by simply deleting all the garbage containers corresponding to the deleted versions. But when deleting a specific version that is not the oldest, reverse deduplication on G-node may cause a garbage container to be referenced by an older version, making it impossible to determine whether the container can be collected. Therefore, during reverse deduplication, a redirection relationship is maintained for each new container, indicating whether any chunks in the old container are redirected to it. Only when all the containers redirected to the garbage container have been deleted, the container can be safely collected.

TABLE 1: **The characteristics of dataset.**

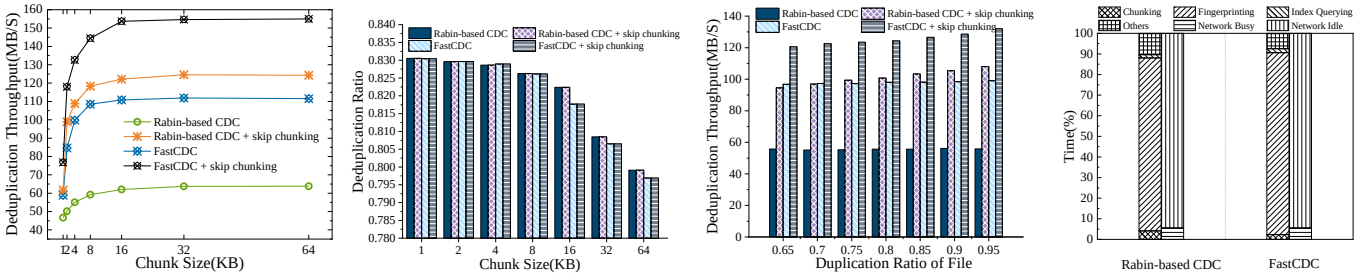
Dataset name	S-DB	R-Data
Total size (TB)	2.44	1.53
# of versions	25	13
# of files	500	7440
Average duplication ratio	0.84	0.92
Self-reference	20%	0.1%

7 EVALUATION

7.1 The Experimental Setup

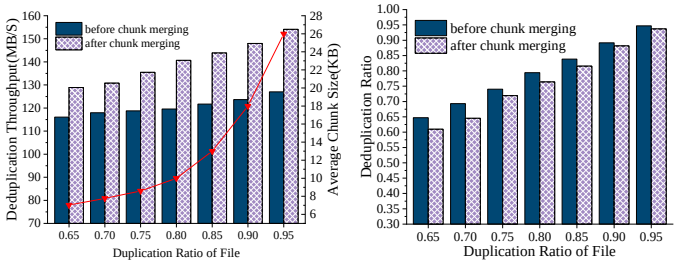
We deployed SLIMSTORE on a cluster of seven Alibaba Cloud elastic compute services (ECS), each one is equipped with a 2.50GHz Intel Xeon processor with 16 cores and 64GB memory. We use six ECSs as L-nodes and one ECS as G-node. And the cloud storage we adopt is Alibaba's OSS [1].

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 [27] to demonstrate the effectiveness of our optimization on the restore process. HAR is the rewriting method that can most accurately identify sparse containers, and OPT cache is a



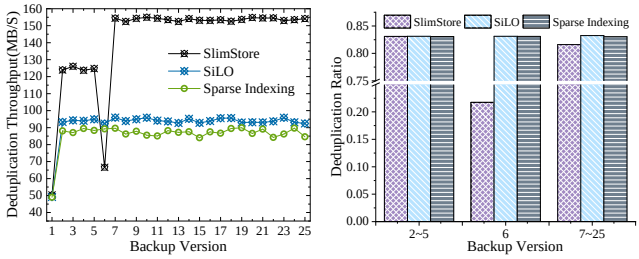
(a) Throughput: vary chunk size (b) Deduplication ratio: vary chunk size (c) Throughput: vary file characteristics (d) CPU and network time breakdown of deduplication

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



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

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



(a) Deduplication throughput (b) Deduplication ratio

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

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 [33], 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) [36], we replace Restic's storage with OSS.

We use two datasets for evaluation as shown in Table 1. 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. The average deduplication ratio between versions of R-Data is about 0.92.

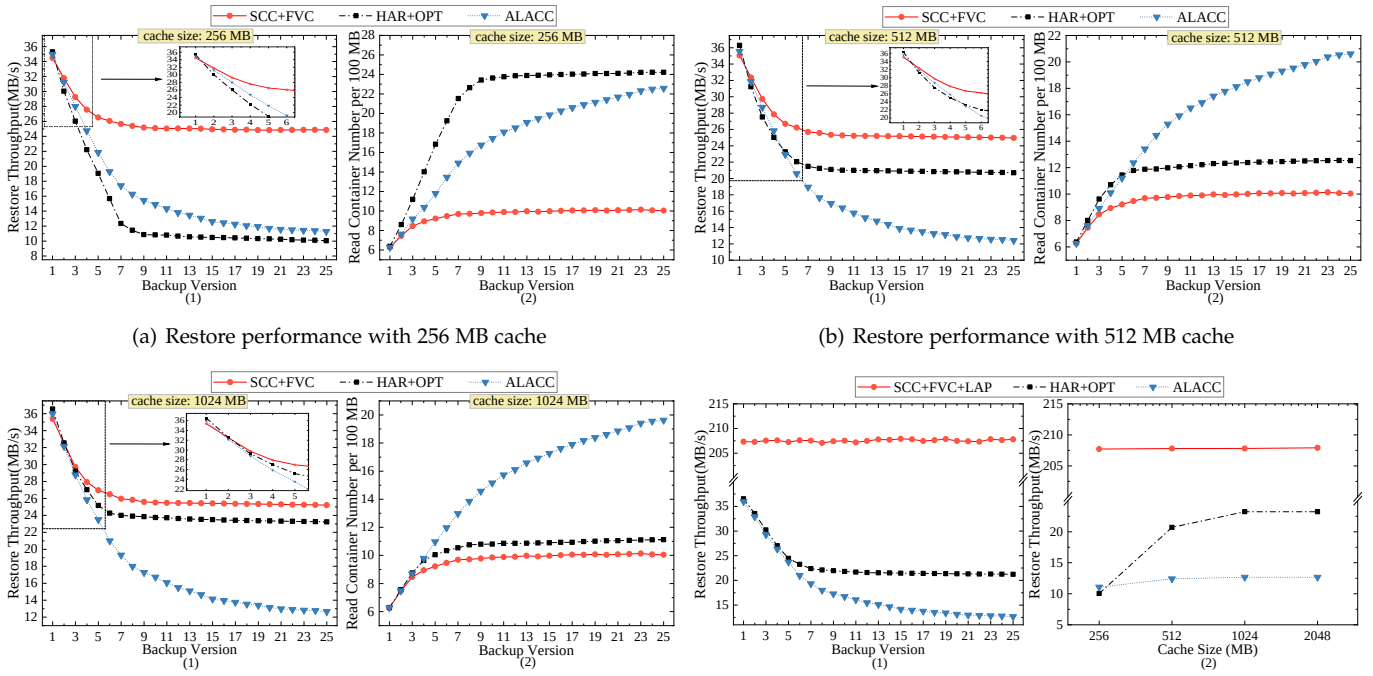
7.2 Deduplication Performance

We evaluate the deduplication performance on L-node with S-DB to demonstrate the effect of fast online deduplication. Deduplication 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 deduplication 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. As history-aware skip chunking can eliminate the computational overhead of byte-by-byte verification, it saves a lot of CPU time and accelerates deduplication. The CPU time breakdown in Fig 5(d) proves 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. In Fig 5(c), files with different duplication ratios in S-DB are used to evaluate the effect of skip chunking. We notice that the performance improvement is related to the duplication ratio. Files with a higher duplication ratio achieve larger performance improvement because they have more consecutive duplicated chunks, which leads to a greater chance of success in skip chunking.

Besides, in Fig 5(a) and 5(b), it can be seen that deduplication throughput increases as chunk size grows, and become stable after 32 KB. In contrast, the deduplication ratio is gradually decreasing, and the downward trend becomes sharper when the chunk size is larger than 16 KB. To achieve a compromise between deduplication speed and deduplication ratio, history-aware chunk merging is further proposed to dynamically tune to an appropriate chunk size.

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 deduplication throughput. For the file with a duplication



(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 look-ahead prefetching and 1024 MB cache

Fig. 8: Comparison of restore performance.

ratio of 0.95, the improvement is more than 20%, from 125MB/s to 155MB/s, at the expense of only a 0.9% drop in 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 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 as shown in Fig 6(b).

We further evaluate the overall deduplication performance of SLIMSTORE and compare it with SiLO and Sparse Indexing. We set the default chunk size as 4KB for all three methods, and the merge threshold is duplicateTimes exceeds 5 for SLIMSTORE. Fig 7(a) shows that the lightweight 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, many superchunks are generated and to be stored 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, about 1.63× and 1.72× faster than them respectively. Because large chunk size may cause the loss of deduplication ratio, SLIMSTORE loses about 1.5% of the deduplication ratio compared to the other two methods. In summary, fast deduplication of SLIMSTORE can significantly improve the deduplication efficiency.

7.3 Restore Performance

The restore optimization aims to improve restore time efficiency and reduce read amplification. Therefore, we measure the restore throughput and read container number per 100 MB to demonstrate the effect of optimization. We backed up 25 versions of S-DB continuously and then restored them under different cache sizes, and we disabled look-ahead prefetching(LAP) to evaluate the effect of full vision cache(FVC) 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 partially enlarged views in Fig 8 show the impact of three restore caches. We observe that FVC always performs the best under different cache sizes. In contrast, because the unit of OPT cache is the container, many useless chunks occupy the precious cache space, thereby causing a low hit ratio. As the read amplification of OPT cache is serious, it has the worst performance as shown in Fig 8(a). Since FVC and ALACC adopt chunk-based cache, they perform better than OPT cache when the cache is small. But due to the limited vision of look-ahead window(LAW), ALACC can not solve the problem of fragmentation that exceeds LAW. FVC with full restore information can address these two kinds of fragmentation well, ensuring that all containers only be read once, so FVC outperforms ALACC.

When cache size reaches 1024 MB, because large cache can preserve more useful chunks, the impact of fragmentation like large-span container and self-reference chunk is reduced, and the main performance loss comes from sparse containers. Therefore, Fig 8(c) shows the effect of SCC in solving sparse containers. With the help of SCC, the read container number per 100 MB is stabilizing af-

TABLE 2: **Vary prefetching thread number**

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

ter version 7, which avoids unlimited read amplification, thereby protecting the restore performance from declining over time. Because ALACC has no optimization on sparse containers, the read amplification continues to increase, resulting in a worse 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 fragmentation that exceeds LAW, the restore throughput of HAR+OPT cache is still 10% lower than SCC+FVC. Therefore, SCC and FVC perform best in combating fragmentation compared with existing methods, the restore throughput almost reaches the upper limit of the single-channel OSS read bandwidth.

Since waiting for reading container from storage is time-consuming, and accessing containers on OSS is extremely onerous, SLIMSTORE prefetches containers that will be accessed soon into memory based on the access pattern in LAW, thus hiding the high latency in the backend and improving the restore efficiency. Fig 8(d) shows the effect of LAP. The results show that excellent performance is achieved when SCC + FVC enables LAP, which is 9.75× and 16.35× of HAR+OPT and ALACC respectively. Besides, LAP achieves the same restore throughput of new and old versions, which avoids restore performance degrades for new versions. We further explore the relationship between prefetch thread number and restore speed in Table 2. After the number reaches 6, restore performance stabilizes, which means that all chunks can be directly obtained in memory, achieving the highest time efficiency of the restore job.

7.4 Space Cost

G-node can effectively manage space by strategies like sparse container compaction, global reverse deduplication, and garbage collection. Fig 9 demonstrates the effect of them 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 most 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. As G-dedupe can achieve exact deduplication, it further reduces the occupied space by 2.4% to 504.2 GB. To measure the effect of garbage collection, we only preserve the last 10 versions. We can observe that after version 10, the growth of space usage slows down, it is because many old containers become garbage and are reclaimed. After version 16, the capacity of the reclaimed garbage containers is larger than newly generated containers, resulting in the total occupied space slowly decrease. Therefore, garbage collection can significantly reduce the space occupation of old versions and improve space efficiency.

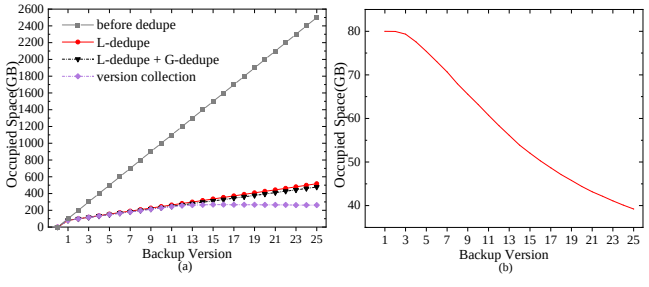


Fig. 9: Effect of space management

Figure 9(b) shows the space occupied by version 0 over time. The garbage collection is disabled. It can be seen that the occupied space is gradually decreasing. Because sparse container compaction will merge some data into new containers to gain a better physical locality, and 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.

7.5 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, we increase the chunk size of SLIMSTORE, which ranges from 512KB 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. When the number of concurrent backup jobs exceeds 13, multiple L-nodes are 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 deduplication jobs are executed in parallel. However, because multiple Restic deduplication jobs need to access the same fingerprint index to identify duplicates, 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 512KB to 2MB, 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%.

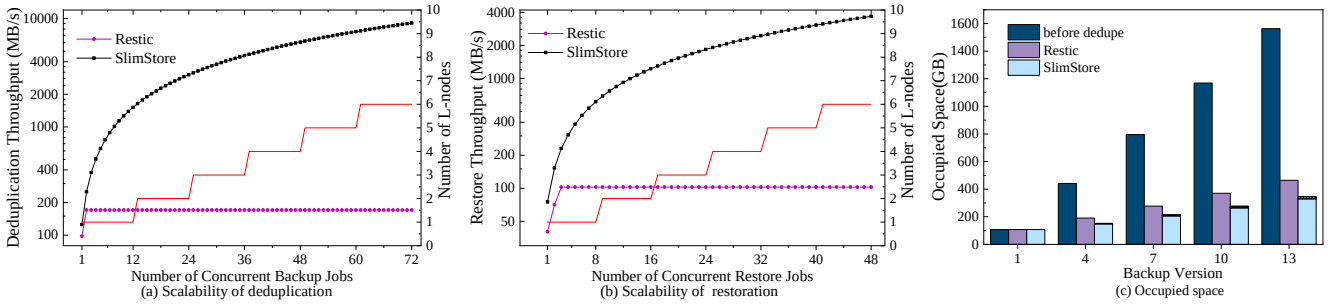


Fig. 10: Comparsion of SLIMSTORE and Restic

8 CONCLUSION

This paper presents SLIMSTORE, a cloud-based deduplication system that provides online deduplication and restore services for large-scale multi-version backups. It performs 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. We conclude the novelty comes from the design of a new cloud deduplication architecture, and the combination of the architecture and optimization techniques 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.

ACKNOWLEDGMENTS

This work was supported by Alibaba Innovative Research (AIR) Program of Alibaba Group, and the National Science Foundation of China under grant number 61772202.

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 *USENIX FAST*, 2008, pp. 269–282.
[4] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Silo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *USENIX ATC*, 2011.
[5] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *USENIX FAST*, 2009, pp. 111–123.
[6] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *USENIX ATC*, 2014, pp. 181–192.
[7] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *ACM SYSTOR*, 2012, p. 11.
[8] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *USENIX FAST*, 2013, pp. 183–198.
[9] *Understanding data deduplication ratios*, Storage Networking Industry Association, 2008.
[10] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage," in *USENIX FAST*, 2009, pp. 197–210.
[11] J. Wei, H. Jiang, K. Zhou, and D. Feng, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *IEEE MSST*, 2010, pp. 1–14.
[12] B. K. Debnath, S. Sengupta, and J. Li, "Chunkstash: Speeding up inline storage deduplication using flash memory," in *USENIX ATC*, 2010.
[13] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.
[14] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang, "Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression," in *USENIX FAST*, 2019, pp. 121–128.
[15] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *ACM Trans. Storage*, vol. 8, no. 4, pp. 13:1–13:26, 2012.
[16] W. Xia, H. Jiang, D. Feng, and L. Tian, "Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets," in *Data Compression Conference*, 2014, pp. 203–212.
[17] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *ACM SOSP*, 2001, pp. 174–187.
[18] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Perform. Evaluation*, vol. 79, pp. 258–272, 2014.
[19] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "Fastcdc: a fast and efficient content-defined chunking approach for data deduplication," in *USENIX ATC*, 2016, pp. 101–114.
[20] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *USENIX ATC*, 2011.
[21] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *IEEE/ACM MASCOTS*, 2009, pp. 1–9.
[22] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication performance in backup workloads," in *USENIX FAST*, 2015, pp. 331–344.
[23] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. C. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *IEEE HPCC*, 2011, pp. 581–586.
[24] Y. Nam, D. Park, and D. H. C. Du, "Assuring demanded read performance of data deduplication storage with backup datasets," in *IEEE MASCOTS*, 2012, pp. 201–208.
[25] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. C. Du, "Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance," in *USENIX FAST*, 2019, pp. 129–142.
[26] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, J. Liu, W. Xia, F. Huang, and Q. Liu, "Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge," *IEEE Trans. Parallel Distributed Syst.*, vol. 27, no. 3, pp. 855–868, 2016.
[27] Z. Cao, H. Wen, F. Wu, and D. H. C. Du, "ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching," in *USENIX FAST*, 2018, pp. 309–324.
[28] D. N. Simha, M. Lu, and T. Chiueh, "A scalable deduplication and garbage collection engine for incremental backup," in *ACM SYSTOR*, 2013, pp. 16:1–16:12.
[29] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. C.

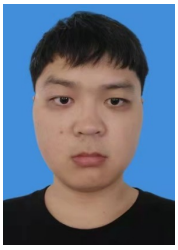
Botelho, "The logic of physical garbage collection in deduplicating storage," in *USENIX FAST*, 2017, pp. 29–44.
[30] "Hydrastor," https://www.necam.com/hydrastor/.
[31] "Netbackup," https://www.veritas.com/protection/netbackup-appliances.
[32] "Avamar," https://www.delltechnologies.com/en-us/data-protection/data-protection-suite/avamar-data-protection-software.htm.
[33] "Restic," https://restic.net/.
[34] A. Z. Broder, "On the resemblance and containment of documents," in *IEEE SEQUENCES*, 1997, pp. 21–29.
[35] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.
[36] "Alibaba ossfs," https://github.com/aliyun/ossfs.

Zihao Zhang received the bachelor's degree in software engineering from East China Normal University in 2018. Currently, he is working toward the Ph.D. degree at the School of Data Science and Engineering, East China Normal University. His research interests are in distributed database systems, including consensus protocols, database backup, and transaction processing.

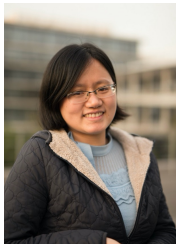
Huiqi Hu is currently an associate professor in the School of Data Science and Engineering, East China Normal University. He received his Ph.D. Degree from Tsinghua University. His research interests mainly include database system theory and implementation, scalable distributed storage system.

Zhihui Xue graduated from the Institute of High Performance Computing of Tsinghua University in 2014 with a master's degree, and is currently engaged in database backup, backup deduplication, and backup data applications in the Alibaba Cloud Database Business Unit.

Changcheng Chen is currently a senior staff engineer in Alibaba Cloud Database team. He received his bachelor's degree in computer science and technology from Quanzhou Normal University. His research interests mainly include database system and large-scale database management.



Yang Yu is currently a master's student at the School of Data Science and Engineering, East China Normal University. He received a bachelor's degree at the School of Material Science and Engineering, Central South University. His research interests include distributed database systems and persistent memory.



Cuiyun Fu received master's degree at the School of Computer Science and Engineering in Sun Yat-Sen University. She used to work in Oracle Berkeley DB team and now in the database team of Alibaba Cloud for distributed databases in Alibaba 11-11 trading systems, Alibaba real-time data replication, Alibaba database backup and disaster recovery.

Xuan Zhou is a professor and a vice-dean at the School of Data Science and Engineering, East China Normal University (ECNU). He obtained his B.Sc from Fudan University (China) and his Ph.D from the National University of Singapore, both in Computer Science. Since his graduation in 2005, he had worked as a scientist at the L3S Research Centre (Germany) and the CSIRO ICT Centre (Australia) until the end of 2010. Before he joined ECNU in 2017, he spent 6 years in Renmin University of China, as an associate professor. Xuan's research interests include database system and information retrieval.

Feifei Li is currently a Vice President of Alibaba Group, ACM Distinguished Scientist, director of the Database Products Business Unit of Alibaba Cloud Intelligence, and director of the Database and Storage Lab of DAMO academy. He has won multiple awards from ACM and IEEE and others. He is a recipient of the IEEE ICDCS 2020 Best Paper Award, ACM SoCC 2019 Best Paper Award Runner-up, IEEE ICDE 2014 10 Years Most Influential Paper Award, ACM SIGMOD 2016 Best Paper Award, ACM SIGMOD 2015 Best System Demonstration Award, IEEE ICDE 2004 Best Paper Award. He has been an associate editor, PC co-chairs, and core committee members for many prestigious journals and conferences, and has led the R&D efforts of building cloud-native database systems and products at Alibaba.

Aoying Zhou received the bachelor's and master's degrees in computer science from Sichuan University, China, and the Ph.D. degree from Fudan University, China, in 1988, 1985, and 1993, respectively. He is the vice president of East China Normal University (ECNU), China, professor of the School of Data Science and Engineering (DaSE). He is the winner of the National Science Fund for Distinguished Young Scholars supported by the National Natural Science Foundation of China (NSFC) and the professorship appointment under the Changjiang Scholars Program of Ministry of Education (MoE). He is a CCF (China Computer Federation) fellow, and associate editor-in-chief of the Chinese Journal of Computer. He served as general chair of the ER2004, vice PC chair of ICDE2009 and ICDE2012, and PC co-chair of VLDB2014. His research interests include Web data management, data management for data-intensive computing, in-memory cluster computing and distributed transaction processing, and benchmarking for big data and performance.