

# Revisiting Fragmentation for Deduplication in Clustered Primary Storage Systems

Lin Wang<sup>1</sup>, Yuchong Hu<sup>1,2</sup>, Shilong Mao<sup>1</sup>, Mingqi Li<sup>1</sup>, Ziling Duan<sup>1</sup>, Yue Huang<sup>1</sup>, Leihua Qin<sup>1</sup>, Dan Feng<sup>1</sup>  
Zehui Chen<sup>3</sup>, Ruliang Dong<sup>3</sup>

<sup>1</sup>Huazhong University of Science and Technology

<sup>2</sup>Shenzhen Huazhong University of Science and Technology Research Institute

{wanglin2021, yuchonghu, goushengdd, mingqili1006, duanziling, huangyue, lhqin, dfeng}@hust.edu.cn

<sup>3</sup>Huawei Technologies Co., Ltd

{chenzehui2, dongruliang1}@huawei.com

**Abstract**— To improve storage efficiency in large-scale clustered storage systems, deduplication that removes duplicate chunks has been widely deployed in *distributed* ways. Many distributed deduplication-related studies focus on backup storage, and some recent studies focus on deploying deduplication in clustered *primary* storage systems which store active data. While fragmentation is one of the traditional challenges in backup deduplication, we observe that a new fragmentation problem arises when performing deduplication in the clustered primary storage system due to the system's concurrent file writes. However, we find that existing state-of-the-art methods that address traditional fragmentation in backup deduplication fail to work effectively for the new fragmentation problem, as they significantly incur additional redundancy or lower the deduplication ratio.

In this paper, we revisit fragmentation-solving methods in memory management and our main idea is inspired by the classic garbage collection methods in memory management: relocating fragments consecutively. Based on the idea, we propose an effective deduplication mechanism for clustered primary storage systems, ReoDedup, which applies: i) a cosine-similarity based chunk relocating algorithm that aims to minimize the fragmentation; ii) an adjacency-table based relocating heuristic that reduces the relocating's time complexity by placing two chunks residing in the same file consecutively; and iii) an index-remapping update scheme that alleviates the extra fragmentation caused by updates. We implement ReoDedup atop Ceph and our cloud experiments show that the average read throughput of ReoDedup can be increased by up to  $1.72\times$  over state-of-the-arts, without any deduplication ratio loss.

## I. INTRODUCTION

With the explosive growth of data, how to cost-effectively manage the large-scale clustered storage systems has become one of the most challenging and important tasks. Deduplication is an effective technique to reduce storage costs and improve storage efficiency for clustered storage systems, called *distributed deduplication* [1], [2], [3], which has been widely deployed in production (e.g., like Ceph [2], [3]). The basic technique of deduplication [4], [5], [6] is to split the input file into multiple *chunks*, remove the redundant ones and only save the unique ones. Deduplication in backup storage systems often introduces a traditional *fragmentation* problem (i.e., the deduplicated chunks of a single file are often stored at non-consecutive disk locations), so many state-of-the-art methods

(e.g., rewriting [7], [8] and superchunking [5]) are proposed to address the traditional fragmentation problem.

While many distributed deduplication studies [9], [10] focus on backup storage, recent studies [11], [12], [1] increasingly target at *primary* (non-backup) storage systems, which often stores active data that is frequently accessed. Interestingly, we observe that a *new* fragmentation problem arises when applying deduplication to distributed primary (called *distributed primary deduplication*), since the distributed system's concurrent file writes will make their deduplicated chunks intermixed, which makes these files fragmented. Unfortunately, we also observe that the state-of-the-art methods (rewriting [7], [8] and superchunking [5]) that address the classic fragmentation problem fail to work effectively for the new fragmentation problem in clustered primary deduplication, since they either incur lots of additional redundancy (rewriting) or lowers the deduplication ratio significantly (superchunking), which will be specified in §III.

To address the new fragmentation problem in clustered primary deduplication, we revisit fragmentation-solving methods in memory management [13] and are inspired by the classic garbage collection methods in memory management: *relocating fragments consecutively*, so as to reduce fragmentation. Unlike rewriting and superchunking, the relocating operation that only changes the chunks' locations neither incurs any redundancy nor impacts the deduplication ratio. However, how to perform relocating efficiently remains to be explored.

In this paper, we propose a **relocating based deduplication** mechanism, ReoDedup, which relocates fragmented chunks in a consecutive way. We first model the relocating problem and propose a cosine-similarity based chunk relocating algorithm (CSR) that aims to minimize the fragmentation. We also propose an adjacency-table based relocating heuristic (ATR) that reduces the relocating's time complexity via relocating two chunks that reside in the same file. We further propose an index-remapping update scheme (IRU) that addresses the extra fragmented chunks caused by frequent updates in primary storage. Our contributions are as follows.

- We are the first to observe the new fragmentation problem in clustered primary deduplication, and then propose

ReoDedup that relocates fragmented chunks consecutively, without incurring additional redundancy or lowering the deduplication ratio caused by existing schemes (§III).

- We model ReoDedup and propose two relocating algorithms CSR and ATR, with the goals of optimizing the fragmentation and lowering the time complexity, respectively. We further enhance ReoDedup via a remapping algorithm IRU (§IV).
- We implement ReoDedup atop Ceph and our experiments via Alibaba Cloud show that ReoDedup increases the average read throughput by up to  $1.72\times$  over the state-of-the-arts without any deduplication ratio loss (§V). The source code of ReoDedup is available at: <https://github.com/YuchongHu/ReoDedup>.

## II. BACKGROUND AND RELATED WORK

We introduce the basics of deduplication (§II-A), describe deduplication (§II-B) and the traditional fragmentation problem (§II-C) in backup storage, present existing state-of-the-art schemes for fragmentation (§II-D), and show the deduplication in primary storage (§II-E).

### A. Deduplication Basics

Deduplication is an effective technique to reduce the redundant data for backup storage or primary storage, and this paper focuses on the latter, which becomes increasingly popular in large-scale settings. Basically, deduplication has the following key steps. First, deduplication splits files into *chunks*, called *chunking* [4], [14]. Second, deduplication computes a cryptographically secure hash signatures (e.g., SHA-1 [5], [4]) for each chunk, called *fingerprinting*. Third, deduplication checks each chunk's fingerprint to detect duplicates, called *indexing* [6], [15], [16]. Finally, duplicate chunks are removed and only unique chunks are saved, such that files can be *restored* by unique ones. Additionally, there is an important metric for evaluating the effectiveness of deduplication, called *deduplication ratio*, which can be calculated as [10]:

$$\text{deduplication ratio} = \frac{S_b}{S_a} \quad (1)$$

where  $S_b$  is the size of data before deduplication (the size of the original data), and  $S_a$  is the size of data after deduplication (the size of the stored data).

**Distributed deduplication.** With the surge of clustered storage systems for handling massive amount of data, deduplication has been widely deployed in distributed ways [5], [1], [2], [3] (called *distributed deduplication* [12]), which often routes each chunk based on fingerprint-based routing mechanisms to a specific node performing deduplication [5]. In this paper, we focus on distributed deduplication in primary storage systems, which we call *distributed primary deduplication*.

### B. Deduplication in Backup Storage

Backup storage stores *inactive* and *immutable* data that is rarely used or not accessed frequently. Backup deduplication systems usually write and read the deduplicated chunks in a

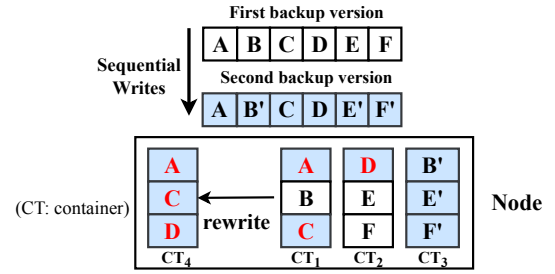


Fig. 1. Fragmentation in backup deduplication and rewriting schemes.

large fixed-size unit, called *container* (often 4MB [10]) for achieving the maximum sequential throughput of hard drives. Workloads in backup storage are usually a stream of backup versions (i.e., successive snapshots of data) that are often written *sequentially* [8], indicating that the chunks of a backup stream are likely to appear in approximately the same sequence in every full backup version with a significant probability [4], called *stream locality* [8]. This feature is extensively utilized to enhance backup deduplication performance [14], [7].

### C. Fragmentation in Backup Deduplication

One of the traditional challenges in backup deduplication is *fragmentation*, which indicates that restoring a backup version requires reading the duplicated chunks that are scattered among previous backup versions. We illustrate this traditional fragmentation problem in Figure 1. Assume that in a backup deduplication system, the container size is three chunks and two backup versions are written into the system sequentially. The first backup version has six chunks  $A, B, C, D, E$ , and  $F$ , and is stored sequentially in two containers  $CT_1$  and  $CT_2$ . The second backup version has six chunks  $A, B', C, D, E'$ , and  $F'$ , where three chunks  $B', E'$ , and  $F'$  are changed from  $B, E$ , and  $F$  of the first backup version respectively, so it needs to store chunks  $B', E'$ , and  $F'$  in another container  $CT_3$ . This causes fragmentation of the second backup version, because it has to read *three* containers  $CT_1, CT_2$  and  $CT_3$  for restoring all its chunks, while the first backup version only needs to read two containers  $CT_1$  and  $CT_2$  for restoring all its chunks.

### D. State-of-the-arts for Fragmentation in Backup Deduplication

To address the above traditional fragmentation problem in backup deduplication, there are mainly two types of schemes (rewriting [7], [8] and superchunking [5]) are proposed, specified as below.

**Rewriting schemes.** Most studies [7], [8] use the *rewriting* schemes, which add storage redundancy to reduce fragmentation by selectively rewriting some duplicated fragmented chunks. Due to the stream locality (§II-B) in backup storage, the added redundancy is often not much [7], [8]. As shown in Figure 1, without rewriting (as stated in §II-C), the second version has to read three containers  $CT_1, CT_2$  and  $CT_3$ . Alternatively, based on rewriting, we select the duplicate chunks  $A, C$ , and  $D$  and rewrite them to a *new* container

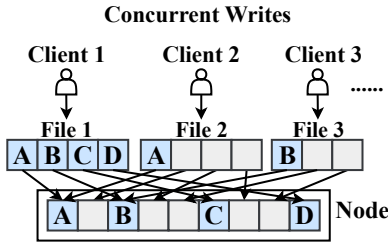


Fig. 2. The new fragmentation problem in distributed primary deduplication.

$CT_4$ , such that the second version just needs to read only two containers  $CT_3$  and  $CT_4$ , thus reducing the fragmentation.

**Superchunking schemes.** Some studies [5] use the *superchunking* schemes, which group a collection of consecutive chunks into a *superchunk*, where the stream locality (§II-B) ensures that similar superchunks are very likely to exist in every backup version. In this way, each superchunk preserves the stream locality among its composed chunks and thus reduces the fragmentation, while maintaining a high deduplication ratio since similar superchunks have many identical chunks that can be deduplicated.

#### E. Deduplication in Primary Storage

Primary storage stores *active* and *mutable* data that is frequently accessed and serves multiple clients that usually write their files concurrently [17]. In primary storage, read requests are very frequent and thus need to be served more efficiently than the restore operations of backup storage [12], so many studies [7], [18] focus on reducing the read latency in primary deduplication. In this paper, we find a new fragmentation problem in distributed primary deduplication (see §III-A for details), so our main goal is to reduce the fragments so as to improve the read performance in distributed primary deduplication.

### III. ANALYSIS AND MOTIVATION

We introduce a new fragmentation problem in distributed primary deduplication via analysis and show the limitations of existing schemes (§III-A), and show our main idea based on a motivating example (§III-B).

#### A. Analysis

**Problem significance analysis: a new fragmentation problem arises in distributed primary deduplication.** We find that a *new* fragmentation problem is caused by the concurrent file writes of multiple clients in distributed primary deduplication. As illustrated in Figure 2, *File 1*, *File 2*, and *File 3* from *Client 1*, *Client 2* and *Client 3*, respectively, are split into chunks for deduplication and being written to the system concurrently. Since the chunks of each file (say *File 1*) and the chunks from other files are being written concurrently, their chunks will be intermixed together on the storage node, which makes the chunks of *File 1* scattered, thus causing fragmentation of *File 1*.

To show how fragmented the files are after distributed primary deduplication, we conduct a simulation as below.

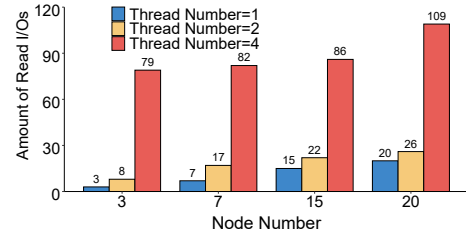


Fig. 3. Significance of the new fragmentation problem via simulation #1: The amount of read I/Os versus the number of write threads and nodes.

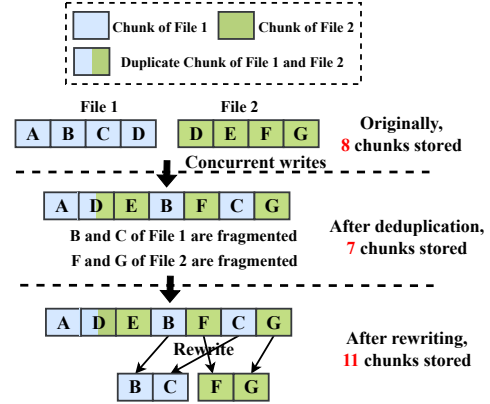


Fig. 4. Limitation of the rewriting scheme.

**Simulation #1.** We simulate a cluster with distributed primary deduplication. The cluster has multiple nodes and each node can only remove its own duplicated chunks. We use multiple write threads to write files in the cluster to simulate the concurrent file writes. We set the node number ranging from 3 to 20 and the write thread number ranging from 1 to 4, as in distributed deduplication studies [5], [3]. We split each file into chunks for deduplication and route each chunk based on its fingerprint to a specific node as [5]. We use every thread to write a file that is generated by FIO [19], which is a widely-used tool that generates files to evaluate and benchmark the I/O performance. We evaluate the average amount of read I/Os when reading all generated files.

**Result.** Figure 3 shows that the amount of read I/Os increases greatly with the number of write threads. For example, the amount of read I/Os can be increased by up to 96.2% when the write thread number changes from one to four when the node number is three. Therefore, the new fragmentation problem is serious in distributed primary deduplication, especially in large amounts of clients.

**Discussion.** One may wonder the difference between the traditional fragmentation in backup deduplication (§II-C) and the above new fragmentation in distributed primary deduplication. The major difference is that the former arises from *sequential writes* (§II-B) of a stream of backup versions, while the latter arises from *concurrent writes* (§II-E) for different client requests. The difference will make existing schemes (§II-D) for the former fail to work for the latter, which will be specified in the following.

**Limitation analysis of rewriting.** We find that using the

TABLE I  
LIMITATION OF THE SUPERCHUNKING SCHEME VIA SIMULATION #2: THE  
DEDUPLICATION RATIO VERSUS SUPERCHUNK SIZE.

Superchunk size	1	4	16	64	256
Home	1.250	1.107	1.072	1.070	1.067
Mail	2.326	1.538	1.266	1.088	1.088
Web-vm	1.136	1.103	1.089	1.088	1.087

rewriting schemes in distributed primary deduplication will incur significant storage redundancy. As illustrated in Figure 4, each of files *File 1* and *File 2* is split into four chunks for deduplication and being written concurrently into the disk. The concurrent writes cause most chunks of *File 1* and *File 2* to be intermixed, which makes each file quite fragmented. Then if we use the rewriting schemes to reduce the fragmentation, we have to rewrite the two fragmented chunks of *File 1* (i.e., *B* and *C*) and two fragmented chunks of *File 2* (i.e., *F*, and *G*) to ensure that each file can be read by two sequential I/Os, which reduces the fragmentation as in Figure 1. Clearly, the rewriting schemes incur additional redundancy (i.e., *B, C, F*, and *G*), so they cannot be cost-effectively applied to distributed primary deduplication.

**Reason.** The reason why rewriting schemes are ineffective in distributed primary deduplication is that unlike backup deduplication that leverages the stream locality (§II-B), distributed primary storage systems often face different concurrent writes, so their written data hardly have the stream locality, which makes rewriting schemes have to add substantial additional redundant chunks to maintain sequential reads for distributed primary storage.

**Limitation analysis of superchunking.** We find that the superchunking schemes greatly lower the deduplication ratio in clustered primary storage. We conduct the following simulation to verify this limitation.

**Simulation #2.** We simulate a cluster composed of 20 storage nodes and vary the superchunk size from 1 to 256 (similar setting in [5]). Then we use multiple threads to write these chunks to the cluster. We evaluate the deduplication ratio in three real-world traces in primary storage [20]: Home, Mail, and Web-vm, adopted by many deduplication studies [20], [18] and open-sourced at [21].

**Result.** Table I shows that the deduplication ratio decreases with the superchunk size. Even if the superchunk size slightly increases from one to only four chunks, the deduplication ratio still drops significantly (by 33.9% for Home trace).

**Reason.** The reason why the superchunking schemes are ineffective in distributed primary deduplication is also lack of stream locality in the primary storage data. Without the stream locality, even if some chunks can be deduplicated, their superchunks may not be similar and thus are routed into different nodes, so these chunks that were supposed to be deduplicated cannot perform deduplication at different nodes, thus degrading the deduplication ratio drastically.

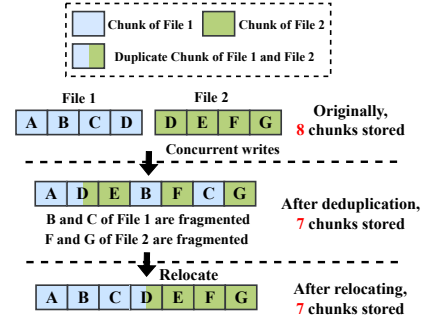


Fig. 5. Our method.

## B. Motivation

**Traditional wisdom.** To address the new fragmentation problem in distributed primary deduplication, we revisit the classic garbage collection methods in memory management [13] that often *relocate* fragments in a consecutive manner, aiming for reducing fragmentation. This inspires us to consider if we can rethink this classic relocating schemes to alleviate the new fragmentation in distributed primary deduplication.

**Motivating example.** Figure 5 shows a motivating example of our main idea based on the above relocating wisdom. Specifically, we relocate the chunks such that *File 1*'s entire chunks and *File 2*'s entire chunks can be read in a sequential way. In this way, we can reduce the fragmentation and read *File 1* and *File 2* sequentially by only changing the chunk location, while incurring no redundancy (unlike rewriting) and maintaining high deduplication ratios (unlike superchunking).

Compared to the rewriting schemes in Figure 4, our relocating method does not incur any redundancy (the number of chunks is both seven before and after our relocating method, while in Figure 4, the number of chunks is seven before rewriting but 11 after rewriting, which causes  $11 - 7 = 4$  chunks redundancy).

Compared to the superchunking schemes in Table I, our relocating method does not lower the deduplication ratio (the deduplication ratio is both  $8/7 = 1.143$  before and after our relocating method, while in Table I, the deduplication ratio of the superchunk schemes drops significantly with the superchunk size).

**Main idea.** The above motivating example motivates us to propose ReoDedup, a relocating based deduplication mechanism that places fragmented chunks consecutively so as to reduce the fragmentation in distributed primary deduplication. However, how to perform effective (fragmentation reduction) and efficient (resource reduction) relocating remains to be explored, which will be specified in §IV.

## IV. DESIGN

We design ReoDedup with the following goals:

- **Optimized chunk relocating.** We first model ReoDedup via cosine similarity, then based on this model propose a

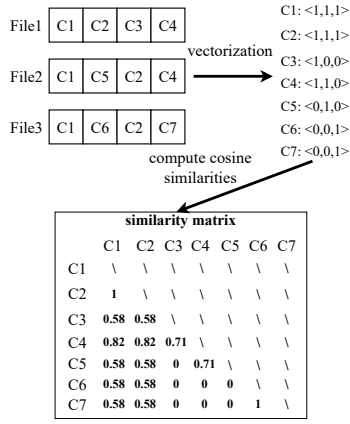


Fig. 6. An example of modeling.

brute-force relocating algorithm that minimizes the fragmentation, and finally propose a greedy algorithm by always choosing the two chunks that have the largest cosine similarity to relocate (§IV-A).

- **Low time complexity.** We propose a chunk relocating heuristic that reduces time complexity via placing chunks that simultaneously reside in the same file consecutively based on an adjacency table (§IV-B).
- **Low fragmentation caused by updates.** We reduce the additional fragmentation caused by updates via remapping the index of the updated chunks (§IV-C).

#### A. Cosine-Similarity Based Relocating

**Challenge.** To optimize the chunk relocating, we aim at minimizing the overall fragmentation. However, it is challenging to define the benefit of relocating chunks, where the benefit means how many fragmented chunks can be reduced if we place these chunks consecutively.

**Design idea.** If many files have common chunks, then re-ordering them in a sequential way can clearly reduce lots of fragmentation among these files. Thus, we tend to relocate and place chunks that simultaneously exist in more files consecutively.

To this end, we introduce *cosine similarity* [22], which is a metric used to assess the similarity between two vectors, in a way that measures the cosine of the angle between the two vectors and judges if the two vectors are pointing to roughly the same direction. The definition of this metric motivates us to model ReoDedup in two steps: first, we assign each chunk a vector to show which files it resides; second, we calculate the cosine similarity of any two chunks' vectors to judge the number of files that two chunks simultaneously exist in. In this way, if two chunks' assigned vectors have a larger cosine similarity, then they simultaneously exist in more files, so relocating them can reduce more fragmentation.

**Model.** Based on the design idea, we describe how to model ReoDedup as follows.

**Step 1 (Vectorization).** We first assign each chunk a vector to show which files it resides in, called *vectorization*. The length of the vector is equal to the number of files. We number each

file, and the  $i^{th}$  file corresponds to the  $i^{th}$  bit of the vector. If the chunk exists in the  $i^{th}$  file, then set the  $i^{th}$  bit of the vector to 1, otherwise set to 0. In this way, each chunk has a corresponding 0-1 vector that identifies the files in which the chunk exists. As shown in Figure 6, we have three files *File1*, *File2* and *File3*, and seven chunks  $C_1, C_2, \dots$ , and  $C_7$ . For  $C_1$ , we find that it exists in all three files, so we assign a vector  $\langle 1, 1, 1 \rangle$  to  $C_1$ .

**Step 2 (Calculate cosine similarity matrix).** We then quantify the fragmentation reduction when placing two chunks consecutively. Specifically, after Step 1 that vectorizes two chunks, we have two vectors  $A$  and  $B$  (corresponding to the two chunks), we then calculate the cosine similarity between  $A$  and  $B$  based on Equation (2) [22]:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}} \quad (2)$$

Next, we compute the cosine similarity between any two vectors among  $n$  vectors, and all the cosine similarities can construct a *similarity matrix* shown in Figure 6, which is  $n \times n$  and the  $i^{th}$  row and the  $j^{th}$  column corresponds to the cosine similarity between  $i^{th}$  chunk and  $j^{th}$  chunk. For example, for the two chunks  $C_1$  and  $C_2$  in Figure 6, the cosine similarity between them can be calculated as

$$\cos(\langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle) = \frac{3}{\sqrt{3} \cdot \sqrt{3}} = 1 \quad (3)$$

Based on the above two steps of modeling, we have all the cosine similarities between any two chunks, such that for any given chunk relocating (from an original chunk location to a new chunk location), we evaluate its benefit by subtracting the sum of the cosine similarities of any two sequential chunks in the new location from the sum of the cosine similarities of any two sequential chunks in the original order. For example, assume that the original location of the seven chunks in Figure 6 is  $C_1, C_2, C_3, C_4, C_5, C_6$ , and  $C_7$ , then we can calculate the sum of the cosine similarities of any two sequential chunks based on the similarity matrix, which is equal to  $1 + 0.58 + 0.71 + 0.71 + 0 + 1 = 4$ . Assume that we obtain a new location  $C_5, C_2, C_1, C_4, C_3, C_6$ , and  $C_7$  based on the chunk relocating, we can also calculate the sum of the cosine similarities of any two sequential chunks, which is equal to  $0.58 + 1 + 0.82 + 0.71 + 0 + 1 = 4.11$ . Therefore, the benefit of the chunk relocating is  $4.11 - 4 = 0.11$ .

**Design details.** Based on the above model, we propose two cosine-similarity based relocating (CSR) algorithms.

**Brute-force algorithm.** We optimize the chunk relocating by maximizing its benefit, i.e., the difference between the sum of the cosine similarities of any two sequential chunks in the new location and that in the original location. We can find this maximum difference via traversing all possible new chunk orders and choose the one with the largest difference from the original chunk order. Since for  $n$  chunks, there are  $n!$  possible orders in total. Therefore, the time complexity of obtaining the optimal location is at  $O(n!)$  level.

TABLE II  
RATIO OF CHUNKS THAT EXIST IN MORE THAN ONE FILE.

	Home	Mail	Web
Ratio	12.9%	19.3%	12.6%

**Greedy algorithm.** We see that the time complexity of the brute-force algorithm is too high to be used in practice, so we propose a greedy algorithm to make the cosine-similarity based chunk relocating algorithm practical. Instead of traversing, we always place two chunks with the *largest* cosine similarity consecutively, and compare the sum of the cosine similarities of any two sequential chunks in the new location with that in the original location and choose the larger one. Based on the idea, we specify this greedy algorithm in the following steps:

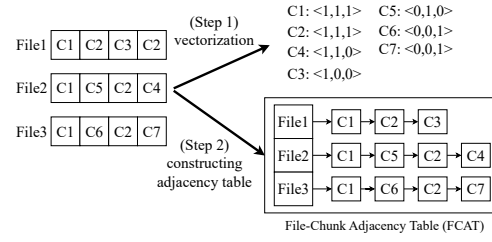
- (Step 1) We first sort each row of the similarity matrix in descending order to find the largest cosine similarity for each chunk in  $O(1)$  time complexity;
- (Step 2) For each chunk (say  $C_1$ ), we obtain the chunk (say  $C_2$ ) based on the similarity matrix that has the largest cosine similarity with  $C_1$ ;
- (Step 3) we judge if  $C_2$  has already had a consecutive chunk. If not, we place  $C_1$  and  $C_2$  consecutively; If yes, we continue to choose another chunk based on the similarity matrix and judge it again;
- (Step 4) Loop Step 2 and Step 3, until each chunk (except the last chunk) has a consecutive chunk;
- (Step 5) We compare the sum of the cosine similarities of any two sequential chunks in the new location with that in the original location and choose the larger one.
- (Step 6) We place the two chunks selected in every loop consecutively.

**Time complexity.** We analyze the above greedy algorithm's time complexity. The time complexity of this greedy algorithm is equal to the sum of the above six steps' time complexity. Assume that we have  $n$  chunks in total. Then Step 1's time complexity is  $O(n^2 \times \log(n))$  because there are  $n$  rows in the similarity matrix to be sorted and sorting each row has a time complexity of  $O(n \times \log(n))$ . Steps 2, 3, 5 and 6 are simple and all have a time complexity of  $O(1)$ . Step 4 has a time complexity of  $O(n)$  since there are  $n$  chunks in total and for each chunk we need to perform Steps 2 and 3. Therefore, the total time complexity of greedy algorithm is  $O(n^2 \times \log(n))$ .

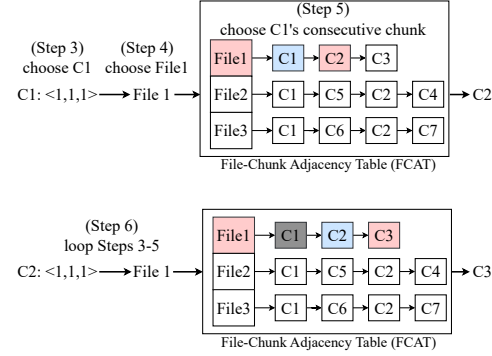
### B. Adjacency-Table Based Relocating

**Challenge.** We can see that compared to the brute-force algorithm of CSR, the greedy one reduces the time complexity, but it is still too high to be deployed in practice, so we need to further lower the time complexity while maintaining significant fragmentation reduction.

**Insight.** We observe that the ratio of chunks that exist in more than one file is relatively small. To verify this insight, we conduct a test based on Home, Mail, and Web traces (see §III-A). We measure the ratio of the number of chunks that



(a) Vectorization and constructing FCAT



(b) Relocating based on FCAT

Fig. 7. Illustration of FCAT.

exist in more than one file to the total number of chunks. Table II shows that all ratios in these three traces are small (e.g., 12.9% in Home, 19.3% in Mail, and 12.6% in Web). This motivates us to perform relocating only the chunks that simultaneously exist in *single* file.

**Design idea.** Based on the insight, our proposed heuristic only considers the chunks that simultaneously exist in *single* file and places them consecutively, instead of relocating chunks that simultaneously exist in *multiple* files (like §IV-A).

This heuristic can significantly reduce the time complexity, since there are a huge number of possibilities for finding the two chunks residing in multiple files.

In addition, we further boost the heuristic via constructing a new metadata structure based on an adjacency table [23]. Specifically, the adjacency table consists of multiple sequentially stored entries and each entry is associated with a list of related elements, so for each chunk, the adjacency table can help it to quickly find the chunk (as *element*) that is in the same file (as *entry*), thus speeding up the relocating, specified as below.

**File-chunk adjacency table.** Based on the design idea, we construct a new metadata structure, called *File-chunk adjacency table (FCAT)*, which is an adjacency table that uses a sequential array to store entries representing all files and each entry points to a list of elements that store the chunks' indices of this file. In this way, for any given file (entry), we can quickly obtain the chunks (elements) that this file has in  $O(1)$  time complexity since the entries are sequentially stored. FCAT will not consume much memory space, since each entry or element only consists of an index whose size is 4 bytes and a pointer whose size is 8 bytes. Assume the chunk size is 4KB,

then relocating each 1GB data only consumes 3MiB memory space by FCAT (accounts for less than 0.3%).

**Design details.** We boost this heuristic via FCAT in the following steps:

- (Step 1) We first assign each chunk a vector, which is the same as the vectorization step in §IV-A;
- (Step 2) We then construct a FCAT based on the metadata that contains each file has which chunks;
- (Step 3) We choose a chunk that has never been considered (say  $C_1$ ), and we find any one file (say  $File_1$ ) that it exists in based on its vector;
- (Step 4) We obtain the list table containing which chunks that  $File_1$  has based on the FCAT;
- (Step 5) We choose another chunk (say  $C_2$ ) that exists in  $File_1$  as  $C_1$ 's consecutive chunk while ensuring that  $C_2$  have not been considered in Step 3;
- (Step 6) Loop Steps 3-5, until all chunks (except for the last chunk) have their consecutive chunks.
- (Step 7) We place the two chunks selected in every loop consecutively.

We illustrate this heuristic in Figure 7. Assume that three files *File 1*, *File 2* and *File 3* are written in the system. All these three files have four chunks shown in Figure 7(a). Step 1 vectorizes all chunks  $C_1, \dots, C_7$  and Step 2 constructs a FCAT in which the entries represent three files *File 1*, *File 2* and *File 3* and each entry points to a list of chunk indices in this file. In Step 3, we choose  $C_1$  and obtain from its vector  $\langle 1, 1, 1 \rangle$  that it exists in *File 1*, *File 2* and *File 3*, and randomly choose one file (say *File 1*). Step 4 obtains the list table that contains chunks  $C_1 \rightarrow C_2 \rightarrow C_3$  of *File 1*. In Step 5, in the list table, we find another chunk  $C_2$  which has not been considered before. Step 6 loops Steps 3-5 where we choose  $C_2$  in Step 3. Step 7 places all the above chosen two chunks selected in every loop consecutively, e.g., at the first loop, we place  $C_1$  and  $C_2$  consecutively.

**Time complexity.** The time complexity of this heuristic is equal to the sum of the above seven steps. Assume that we have  $n$  chunks and  $m$  files, Step 1's time complexity is  $O(n \times m)$  because for each chunk, vectorizing it has a time complexity of  $O(m)$  (see §IV-A). Step 2's time complexity is  $O(n)$  because we insert  $n$  chunks into FCAT and the time complexity of inserting is  $O(1)$ . Steps 3, 4, and 7 are simple and all have a time complexity of  $O(1)$ . Step 5 also has a time complexity of  $O(1)$  since we delete the chunks that have already had consecutive chunks, which makes Step 5 not require looping too many times. Step 6 has a time complexity of  $O(n)$  since there are totally  $n$  chunks that needs to be relocated. Therefore, the overall time complexity of this heuristic is  $O(n \times m)$  which is much less than that of CSR (see §IV-A).

### C. Index-Remapping Updating

**Challenge.** Different from backup storage that stores on immutable backup data, primary storage often stores active data that may need to be updated frequently [12]. This difference makes us consider whether updates will have an impact on

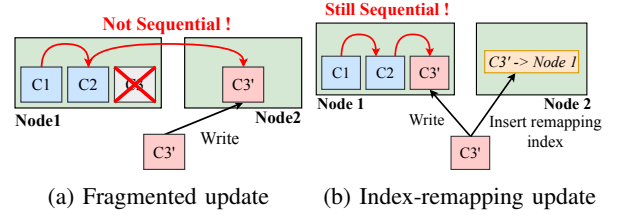


Fig. 8. Fragmented update and index-remapping update.

in distributed primary deduplication. In fact, we observe that distributed primary deduplication will incur extra fragments during updates, called *fragmented updates*. As illustrated in Figure 8(a), assume that three chunks  $C_1$ ,  $C_2$ , and  $C_3$  are sequentially stored within *Node 1*, and the *old chunk*  $C_3$  is updated to a *new chunk*  $C'_3$ . Since distributed deduplication always routes chunks based on their fingerprints (II-A), so the new chunk  $C'_3$  that has a new fingerprint has to be re-routed from its original node (i.e., called *old node*) to another node (i.e., called *new node*), such that  $C'_3$  is non-sequential from  $C_1$  and  $C_2$ , thus incurring fragments.

**Design idea.** To address the fragmented update issue, our idea is to propose an index-remapping update scheme (IRU) that writes the new chunk back to the old node by remapping the new chunk's index from the new node to the old node, thus avoiding the fragmented updates.

**Design details.** Based on the design idea, we elaborate the IRU scheme as follows. (Step 1) We first update the content of the old chunk to the new chunk; (Step 2) We then insert a remapping index, which remaps the new chunk from the new node to the old node, to the new node. As illustrated in Figure 8(b), when an old chunk  $C_3$  in *Node 1* is updated to a new chunk  $C'_3$ , we still write  $C'_3$  into *Node 1* and insert a remapping index  $C'_3 \rightarrow \text{Node 1}$  to the new node *Node 2* which was supposed to store  $C'_3$  based on its fingerprint.

In this way, the updated chunk can be written to the old node, which will not incur fragmented updates. When reading the updated chunk, system will first send a read request to the new node. If the system receives a remapping index that contains a node ID, then system can be redirected to the  $ID^{th}$  node to read the updated chunk.

## V. EVALUATION

We show the implementation of ReoDedup (§V-A), the experimental setup (§V-B) and the experiment results (§V-C).

### A. Implementation

**Framework.** We implement ReoDedup in C++ with about 12000 lines of code (LoC) atop *Ceph* [24] and *Redis* [25]. The overall framework of our implementation is shown in Figure 9. Users write, read and update their data via the *client*. The system is organized in a decentralized way and each node in the cluster runs an *agent* daemon, which has multiple *worker* threads that provide some interfaces to process and response the requests from the *client* in parallel.

**Deduplication plugin.** We implement a deduplication module in the *worker*, including *chunking* and *fingerprinting* modules.

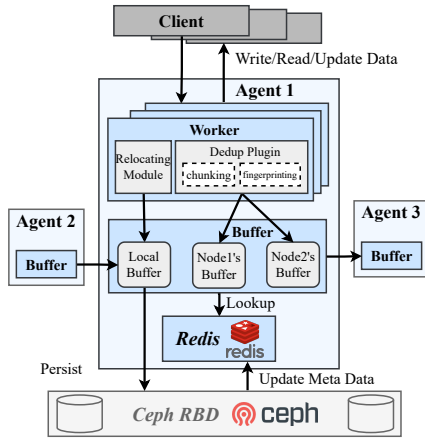


Fig. 9. Framework of implementation.

Chunking and fingerprinting modules are implemented as a plugin, so multiple chunking algorithms [26], [4], [14] and fingerprinting algorithms (e.g., SHA-1, SHA-256, MD5, etc.) can be supported.

**ReoDedup algorithms.** We implement CSR, ATR, and IRU algorithms in the *worker*. The node first buffers chunks on each node and when the buffer is full, the node will call the CSR or ATR algorithms to relocate the chunks into a new location and persist them in Ceph. When users update data, the node will call the IRU scheme to reduce the extra fragments caused by updates.

**Storage backend.** We use the *Ceph RBD device* [27], [24], which is a popular, scalable and reliable clustered storage service, to act as the storage backend of ReoDedup, such that we just view the ceph RBD device as a large and sequential storage backend.

**Metadata management.** The metadata is consisted of two types, the first one is a file contains which segments and the location of these segments, and the second one is about which chunks a segment contains and the location of these chunks. To ensure load balance, we also use the hash function to map each metadata to a specific node which will store the metadata in *Redis* [25], a popular and efficient in-memory data structure store. We organize the metadata in a *decentralized* way instead of using a centralized metadata server to avoid metadata access bottleneck.

**Buffer optimization.** We set a write buffer and a read buffer to improve the write and read performance, respectively. Writing segments in small chunks will cause performance degradation, since multiple small chunks will cause many small network I/Os and cannot utilize the locality. We set several local write buffers for each node first [28] and each buffer simulates a specific remote node. We group each chunk in a specific local write buffer rather than route it to a remote node directly. After the segments are finished chunking, we transfer all the chunks within the each write buffer to a specific remote node via *one* network I/O. In this way, we can reduce the multiple network I/Os and utilize the data locality to improve the write performance. We also set a read buffer on each node to cache

the duplicated chunks for higher read performance when the workload is highly duplicated.

## B. Experimental Setup

**Testbeds.** We conduct our experiments via Alibaba Cloud [29] with 8-24 *ecs.n1.large* instances that are equipped with an Intel Xeon E5-2680v3 CPU (4 cores) and an 8GiB RAM. Each instance has a 40GiB SSD installed with Ubuntu 20.04 and two 1TiB SATA HDDs to store the data. All instances are connected via a 10Gbps network. Our evaluation is based on the Ceph block storage service and each node runs 2 OSD daemons.

**Configurations.** We vary some configuration parameters to observe their impacts on performance. We set the node number to 8 by default and range it from 8 to 24 [1], [3] to find the impact of node number. To maintain the spatial locality on disks, we read 1024 consecutive chunks on disk in one read I/O by default. We observe the impact of chunk size by ranging it from 4KB to 32KB. We report each result on average over 5 runs.

**Evaluation workloads.** Our evaluation workloads are artificially formed by the following two tools:

- (i) **FIO** [19] is a tool to generate specified type of I/O action. We use FIO to form workloads that have the specific deduplication ratio. We generate FIO workloads with three different deduplication ratio: 1.5 (named as *FIO\_15*), 2 (named as *FIO\_20*), and 3.3 (named as *FIO\_33*). Some related studies [2] also use FIO to conduct their experiments;
- (ii) **DEDISbench** [30] is a popular tool to simulate the real-world workload, which can extract the content distribution from the real-world datasets and then synthesizes workloads that follow the percentage and distribution of the duplicates existing in real-world storage systems. We use two default primary storage distributions that come with DEDISbench: *dist\_personalfiles* (extracted from a personal files storage) and *dist\_highperf* (extracted from a high performance storage). We generate workloads based on the above two distributions and named as *DEDIS\_PF*, *DEDIS\_HP*, respectively. The chunk size are set to 4KiB. DEDISbench has been also adopted in recent deduplication studies [1], [31].

**Comparison schemes.** We mainly use one baseline scheme and two state-of-the-art schemes as our comparison schemes.

- (i) **Baseline.** We choose the scheme without any optimization as our comparison baseline. We call it *Base*.
- (ii) **Rewriting scheme** (see §II-D) [7], [8]. The rewriting scheme rewrites the chunks of each file to make them sequential. We call it *Rewrite*.
- (iii) **Superchunking scheme** (see §II-D) [5]. The superchunking scheme splits files into small chunks and groups a collective of consecutive chunks into a larger superchunk. We call it *SC*. We set the superchunk size as 1MB by default, like [5].
- (iv) **ReoDedup-C (our scheme).** We call ReoDedup based on the CSR algorithm ReoDedup-C.
- (v) **ReoDedup-A (our scheme).** We call ReoDedup based on the ATR algorithm ReoDedup-A.

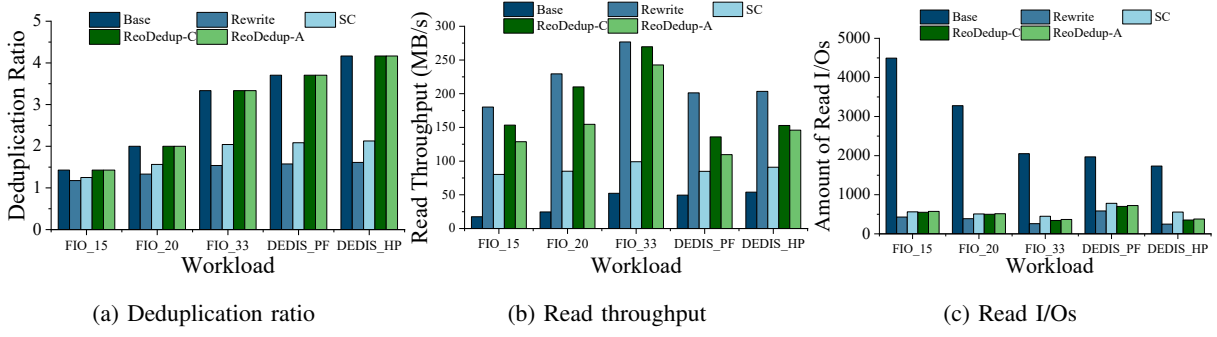


Fig. 10. Experiments 1-2: Read throughput and the amount of read I/Os.

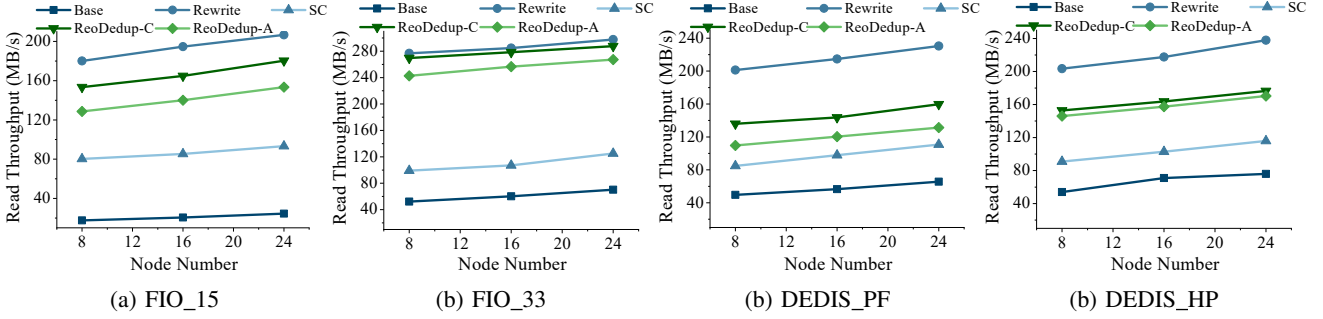


Fig. 11. Experiment 3: Scalability.

### C. Experiment results

**Experiment 1 (Deduplication ratio).** We measure the deduplication ratio of *Base*, *Rewrite*, *SC*, *ReoDedup-C* and *ReoDedup-A* under different workloads. Figure 10(a) shows that the deduplication ratios of *ReoDedup-C* and *ReoDedup-A* are the same as that of *Base*, which means *ReoDedup* has no deduplication ratio loss. The reason is that relocating neither adds additional storage redundancy nor lowers the deduplication ratio. We also see that the deduplication ratios of *ReoDedup-C* and *ReoDedup-A* are much higher than those of *Rewrite* and *SC*. The reason is that *Rewrite* adds lots of additional redundancy and *SC* cannot route chunks that can be deduplicated into identical nodes for performing deduplication if they are in different superchunks (see §III-A). Specifically, compared to *Rewrite*, *ReoDedup-C* and *ReoDedup-A* can increase the deduplication ratio by up to 158.3% (in *DEDIS\_HP*), and compared to *SC*, *ReoDedup-C* and *ReoDedup-A* can increase the deduplication ratio by up to 95.8% (in *DEDIS\_HP*).

**Experiment 2 (Read throughput and read I/Os).** We measure the average read throughput of *Base*, *Rewrite*, *SC*, *ReoDedup-C* and *ReoDedup-A* under different workloads, respectively. Figure 10(b) shows the average read throughput of different schemes. Compared to *Base*, *ReoDedup-C* and *ReoDedup-A* can significantly increase the average read throughput by up to  $7.75\times$  (in *FIO\_15*) and  $6.34\times$  (in *FIO\_15*), respectively. The reason is that *Base* has no optimization for reducing fragmentation, thus leading to a low read throughput. Compared to *SC*, *ReoDedup-C* and *ReoDedup-A* can increase the average read throughput by up to  $1.72\times$  (in *FIO\_33*) and  $1.45\times$  (in *FIO\_33*), respec-

tively. The reason why *ReoDedup* outperforms *SC* is that superchunking can only ensure the chunks within a superchunk not to be fragmented, but different superchunks are still fragmented between each other, while *ReoDedup* can relocate more fragmented chunks and place them consecutively, thus having a higher read throughput. We see that the average read throughput of *Rewrite* is higher than both that of *ReoDedup-C* and *ReoDedup-A*. The reason is that *Rewrite* can minimize the fragmentation via rewriting all fragmented chunks consecutively, while *ReoDedup-C* and *ReoDedup-A* only rewrite most of the fragmented chunks based on the greedy algorithm (*ReoDedup-C*) and heuristic (*ReoDedup-A*). However, rewriting schemes will incur significant additional redundancy thus leading to a low deduplication ratio (see Experiment 1), while *ReoDedup-C* and *ReoDedup-A* maintain the same deduplication ratio as *Base*. We can also see that the read throughput of *ReoDedup-C* is higher than that of *ReoDedup-A*. The reason is that *ReoDedup-A* only relocates chunks that simultaneously exist in single file, while *ReoDedup-C* can relocate chunks that simultaneously exist in multiple files and thus reduces more fragmentation.

We also measure the average amount of read I/Os when we read the same amount of data via different schemes under different workloads. Figure 10(c) shows that *ReoDedup-C* and *ReoDedup-A* can greatly reduce the average amount of read I/Os compare to *Base*. Specifically, compared to *Base*, *ReoDedup-C* and *ReoDedup-A* can greatly decrease the average amount of read I/Os by up to 87.7% (in *FIO\_15*) and 87.2% (in *FIO\_15*), respectively. Compared to *SC*, *ReoDedup-C* and *ReoDedup-A* can decrease the average amount of read I/Os by up to 36.4% (in *DEDIS\_HP*) and 32.1% (in *DEDIS\_HP*), respectively. Com-

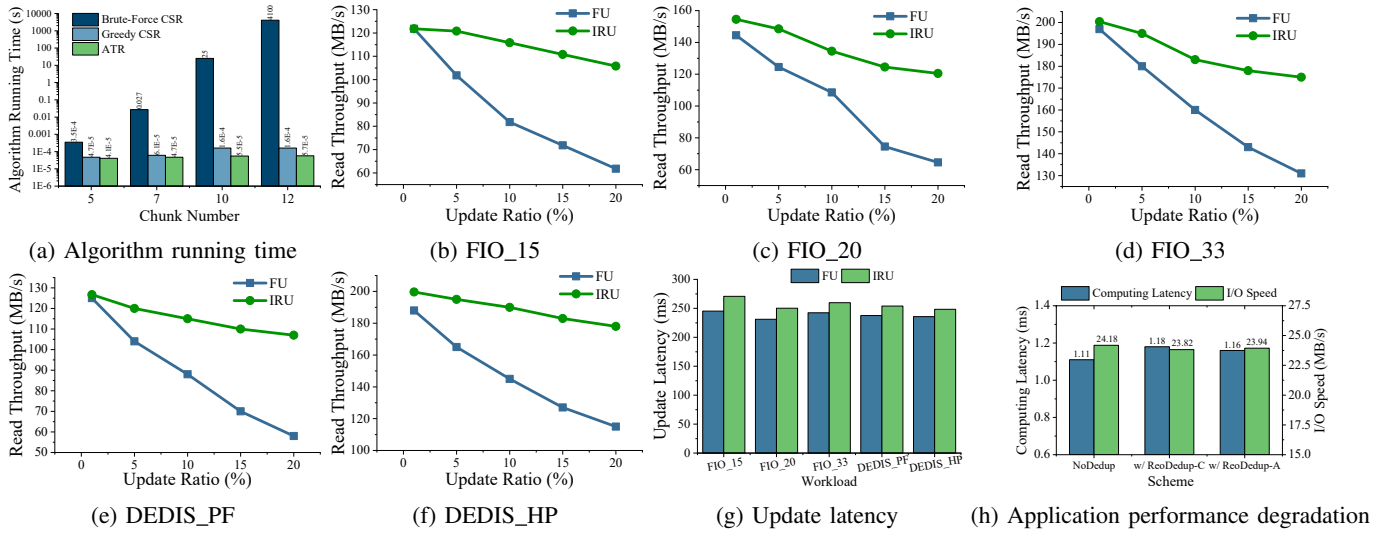


Fig. 12. Experiments 4-7: Algorithm running time, improvement of IRU, update latency, and application performance degradation.

pared to *Rewrite*, ReoDedup-C and ReoDedup-A have relatively higher amount of read I/Os. We also see that the read throughput increases with the deduplication ratio. The reason is that we set a read buffer (see §V-A) for each node to cache the duplicated chunks, such that the more duplicates in the workload, the more chunks will be read in the cache, thus increasing read throughput.

**Experiment 3 (Scalability).** We measure the average read throughput of *Base*, *Rewrite*, *SC*, ReoDedup-C and ReoDedup-A under different workloads versus node number to evaluate the scalability of ReoDedup. Figure 11 show that all schemes' average read throughputs increase with the node number. We see that when ranging the node number from 8 to 24, the average read throughput of *Base*, *Rewrite*, *SC*, ReoDedup-C and ReoDedup-A can be increased by up to 44.6%, 16.9%, 30.7%, 17.6%, and 19.8%, respectively. The reason is that the increasing node number will add more resources to the system, such that the read parallelism can be further improved. For all different node numbers, ReoDedup always outperforms *Base* and *SC*.

**Experiment 4 (Algorithm running time).** We measure the algorithm running time of brute-force CSR, greedy CSR and ATR. Since the time complexity of brute-force CSR is too high, we just vary the chunk number from 5 to 12 to see how much greedy CSR and ATR can reduce the time complexity compared to brute-force CSR. Figure 12(a) shows that greedy CSR and ATR can significantly reduce the algorithm running time of brute-force CSR. We see that when chunk number is just 5, the algorithm running time of brute-force CSR is just a little higher than that of greedy CSR and ATR. But when chunk number getting larger, the algorithm running time of brute-force CSR increases drastically, while the algorithm running time of greedy CSR and ATR are still very low. We also see that the algorithm running time of ATR is lower than that of greedy CSR. Specifically, ATR can decrease the algorithm running time by up to 65.6% (when chunk number is 10).

**Experiment 5 (IRU improvement).** We measure the average read throughput by fragmented update (FU) and index-remapping update (IRU), after updating a certain ratio of files (e.g., half of files are updated means the ratio is 50%). Figure 12(b)-(f) show that as the update ratio increases, the average read throughput of IRU is always higher than that of FU. For example, IRU increases the average read throughput by up to 86.8% (i.e., when update ratio is 20% in FIO\_20). The reason is that FU will introduce fragmented updates while IRU uses remapping to avoid this kind of fragments. The average read throughput of both FU and IRU decrease as the update ratio increases. For FU, the reason is that the fragmented introduced by FU increases as the update ratio increases. For IRU, the reason is that remapping indices will impair the read performance due to additional I/Os to the new nodes (see §IV-C). Therefore, the high the update ratio is, the more chunks have to be read via remapping indices, thus decreasing the read throughput.

**Experiment 6 (Update latency).** We measure the update latency of fragmented update (FU) and index-remapping update (IRU). Figure 12(g) shows that the latency of IRU is slightly higher than FU. The reason is that IRU adds a few steps (such as inserting remapping indices in the new node). However, the extra latency introduced by IRU is acceptable, which is only 7.6% higher on average.

**Experiment 7 (Application performance degradation).** We run some applications (including matrix computing and disk I/O) alongside ReoDedup-C and ReoDedup-A and measure the application performance degradation due to the deduplication. We also measure the application performance without ReoDedup-C and ReoDedup-A as the comparison baseline (called *NoDedup*). Figure 12(h) shows the computing latency of a matrix-matrix multiplication computing task (one matrix's shape is  $100 \times 10$  and the other one's shape is  $10 \times 4$ ). We can see that the computing latency increases slightly when the computing process runs alongside the deduplication (no more than 6%). Figure 12(h) also shows the I/O speed. We can see

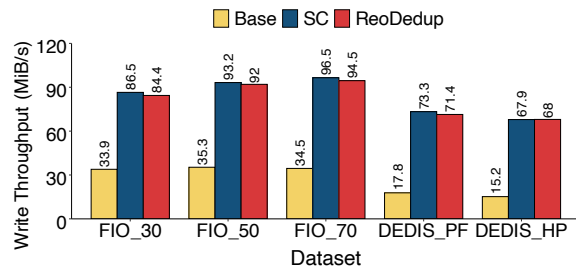


Fig. 13. Experiment 8: Write throughput.

that the I/O speed also decreases slightly when the I/O task runs alongside the deduplication (no more than 1.5%).

**Experiment 8 (Write throughput).** We measure the write throughput of Base, SC, and ReoDedup. Figure 13 shows that ReoDedup has a significant higher write throughput over Base scheme, and has a similar write throughput with SC scheme. This is because we use a write buffer to batch incoming small chunks and send them as a single, large sequential network I/O, improving efficiency. This is similar to SC, where each transmission also sends a single large superchunk to boost write performance. In contrast, the Base scheme sends only one small chunk per transmission, resulting in many small network I/Os and thus the lowest efficiency.

## VI. DISCUSSION AND FUTURE WORK

**SSD Optimization.** ReoDedup rearranges fragmented chunks into a contiguous layout to improve read performance, thereby significantly enhancing read throughput and reducing read I/Os in HDD environments. When deployed in an all-SSD environment, however, these design assumptions need to be revisited: SSDs have no mechanical seek cost but are more affected by write amplification and limited erase endurance. We suggest extending the relocation cost model to account for both read benefits and write costs, including write amplification and device durability. The algorithm should incorporate relocation granularity aligned to erase-block sizes, placement strategies that are friendly to device parallelism (channels or namespaces), wear-aware scheduling, and deferred execution during low-load periods. Future work should evaluate read throughput, write amplification, physical bytes written, and device lifetime on a full-SSD testbed to verify the trade-off between performance and durability in SSD-aware ReoDedup.

**Topology Awareness.** The design and evaluation of ReoDedup primarily assume a balanced, large-scale cluster where nodes are evenly distributed and network topology effects are amortized. In practice, deployments can be in multi-rack clusters with varied placement policies, where same-rack placement can reduce cross-rack latency and bandwidth contention, yet such affinity is not always guaranteed in real-world cloud environments. To address these challenges, future work should integrate topology awareness into the scheduling and relocation decisions of ReoDedup, incorporating metrics such as node degree, link bandwidth, and rack affinity into

the cost model. Evaluations should cover diverse topologies, from minimal-node to multi-rack scenarios, to understand how the design adapts across deployment scales and to refine strategies for maintaining consistent performance regardless of topology.

## VII. CONCLUSION

A new fragmentation problem arises in clustered primary storage when performing deduplication, while existing schemes for traditional fragmentation in backup storage cannot address the new fragmentation effectively. We propose a relocating-based deduplication mechanism, ReoDedup, which smartly places fragmented deduplicated chunks consecutively to reduce fragmentation significantly without compromising the deduplication ratio. We propose for ReoDedup two algorithms that aim to optimize the relocating and lower the time complexity, respectively. We further propose a remapping algorithm to further enhance ReoDedup's relocating and read performance. We implement ReoDedup atop Ceph and our experiments via Alibaba Cloud demonstrate the efficiency of ReoDedup in deduplication ratio and read performance.

## ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (No. 62272185), Shenzhen Science and Technology Program (No. JCYJ20220530161006015) and Key Laboratory of Information Storage System Ministry of Education of China. The corresponding authors are Yuchong Hu and Leihua Qin.

## REFERENCES

- [1] J. Paulo and J. Pereira, "Efficient deduplication in a distributed primary storage infrastructure," *ACM Trans. on Storage*, vol. 12, no. 4, pp. 1–35, 2016.
- [2] M. Oh, S. Park, J. Yoon, S. Kim, K.-w. Lee, S. Weil, H. Y. Yeom, and M. Jung, "Design of global data deduplication for a scale-out distributed storage system," in *Proc. of IEEE ICDCS*. IEEE, 2018.
- [3] M. Oh, S. Lee, S. Just, Y. J. Yu, D.-H. Bae, S. Weil, S. Cho, and H. Y. Yeom, "TiDedup: A new distributed deduplication architecture for ceph," in *Proc. of USENIX ATC*, 2023.
- [4] 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 *Proc. of USENIX ATC*, 2016.
- [5] W. Dong, F. Douglass, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," in *Proc. of USENIX FAST*, 2011.
- [6] 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 *Proc. of USENIX ATC*, 2011.
- [7] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: latency-aware, inline data deduplication for primary storage," in *Proc. of USENIX FAST*, 2012.
- [8] X. Zou, J. Yuan, P. Shilane, W. Xia, H. Zhang, and X. Wang, "The dilemma between deduplication and locality: Can both be achieved?" in *Proc. of USENIX FAST*, 2021, pp. 171–185.
- [9] Y.-K. Li, M. Xu, C.-H. Ng, and P. P. Lee, "Efficient hybrid inline and out-of-line deduplication for backup storage," *ACM Trans. on Storage*, vol. 11, no. 1, pp. 1–21, 2014.
- [10] 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 *Proc. of USENIX FAST*, 2015.

- [11] A. Levi, P. Shilane, S. Sheinvald, and G. Yadgar, "Physical vs. logical indexing with IDEA: Inverted Deduplication-Aware index," in *Proc. of USENIX FAST*, 2024.
- [12] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Computing Surveys*, vol. 47, no. 1, pp. 1–30, 2014.
- [13] R. Jones and R. Lins, *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.
- [14] F. Ni and S. Jiang, "RapidCDC: Leveraging duplicate locality to accelerate chunking in CDC-based deduplication systems," in *Proc. of the ACM SoCC*, 2019.
- [15] F. Guo and P. Efsthopoulos, "Building a high-performance deduplication system," in *Proc. of USENIX ATC*, 2011.
- [16] D. Meister, J. Kaiser, and A. Brinkmann, "Block locality caching for data deduplication," in *Proc. of the International Systems and Storage Conference*, 2013.
- [17] "Lustre storage white paper," <https://www.lustre.org/wp-content/uploads/architecting-lustre-storage-white-paper.pdf>, Aug. 2023.
- [18] B. Mao, H. Jiang, S. Wu, and L. Tian, "Leveraging data deduplication to improve the performance of primary storage systems in the cloud," *IEEE Trans. on Computers*, vol. 65, no. 6, pp. 1775–1788, 2015.
- [19] "FIO," <https://github.com/axboe/fio>.
- [20] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," *ACM Trans. on Storage*, vol. 6, no. 3, pp. 1–26, 2010.
- [21] "FIU I/O deduplication," <http://iota.snia.org/traces/block-io/391>.
- [22] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.
- [23] J. E. Hopcroft, J. D. Ullman, and A. V. Aho, *Data structures and algorithms*. Addison-wesley Boston, MA, USA:, 1983, vol. 175.
- [24] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *Proc. of ACM/IEEE conference on Supercomputing*, 2006.
- [25] "Redis," <https://redis.io/>, Aug. 2024.
- [26] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proc. of IEEE INFOCOM*. IEEE, 2015.
- [27] "Ceph rbd," <https://docs.ceph.com/en/latest/rbd/>, Aug. 2024.
- [28] Z. J. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "Cluster and single-node analysis of long-term deduplication patterns," *Trans. on Storage*, vol. 14, no. 2, pp. 1–27, 2018.
- [29] "Alibaba Cloud," <https://www.alibabacloud.com/product/ecs-pricing-list/en>.
- [30] "Dedisbench," <https://github.com/jtpaulo/dedisbench>.
- [31] Q. Yang, R. Jin, and M. Zhao, "SmartDedup: Optimizing deduplication for resource-constrained devices," in *Proc. of USENIX ATC*, Jul. 2019.