2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00164

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 old 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 adopts a history-aware strategy to provide fast online deduplication service. L-node also optimizes online restoration to realize high restore efficiency. Meanwhile, the G-node provides exact deduplication offline for the old versions, 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. Experimental results show that SlimStore can achieve fast deduplication, efficient restoration, and effective space reduction. Furthermore, SlimStore attains scalable deduplication and restoration.

I. Introduction

Enterprises used to store backup data with low-cost storage such as disks and tape libraries. Recently, as data scale has increased rapidly and cloud storage has developed, more and more users have chosen to backup data on the cloud due to its quick and convenient disaster recovery capability. A key benefit of the cloud is that it transparently auto-scales in response to workload changes, and its attractive consumption-based pricing saves huge expenses caused by early device investment. 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 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 back up the latest status of files on a regular basis. For instance, database users upload 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 duplicates to reduce the amount of data. Therefore, we build a cloud-based backup system, benefiting from OSS's low-cost storage and the reduction in data volume that comes from deduplication.

Data deduplication is a well-recognized approach to support large-scale backup storage systems. Three main indicators can measure the deduplication system: deduplicate speed, restore speed, and deduplication ratio. It is difficult to perform the best in all of them, so most of the existing work only focuses on one. DDFS [3], SiLO [4], and Sparse Indexing [5] make a trade-off between deduplicate speed and deduplication ratio. HAR [6], CBR [7], and Capping [8] rewrite the fragments 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, so the backup methods that maximize the deduplication ratio are usually chosen [9].

Backing up data on the cloud has led to some changes in the trade-off between three indicators. The first design goal of SlimStore is to hide the cost of OSS's high-latency I/O, which is orders of magnitude higher than attached storage, to provide fast online deduplication and restoration for new backup versions. Another goal comes from the conflicts between biased preferences for the indicators in the old and new backup versions and the physical data layout. For old versions, their storage costs are hoped to be lower because the data value decreases over time; for new versions, which are more likely to be restored, the restore speed is more concerned. However, the data distribution of the old version is more concentrated, while the new version is the opposite. So adjusting the data layout so that SlimStore can meet the users' preferences is our second design goal.

✉Corresponding author.
*Shanghai Engineering Research Center of Big Data Management.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00164

With these two goals in mind, SLIMSTORE is designed to build a cloud-based deduplication system. 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 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. 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, history-aware skip chunking is proposed that uses historical information to further accelerate online deduplication. Besides, SLIMSTORE performs offline reverse deduplication to accurately identify the missed duplicates to achieve exact deduplication.

For restoration, the system must combat the fragmentation on its physical storage, especially for the new versions of data. SLIMSTORE optimizes restoration at two levels. When restoring online, it takes an effective cache with full restore information to achieve high 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, and will not lose or be more conducive to the restore performance of new versions.

Our contributions are summarized as follows:

- We design 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 deduplicate 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, and with a history-aware approach 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 restore efficiency of new versions. It also outperforms the comparing methods by 1.39× in deduplication efficiency. Besides, SLIMSTORE can also achieve scalable deduplication and restoration.

II. ARCHITECTURE

A. Design Features

Separated storage and computation. Decoupling computation and storage is inherent in the cloud. 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. SLIMSTORE 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. Due to the incremental changes between versions, there are many duplicates between adjacent versions, so SLIMSTORE mainly eliminates duplicates between versions. Besides, SLIMSTORE also exploits the information of the historical version to accelerate deduplication.

Fast online deduplication and restoration. Our system aims to provide scalable and fast online deduplication and restoration. Thus we develop the stateless process node named L-node, which allows the system to dynamically deployed multiple L-nodes to cater to different users' workloads.

Deduplication systems suffer from performance loss due to the frequent access to the fingerprint index, which is extremely onerous in cloud environment since the index is placed on OSS. Thus L-node turns into a lighter method by detecting a historical version or similar file for each backup file, and by exploiting the similarity and locality in the detected file, duplicates are identified fast, thus avoiding a lot of OSS access.

As for restoration, because chunks of backup are physically scattered after deduplication, especially for new backup versions, which results in the restore performance degrades over time. However, new versions are more likely to be accessed. Therefore, we design a restore cache with full restore information to provide efficient online restoration.

Offline storage space optimization. We further propose G-node for two purposes. The first is to maximize the 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 offline, thus achieving exact deduplication of all files. And for another purpose of protecting the restore efficiency from declining over time, G-node adjusts new versions' physical storage with sparse container compaction in the backend. 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 improves the restore performance of new versions and reduce the storage cost of old versions.

B. System Components

Fig.1 provides the architecture of SLIMSTORE.

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

Container Store. After eliminating duplicates, the remaining non-duplicate chunks will be aggregated into fixed-size containers and persisted on the container store that resides on OSS. Besides, the container store also keeps chunks' status and offset. Since chunks in a container may have a close position in the backup file, once a chunk is accessed, other chunks in

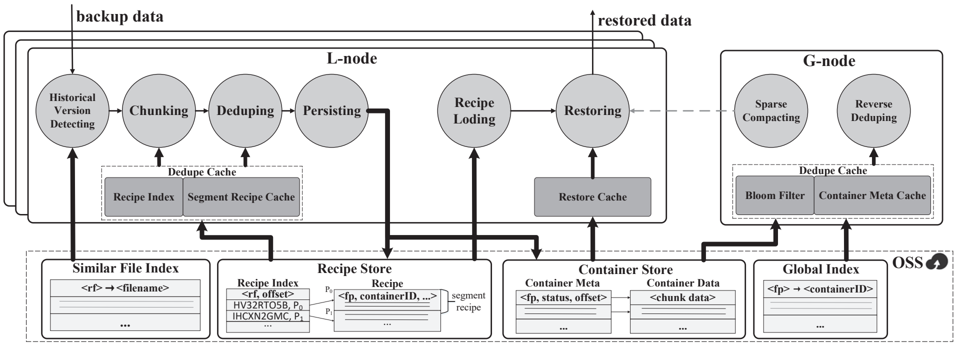


Fig. 1: System architecture of SLIMSTORE.

the container are also likely to be accessed, which gives rise to the *physical locality*. Therefore, accessing a container each time can efficiently utilize OSS's I/O bandwidth.

Recipe Store. Recipe describes the logical sequence of chunks of a backup file. A recipe consists of chunk records, and each chunk record is a triples ⟨ *fp*, *containerID*, *size*⟩, which represents the chunk's fingerprint, the ID of the stored container, and the chunk size. Due to the incremental changes of the backup files, the chunk sequences of two backup versions are similar. To make use of this property named *logical locality*, we exploit the structure called *segment*, which consists of several consecutive chunks, and their corresponding chunk records 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 them and locate their 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 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 [10], 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 deduplicate and restore jobs. With the help of the similar file index and recipe index, similar segments are fetched, and L-node exploits the logical locality in them to remove duplicates. The optimization named *history-aware skip chunking* is further proposed to improve its efficiency (Section III). As for the restore job, L-node reads chunks and splices them together based on the sequence of chunk records in the recipe (Section IV).

Noting that L-node is stateless, all the information required is loaded during the job execution. Thus L-node can expand elastically according to the running workload.

G-node. G-node runs offline, which is responsible for managing the storage space. In order to further identify duplication that is ignored by L-node, G-node uses reverse deduplication to filter containers generated by L-node, find and eliminate duplicate chunks (Section V), 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 IV-B).

III. DEDUPLICATION ON L-NODE

We first introduce the process of online deduplication, then a technique is 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. 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], [11], which selects the fingerprints that $mod\ \mathcal{R} = 0$ in a segment, where $\mathcal{R}$ is an adjustable parameter to control the sampling ratio. For a large file that cannot save all chunks in memory to find a similar file, we adopt the common solution that only samples the header chunks [12]. 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 a similar segment. If a chunk with the same fingerprint exists, it prefetches the corresponding segment recipe into the dedupe cache. Once a sampled chunk is matched, other chunks near it will also appear in this segment with a high probability according to the *logical locality*. By using this feature, a range

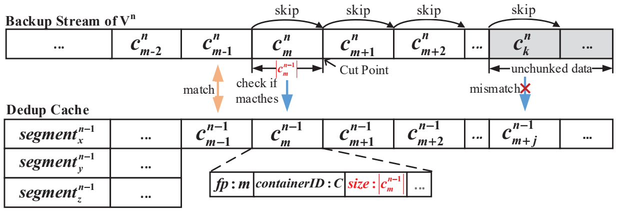


Fig. 2: History-aware skip chunking.

of duplicate chunks in the vicinity can be filtered efficiently. During the process, the metadata of a chunk including its fingerprint, size, and container ID is generated.

STEP 3. Segmenting and persisting. A number of consecutive chunks in the backup file 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. 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 [13] due to the complexity of Rabin hash. Some other algorithms such as FastCDC [14] use a simpler hash function, but running the byte-by-byte checking mechanism is still inefficient.

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 cut point based on the size. If 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 2 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, 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, so the deduplication speed is further accelerated.

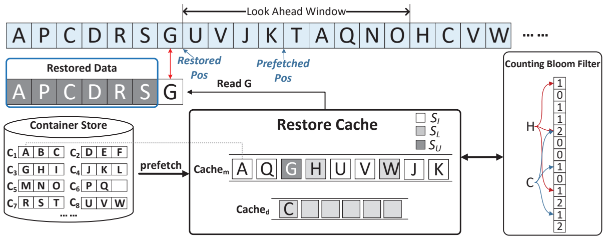


Fig. 3: Restore cache with full vision.

IV. RESTORE

As mentioned in Section II-A, restore performance suffers from the read amplification caused by fragmentation, which wastes a lot of OSS bandwidth. Therefore, we aim to reduce OSS bandwidth consumption to optimize restore performance.

A. *Full Vision Restore Cache*

Due to fragmentation issues, the conventional replacement algorithm like LRU has poor performance. For example, considering the LRU cache can hold up to 3 containers, when restoring the data stream in Fig 3, container C_6 is read to restore chunk P, but chunks in C_6 are too scattered, which causing repeated reading of C_6 to restore Q later. 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]. Existing works [6], [15] use a look-ahead window(LAW) to preserve 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 cannot 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, we design a restore cache with a full vision replacement policy, which uses the full information of chunk sequence in the recipe to protect chunks that will be accessed (includes the fragments that out of LAW) from being evicted, which completely avoiding repeated reading from OSS.

Fig 3 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, chunks that out of LAW can be preserved. 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 chunk (with the 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.

B. Sparse Container Compaction

Besides *large-span container* and *self-reference chunk*, *sparse container* is also needs to be noticed. Since the duplicate chunks of the new version are directed to the old versions, the chunks of a new version are stored 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 3, 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 referenced container is calculated, and the container whose utilization is lower than the threshold (e.g., 30%) is recorded as sparse container. After the current backup is finished, G-node starts the *sparse container compaction*(*SCC*) phase, merges chunks that are useful for the current version into new containers, and updates the file recipe to the new state. After compaction, 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, 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.

V. 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 IV-B caters to this principle, G-node further provides a *global reverse deduplication* technique.

Accurately identifying and removing the ignored deduplicates of fast deduplication can maximize the deduplication ratio and reduce storage costs. Considering that eliminating duplicate chunks that have already been stored 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. 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 new versions and deleting the duplicate chunk in containers of old versions, reverse deduplication reduces the data volume of old versions without sacrificing the restore performance of new versions.

The global index is used to accurately identify duplicates. During backup, G-node initiates a backend job to filter all chunks in new containers that generated by L-node to find if there is a duplicate stored in a container of the old version. If so, reverse deduplication deletes the duplicate chunk in the old container, and updates the location of the chunk in global index to the new container. A global bloom filter is used to quickly filter out unique chunks. Besides, when two chunks are identified as duplicates, according to the *physical locality* of the container, caching the meta of the old container can reduce the access number of Rocks-OSS to accelerate global deduplication. G-node only marks the duplicate chunk as deleted in the meta of the old container. When the percentage of deleted chunks exceeds the threshold (such as 20%), the container will be compacted and rewritten to OSS.

VI. EVALUATION

A. The Experimental Setup

We deployed SLIMSTORE on a cluster of seven cloud elastic compute services (ECS), each one is equipped with a 2.5GHz 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. We also implement HAR+OPT cache [6] and ALACC [15] to demonstrate the effectiveness of our optimization on the restore process.

We use S-DB as the dataset for evaluation. S-DB consists of 2.44TB database files, and each table is simulated by insert, update, and delete operations. By controlling the percentage of the modified data, the duplication ratio between versions of each table file varies from 0.65 to 0.95, and the average duplication ratio between versions is 0.84.

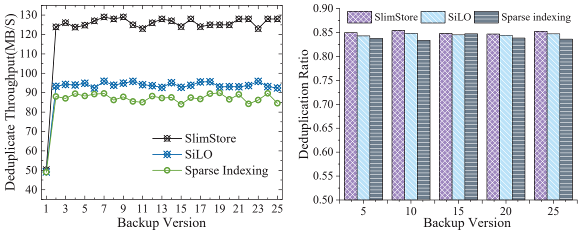
B. Deduplicate Performance

We evaluate the deduplicate performance on L-node to demonstrate the effect of fast online deduplication. Duplicate throughput shows the speed of deduplication, and the deduplication ratio represents the effectiveness, which is measured in terms of the percentage of deduplicates deleted after deduplication, i.e., $\frac{the\ size\ of\ duplicate\ data\ deleted}{total\ size\ before\ deduplication}$.

We evaluate the deduplication performance of SLIMSTORE and compare it with SiLO and Sparse Indexing. The default chunk size is set to 4KB for all three methods. Fig 4(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. Meanwhile, the performance improvement does not sacrifice the deduplication ratio of SLIMSTORE as shown in Fig 4(b).

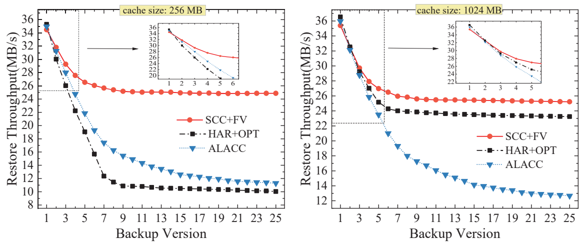
C. Restore Performance

We backed up 25 versions of S-DB continuously and then restored them under different cache sizes. The results are



(a) Deduplicate throughput (b) Deduplication ratio

Fig. 4: Comparison of fast online deduplication of SlimStore, SiLO, and Sparse indexing.



(a) 256MB restore cache (b) 1024MB restore cache

Fig. 5: Comparison of restore performance.

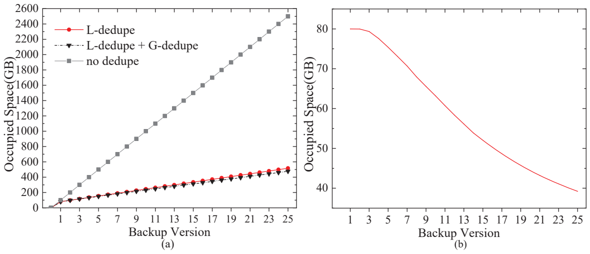


Fig. 6: Effect of space management

shown in Fig 5. 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. Because the unit of OPT cache is container, many useless chunks occupy the precious cache space, which causes OPT cache to have the worst performance (the partial view of Fig 5(a)). FV cache and ALACC adopted chunk-based cache, so they perform better than OPT cache when cache is small. With full restore information, FV cache can address the fragments that exceed the vision of LAW, which ensures all containers only be read once, so FV cache outperforms ALACC.

As for the 1024 MB cache, it can preserve more useful chunks, so the main performance loss comes from sparse containers. With SCC, the read amplification caused by sparse container is effectively alleviated, which prevent the restore performance from declining after version 7 as shown in Fig 5(b). Because ALACC has no optimization on sparse containers, so it has the worst restore performance. HAR has a similar 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. Therefore, SCC and FV cache perform best in combating fragmentation compared with existing methods.

D. Space Cost

Fig 6 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), L-dedupe can achieve a 4.8× reduction in space consumption compared to not apply deduplication, 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.

Fig 6(b) shows the space occupied by version 0 as time goes by. It can be seen that the occupied space is gradually decreasing. Because sparse container compaction and reverse deduplication will transfer some data of old versions to new versions. Therefore, the occupied space of old versions is decreasing over time, which meets user needs that spend less money to store old backup data.

VII. Conclusion

This paper presents SlimStore, a cloud-based deduplication system that provides online deduplicate 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. In isolation, each of these techniques 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 high-speed deduplication and restoration, and can effectively eliminate duplicate data to reduce the storage costs.

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

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

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

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

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

[15] 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.