A Focused Garbage Collection Approach for Primary Deduplicated Storage with Low Memory Overhead

Jingsong Yuan†, Xiangyu Zou†, Han Xu†, Zhichao Cao‡, Shiyi Li†, Wen Xia†*, Peng Wang §, Li Chen §
† Harbin Institute of Technology, Shenzhen ‡ Arizona State University § Huawei Technology Co., Ltd.
* Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies
{js.yuann, crischenli, xuhan603275}@gmail.com, {lishiyi, xiawen}@hit.edu.cn, {wangpeng423}@huawei.com
{xiangyu.zou}@hotmail.com, {Zhichao.Cao}@asu.com

感觉现在Odess的GC也应该差不多用的这个Mark and sweep
大概就是提出了一种加速mark and sweep的mark阶段的方法

***Abstract*—Since one chunk could be shared by many files after data deduplication, Garbage Collection (GC) is an essential but complex task to reclaim stale chunks in large-scale primary deduplication systems. Traditional Mark&Sweep is a widely used approach but suffers from the increasingly traversing time and huge memory overhead of Liveness Array (i.e., a data structure reflects the liveness of alive chunks) in the Mark phase. This paper proposes a new method named Focused Garbage Collection (FGC) to accelerate the Mark phase for primary deduplication storage significantly. Specifically, we design a global Austere Reference Graph with low memory cost that efficiently represents files' reference relationships (i.e., sharing chunks after deduplication) by considering the deduplication characteristics of workloads in primary systems. Austere Reference Graph helps FGC focus on the deleted files and their correlative files to quickly mark stale chunks, while traditional approaches need to traverse all files. Consequently, FGC's traversing time and Liveness Array size will be greatly reduced in the Mark phase. Evaluation results show that compared with traditional Mark&Sweep, FGC decreases the time consumption in the Mark phase $1.3\times$-$7.34\times$ in a stand-alone primary deduplication system and $128\times$-$256\times$ network traffic reduction for the Mark phase while only introducing $< 0.05\%$ extra memory overhead for the reference graph.**

***Index Terms*—Deduplication, Garbage Collection, Overhead**

I. Introduction

Data deduplication, which allows files to share their "common" (i.e., duplicated) data in chunk-level [1], has been used in many use cases, including backup storage [2], [3], data syncing [4], and flash storage [5].

Primary storage is another use case that could benefit from deduplication. However, applying deduplication in this scenario is limited by its complicated and time-consuming Garbage Collection (GC) workflow, which plays a vital role in deduplication to reclaim space. The efficiency of GC suffers from the chunks sharing in deduplication. Specifically, duplicated chunks will be physically stored only once, and these "common" chunks in different files will be referenced many times. With the storage data evolution (i.e., caused by file operations), the deduplicated systems are full of active chunks (i.e., still referenced by some files) and stale chunks (i.e., no longer referenced by any file). And GC workflow has to recognize and remove stale chunks by tracking references of all chunks. Considering the number of chunks can be at billion-level in primary storage, **GC will be a heavy task (very time- and memory-consuming), and it may lead to system stalls or worse latency, which are critical issues for primary storage.** Therefore, an efficient GC approach is necessary for primary deduplicated storage with frequent file delete and update operations.

Generally, there are two kinds of GC approaches for identifying stale chunks: *Reference Count* [3], [6] and *Mark & Sweep* [3], [7]. *Reference Count* [3], [8], which records how many times each chunk is referenced, is straightforward to implement but encounters realistic performance issues in large-scale deduplication, due to requiring additional transactions to guarantee count updates and correctness [9] and gating the critical writing performance in a deduplication system. *Mark & Sweep* is a strategy that first recognizing alive chunks with a Liveness Array and then copy-forward these chunks to merge them together. However, this process involves all logical files and causes unacceptable calculation overhead and memory overhead, and may lead to system stalls, which is another critical issue to primary storage systems.

A better GC approach is desperately needed to satisfy critical concerns of deduplication on primary storage. Based on our observations, we found GC could focus on a much smaller range of files to find potential stale chunks for primary deduplication quickly. The two observations are as follows:

① Only deleted files and other *correlative files* (i.e., the files that share common chunks with deleted files) will be affected by file deletion. Removing all deleted file data will damage the integrity of alive correlative files. Thus, the chunks in the deleted files could be classified into two types, and they should be processed differently: *shared chunks*, which are shared with deleted files and other correlative files, should be kept in storage; *non-shared chunks*, which only exist in the deleted file, are the reclaim targets during GC. Thus **the objective of GC is to focus on deleted files and their correlative files to identify the shared chunks and non-shared chunks.** To speed up accessing correlative files, we could record all the files' correlative relationships in a graph named *RefGraph*. However, saving all correlative relationships needs huge memory overhead to track all chunks and referenced files.

② A shared chunk could be determined by comparing deleted files with a base correlative file (i.e., the first owner of that chunk) instead of all correlative files. This suggests that we can design an *Austere RefGraph*, a simplified sub-graph of

确实，要保证并发安全的

介绍了什么是去重

感觉这篇文章的优化思想也体现在了Odess中

这里说的两个分类在代码中也是有体现的

感觉这段也可以作为什么“碎片影响GC”的参考

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00053

2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00053

RefGraph to record the representative correlative relationships, which significantly reduces the traversing overhead.

Thus, we propose a new Focused Garbage Collection (FGC) approach to accelerate the Mark phase for primary deduplication significantly. Specifically, when new files are being deduplicated, FGC keeps the relationships of files and their base correlative files in a global Austere Reference Graph, which facilitates acquiring correlative files directly. Instead of accessing all files in the traditional method, FGC only focuses on the deleted files and their base correlative files to build the Liveness Array and mark the stale chunks. Therefore, by reducing the scope of traversing, the traversing time and Liveness Array memory overhead will also be significantly reduced and become proportional to the size of deleted files. Generally, the contributions of this paper are three folds:

- We propose a new chunk classification that helps to distinguish stale chunks in the deleted files. By using the new classification methodology, FGC only focuses on traversing the deleted files and their correlative files, ignoring all the other irrelevant files.
- We propose an Austere Reference Graph to only record the base correlative relationship for each file, which still works to get the deleted files' correlative files.
- Evaluation results suggest that FGC reduces about 1.3×-7.34× time consumption of Mark phase in a stand-alone system and 128×-256× network traffic in Ceph's distributed deduplication framework [10], [11]. Meanwhile, the extra memory overhead ratio of our austere graph in FGC is negligible ($< 0.05\%$) due to only recording representative correlative relationships in Austere RefGraph.

II. Related Work

A. Garbage Collection in Deduplication

很好，这个部分应该可以大看特看

Deduplication [12]–[17] is a widely used technique for efficiently saving storage costs by splitting data sources (e.g., different files) into chunks and sharing the common chunks. File recipe keeps the chunk order with fingerprint (i.e., the cryptographical secure hash of chunk).

这个部分主要讲的就是GC在去重系统的作用。应该可以看看这段的思路学习一下

However, the complex ownership of physical chunks makes it difficult to determine the chunk liveness due to sharing chunks. Specifically, a physical chunk could belong to many files in deduplication systems, instead of only one in non-deduplication systems. Hence, when reclaiming chunks for releasing storage space, deleting any physical chunk must be very careful in case of causing data corruption in other files. It means we can safely remove a physical chunk only when it is no longer referenced by any stored file in a dedup system, while a non-dedup system could directly update or delete a particular file. Since all valid files will be involved in checking a specific chunk's liveness, GC is nearly a **global-scale** work. Therefore, GC is a critical but heavy task.

The global-scale GC performance becomes more severe when running in a distributed system. Previous works [18], [19] suggest the network traffic becomes a major overhead in the distributed system (such as Ceph [10]). Specifically, GC requires globally checking chunk's liveness across nodes and causes heavy network traffic with the data growth. Therefore, reducing GC's network traffic among nodes is the main concern in the distributed system.

TABLE I
Typical GC approaches in different deduplication systems.

Name	Taxonomy	Mode	Scale-out
iDedup [20]	Reference Count	Inline	✗
CMA [21]	Reference Count	Inline	✗
GMS [3]	Mark & Sweep	Inline	✓
LGC/PGC [7]	Mark & Sweep	Offline	✓

Thus, GC in deduplication is usually time-consuming but essential, and improvements on GC have been of much interest to researchers and users.

B. Classification of Garbage Collection Techniques

Several previous works about GC for deduplication frameworks are shown in Table I, and the typical GC approach can be classified into two types.

Reference Count [20], [21] tracks the precise references of all chunks in the system. However, counting all references meets performance degradation and correct value guarantee problems as the data scale grows. To guarantee the correctness of reference count, it needs an extra transaction to roll back the counter value when an error occurs. There are some variants based on Reference Count. iDedup [20] is the first such framework used in the primary storage. It proposes counting every physical block to accelerate the remove operation. To simplify the managed data granularity, Container-Marker Algorithm (CMA) [21] maintains a counting reference of container (i.e., a physical block to store the fixed size of chunks) instead of all chunks. It is designed for backup storage and unsuitable in primary storage since it deletes an entire container completely based on out-of-date backup data.

Mark & Sweep has two phases: 1) analyses the liveness of all chunks, 2) and removes stale chunks in different phases. In the first Mark phase, it traverses all files recipes to build a *Liveness Array* (e.g., Bloom Filter [22], Perfect Hash Vector [22], Bitmap [3]), which represents the existence of chunks in the system. And then, it accesses physical containers to remove stale chunks one by one in Sweep phase. There are multiple developed Mark&Sweep approaches to accelerate Mark phase, as shown in Table I. GMS [3] groups different backup session in the dedup phase and tracks changed backup groups. It only marks the changed backup groups and ignores the unchanged backup groups. However, there is no explicit backup session in primary storage, meaning GMS is unsuitable in primary scenario. LGC/PGC [7] changes the traversing order to scan the storage containers sequentially. Similarly, a complete traversing of all the existing files is necessary to mark all stale chunks, which is still time-consuming.

Although various GC approaches exist, they meet different problems in large-scale primary storage, and we will discuss them next.

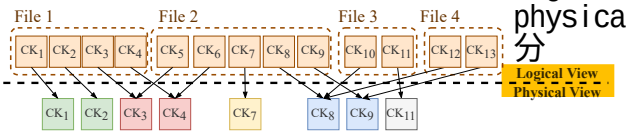
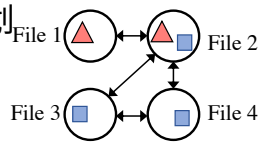
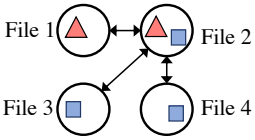


Fig. 1. An example of two categories of chunks when GC runs after deleting *File 1*. Green-shaded chunks are **Non-shared** Chunks; Red-shaded chunks are **Shared** Chunks.

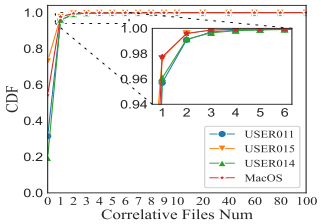


(a) The complete correlative graph.

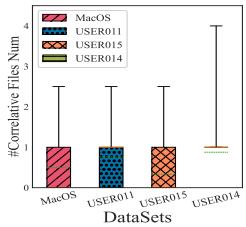


(b) The simplified correlative graph.

Fig. 2. The comparison between the complete and simplified correlative graph for the four files shown in Fig. 1.



(a) CDF Plot



(b) Boxplot

Fig. 3. The statistics of CDF and Boxplot about base correlative file number under four different Datasets.

III. Observation and Motivation

A. Problems in Existing Approaches

Compared with backup storage, primary storage systems are more performance- and latency-sensitive [13], [23]. Thus primary storage requires a more efficient GC approach after deduplication to (1) quickly recognize and clear invalid data, and (2) cause fewer impacts on the system's I/O performance as much as possible. However, existing GC approaches could not satisfy both requirements well at the same time.

As discussed in §II-B, **Reference Count** suffers from performance degradation and complicated count value guarantee transactions. It is because Reference Count tries to identify stale chunks in real-time, and thus these issues could be alleviated but can not be avoided totally.

Mark&Sweep is another approach that only concerns whether a chunk is invalid but never reflects the referenced degree of a chunk. The main idea of Mark&Sweep is explained in §II-B, which detects and frees the stale chunks through two passes. Thus, Mark&Sweep could avoid performance degradation of updating counts and complicated correct reference counts guarantee problems. However, when applying it to the primary storage, it brings in new problems. The first problem is severe latency caused by reading all file recipes in the Mark phase. With the increasing number of files in the primary storage, the size of the total file recipes also grows fast. Scanning all these file recipes in Mark phase is time-consuming. Also, the file recipes may change during the scanning time period due to new updates or deletion. The issue becomes more complicated.

Another problem is the memory cost of Liveness Array. A common choice of Liveness Array is Bloom Filter, which balances between querying performance and storage overhead. However, it still occupies a large memory to store billions of chunks. Assuming a system stores 1PB logical data, 8KB avg. chunk size, and 30% deduplication ratio, it needs a 214GB Bloom Filter to keep billions of chunks with 0.01% false–positive rate. If the size of Bloom Filter exceeds memory capacity, it brings extra disk I/Os to update Bloom Filters [24].

Thus, when applying Mark & Sweep in the primary storage, **scanning latency time** and **memory overhead** problems should be addressed.

B. Focused Mark Strategy

Considering sweeping stale chunks in disk is a similar process for different Mark & Sweep variants and all file recipes will be traversed in Mark phase, we focus on reducing the traversing scope based on our following observations.

Observation 1: When running GC after deleting a file F_k, we could focus on the deleted file F_k and other files that share chunks with F_k to find the stale chunks. Besides, we build a graph structure to pin-point the sharing-chunks files quickly.

When deleting a file F_k, all chunks in the deleted files can be classified into two categories as follows:

- *Non-shared Chunks*: only referenced by F_k.
- *Shared Chunks*: referenced by F_k and other alive files.

Fig. 1 provides an example of two categories of chunks after deleting *File 1*. Among these categories, ① Non-shared Chunks(i.e., CK_1, CK_2) could be reclaimed directly since deleting them will not damage the integrity of other files; ② Shared Chunks (i.e., CK_3, CK_4) cannot be removed in case of causing other files' data loss. Thus, to clear the stale chunk means to find the Non-shared chunks in the deleted files.

Based on the data-sharing mechanism in deduplication storage, we consider the files as *Correlative Files* if they share at least one common chunk and they have a *Correlative Relationship* with each other. By comparing correlative files with deleted files, we could find non-shared chunks easily. Moreover, to accelerate obtaining correlative files, we save all correlative relationships in a graph named *RefGraph*. RefGraph takes the file as the vertex and the bi-directed correlative relationship as the edge. A complete RefGraph is shown as Fig. 2(a). The pink-triangle chunk and the blue-rectangle chunk bring correlative relationships for these files.

This indicates that the original GC problem is finding stale chunks among all the stored data and now GC problem becomes finding non-shared chunks in deleted files by concentrating on correlative files through RefGraph. Thus, GC complexity is simplified from **global-scale** to **local-scale**.

However, recording all relationships among stored files based on their chunk referencing situations in RefGraph causes a large memory overhead. In the worst case when all files share common chunks, all files become correlative files with each other, and now the RefGraph is a complete graph. Storing a complete graph with 1 million files takes up to $10^6 \times 10^6 \times 8bytes = 7.9TB$ memory. Thus, we need to find an effective graph structure to save memory.

意思就是对于每个chunk，实际上要判断它是不是shared的只需要再多一个文件就行了，不用非得全图
所以关键就是选择哪个文件反映是不是shared
这点确实6666，不知道怎么实现的

Observation 2: Because only one and not all alive correlative files are needed to determine the liveness of a shared chunk, the RefGraph could be simplified to only record the representative correlative relationships to greatly reduce the graph size (i.e., aforementioned memory overhead).

Choosing one representative file for a chunk to reflect its liveness is important and considerable. Specifically, for a chunk, we consider its earliest and still alive owner (i.e., the first file containing that chunk) as the representative *base file* and the other later owners as the *leaf files*. Since the earliest file can record all relationships among later files, it does not require an extra process to update the relationship if choosing the new file as the base file. We only record the relationship between the base file and leaf file in an incomplete RefGraph, named *Austere RefGraph*, which is a sub-graph of RefGraph with the same vertices. Note that ① for different chunks, a file could be a base or leaf file at the same time, but a chunk only has one base file at a time. ② Comparing representative correlative files with deleted files through Austere Graph is enough to identify the non-shared chunks in the deleted files.

并且根据统计信息可知，一个文件一般只有一个base file

Fig. 2(b) shows an example of the simplified correlative graph. It dismisses the correlative relationship between *Files 3* and *4* since *File 3* is not the base file for the chunk CK_8. When removing *File 4*, it is sufficient to compare with *File 2* to determine the logical chunks CK_{12} and CK_{13} as shared chunks in simplified graph (see Fig. 2(b)) while it additionally compares *Files 3* and *4* in complete graph (see Fig. 2(a)).

The statistical data also proves that the simplified graph is small enough. Fig. 3(a) shows the cumulative number of base correlative files (recorded in Austere RefGraph) distribution on four different datasets. A detailed introduction to these datasets will be discussed in §V. Over 95% of files have no more than 1 base correlative file, and over 99% of files have fewer than 10 base correlative files. Fig. 3(b) shows the boxplot of different datasets. It shows that the average number of correlative files (in green line) on different datasets is less than 1, and at least 75% of files only have 1 base correlative file. Since we record the correlative relationships in RefGraph, the size of RefGraph is proportional to the number of files. Therefore, considering the above analytical data and other research [25], [26], we can conduct that the simplified correlative relationship graph takes little memory overhead. Assuming the average correlative files is 10, by using the simplified graph, the memory overhead of RefGraph is reduced from 7.9 TB to $10^6 \times 10 \times 8bytes = 80MB$.

In summary, the above observations **motivate** us to design Focused Garbage Collection (FGC), which focuses on finding correlative files to mark the non-shared chunks in deleted files as stale chunks. Specifically, ①to accelerate accessing correlative files, we record the correlative relationships between files in RefGraph during deduplication Index phase, and the RefGraph can be optimized as Austere RefGraph to save memory overhead; ② when deleting files and running GC, we compare the recipes of deleted files and their correlative files to pick up all non-shared Chunks.

66666，协议这个抽象很帅

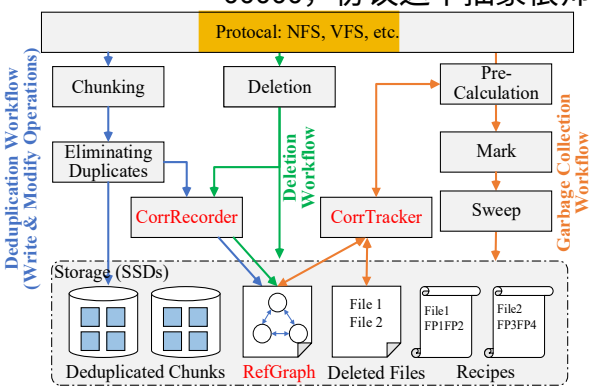


Fig. 4. Architecture overview and processing workflow in the primary deduplication system with FGC. The three components in red are proposed in our approach.

IV. Design and Implementation

A. System Overview

Fig. 4 shows the basic components of a primary deduplication system and its three basic data processing workflows with FGC: deduplication, deletion, and GC.

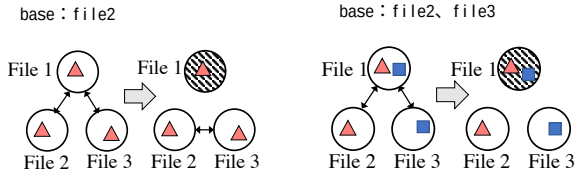
In general, ① when deduplicating a new file (workflow in blue line), FGC uses **CorrRecorder** to record the correlative (deduplication) relationship between the new incoming file and existing files in an in-memory data structure named **RefGraph**; ② when deleting existing files (workflow in green line), FGC uses **CorrTracker** to update RefGraph and record the deleted files in Deleted FileID Sets; ③ when running GC (workflow in orange line), FGC uses **CorrTracker** to focus on all correlative files acquired by Deleted FileID Sets and RefGraph in Pre-Calculation Phase. Then FGC will traverse the correlative files recipes to detect stale chunks in Mark phase and remove marked chunks in Sweep phase. Therefore, FGC filters out irrelevant files in Pre-Calculation Phase and focuses on the deleted files and their correlative files in Mark phase, which greatly reduces the traversing scope for the Mark phase. To support Pre-Calculation Phase, FGC consists of two key modules:

- **CorrRecorder.** CorrRecorder is designed for maintaining correlative (deduplication) relationships in RefGraph. CorrRecorder uses RefGraph to record correlative relationships among files in the system. It will update the relationships when writing or deleting files.
- **CorrTracker.** CorrTracker is designed to provide correlative files for Mark phase to focus on a specific range. CorrTracker works in the Pre-Calculation phase, which uses RefGraph to find out all correlative files of deleted files (saved in Deleted FileID Sets).

In the following subsections, we will introduce RefGraph and the workflow of these two modules in detail.

B. Austere Reference Graph

In FGC, the duty of Austere Reference Graph (short for *RefGraph*) is to maintain base correlative relationships in memory, which helps to obtain base correlative files of deleted files quickly. A file could be considered as a vertex in RefGraph, and the correlative relationship between two files could be considered as an edge. When writing a new file into



(a) Deletion case 1 (b) Deletion case 2

Fig. 5. Two different deletion cases when deleting base file in the RefGraph.

the primary system, RefGraph connects a new edge between the new file vertex and an existing base file vertex. When removing an existing file, RefGraph maintains the evolution of the edge. RefGraph is kept in the memory for quickly accessing, but it will be flushed into the disk periodically.

删除时对图上锁，添加时视为无序的异步命令流

Since data write and deletion is frequent in primary storage, RefGraph uses a simple but efficient mechanism to prevent data loss. When writing new data, RefGraph will build new relationship edges and consider adding new edges as sequences of append command, ignoring the adding relationship order. When deleting existing data, RefGraph will remove the existing relationship edges. It should lock the deleted file and their correlative files in case of data relationships loss.

As shown in Fig. 1, *Files 1* and *2* share common chunks CK_3 and CK_4; *Files 2* and *3* share a common chunk CK_8; *Files 2* and *4* share common chunks CK_8 and CK_9. As Fig. 2(b) shown, we will record *File 2* in the bucket of *File 1*, *Files 1*, *3*, and *4* in the bucket of *File 2*, *File 2* in the bucket of *File 3* and *File 2* in the bucket of *File 4*. Note that, although *Files 3* and *4* share the common chunk CK_8, we ignore that correlative relationship and only record the correlative relationship between base files and leaf files.

<fileID, vector<fileID>>

In our implementation, we use a HashTable to implement RefGraph, which saves the fileID (an 8byte Integer) as the key and its correlative file IDs as the value. Since the deduplication ratio is usually low in primary storage and the base correlative relationship is small, the size of RefGraph is limited. Assuming the system stores 1TB logical data, the average chunk size is 8KB, the deduplication ratio is 30%, the average file size is 10MB, and the average number of base correlative files of a file is 5. Then there are about $1TB/10MB = 1 \times 10^5$ files in primary system, and every bucket of each file has five correlative file IDs. Thus, the size of austere RefGraph is $(1\times10^5)\times(8+5*8)bytes = 4.8MB$, which is about 0.196% of Reference Count Table (2.45GB) and 0.006% of a complete RefGraph $((1\times10^5)\times(8+((1\times10^5)-1)*8)bytes = 80GB)$.

C. CorrRecorder

The main task of CorrRecorder is to save correlative relationships in RefGraph after deduplication, which helps FGC quickly acquire base and leaf correlatives files of a given file. CorrRecorder participates in three kinds of file operations (i.e., **write, delete, modify files**) and updates RefGraph accordingly.

Write Operations. When writing a new file F_{new}, CorrRecorder will record the correlative relationship of each chunk accordingly. Specifically, as shown in Fig. 4, CorrRecorder works in the "Eliminating Duplicates" step. ① When finding a unique chunk in F_{new}, CorrRecorder will record F_{new} as its base correlative file in the Fingerprint Index. ② When finding a duplicate chunk in F_{new}, CorrRecorder

感觉也是相当于在index的阶段多增加了一个字段了

finds this chunk's base correlative file F_k from Fingerprint Index, and adds a new pair $< F_{new}, F_k >$ into RefGraph.

Delete Operations. Deduplication usually runs logically file deletion, which only deletes corresponding recipe but not immediately identifies and reclaims invalid chunks.

leaf file的删除很简单，直接remove就行了。base file的删除需要慎重，需要遍历它所有的子相关文件进行图重构。

FGC requires additional operations to maintain RefGraph. When removing some existing files in primary storage, RefGraph should remove the existing edges of deleted files with other correlative files. Since RefGraph only records the correlative relationships between base files and leaf files, RefGraph could directly remove the relationships when deleting leaf file while it needs to find a new base file when deleting the original base file. We should note that a file could be a leaf or base file for different chunks. Thus we need to read all correlative files to reconnect correlative relationships in case of losing correlative relationships.

Fig. 5 shows two types of graph evolution in the same RefGraph structure. When deleting the leaf *File 2* or *3*, they take the similar actions to directly remove the existing edge. However, they have different steps when deleting the base *File 1*. In Fig. 5(a), FGC should remove the existing edges with *File 1* and find a new base *File 2* (for the pink triangle chunk) to rebuild new correlative relationship between *Files 2* and *3*. While, in Fig. 5(b), FGC tries to find new base files for the pink-triangle chunk and the blue-rectangle chunk, respectively. FGC compares the recipes of *File 2* and *3* but finds no common chunk. Thus, *File 2* becomes the base file to the pink-triangle chunk, so does *File 3* to the blue-rectangle, and there is no more correlative relationship between them.

第二个情况是两个都成为了base

RefGraph has the same structure but evolves differently in two different scenarios in Fig. 5. It is because RefGraph only records the correlative relationships in file level and ignores binding relationship with the specific chunk. Thus when removing relationships, FGC should begin a small-range reconstruction process over all correlative files by comparing files recipes in case of relationship loss. If a new relationship is detected, FGC will add it in RefGraph. Once finding the new base file and reconnecting the correlative relationships are finished, stale relationships and files can be safely removed.

fig5介绍的是对于删除文件的某个块的范围重建过程。

这里要开始讨论对于整个删除文件的所有块的重建过程

Since updating correlative relationships generates extra storage I/Os, FGC delays updating correlative relationships until running GC. If the new files are connected with the deleted files in RefGraph before actually updating RefGraph, we still add new edges and reconnect new correlative relationships when updating RefGraph.

更新图的工作会推迟到GC做，这也就是为啥标题是GC了。。。

Modify Files. When modifying the file content in the primary storage, it could write new or remove data in an existing file. Writing new content into a file can be handled as **Write Operations**. Similarly, these new correlative relationships are still incomplete. When removing data in the file, it can be handled as **Delete Operations**, in which RefGraph removes the existing relationships and reconnects new relationships. However, considering the whole updating process is time-consuming and modifying operation is frequent, we still keep the old relationship in the system. We can adopt the extra cost in the Mark phase instead of frequently updating file

我们依然选择在GC的时候进行修改操作的图更新

也就是说总结一下，它这个GC需要做的最先是Refgraph的更新，然后就一个是Pre-calculate，一个是Mark，一个是sweep。

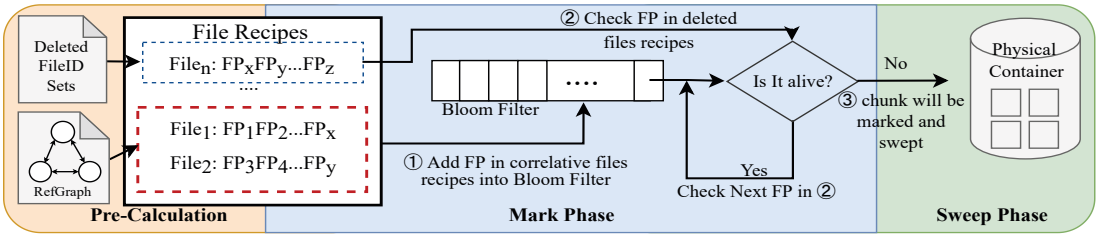


Fig. 6. The Garbage Collection workflow of FGC.

relationships in RefGraph.

D. CorrTracker

CorrTracker is designed for quickly filtering out the irrelevant files according to RefGraph and the Deleted FileID Sets, which helps Mark phase to precisely focus on a specific range with low overheads. In the legacy Mark phase, all file recipes are accessed to determine the liveness of chunks. On the contrary, FGC uses CorrTracker to focus on the correlative and deleted files to detect stale chunks.

The typical GC workflow in FGC is shown as Fig. 6. ① When beginning GC, CorrTracker accesses Deleted FileID Sets and RefGraph to find out all correlative files of deleted files and filter out other irrelevant files in Pre-Calculation phase. ② FGC will read all correlative files fingerprints in recipes from disk and insert them into Liveness Array in Mark phase. We use Bloom Filter to implement the Liveness Array since it could balance the performance and memory overhead. Other data structures (e.g., Bitmap, Perfect Hash Vector) can also be used as the alternative to Bloom Filter, and FGC also helps to reduce its size. ③ After building the Bloom Filter completely, FGC checks the fingerprint of the deleted files recipes one by one. If the fingerprint is not found in the Bloom Filter, the chunk is determined as a non-shared chunk, and it could be removed from the container.

By using CorrTracker and RefGraph to precisely determine the traversing scope in the Pre-Calculation phase, FGC focuses on identifying non-shared chunks in the correlative files. Since we have saved the correlative relationships in RefGraph, the time complexity of accessing correlative files of one file is $O(1)$, which means FGC performance is proportional to the size of deleted files.

Our **Observation 2** in §III-B shows that the correlative number of a file usually is small. We should note some extreme cases: ① If the deleted files are globally unique in some cases, there is no additional correlative file needed to be read and compared. In that case, the chunks in deleted files are all **non-shared**, and we could directly mark them. ② If the correlative files cover all files in the system, FGC now is degraded as the normal Mark approach, which reads all the files recipes and builds a large Bloom Filter.

We implement the Bloom Filter as Liveness Array to query the chunk liveness, and false-positive rate is an important factor in Bloom Filter. ① False positive rate is determined by the size of Bloom Filter, the number of hash functions, and the number of inserted items. Since FGC reduces the traversing range, the number of chunks to be inserted is largely decreased. Assuming the system stores 1PB logical files, 8KB average chunk size, 0.01% false-positive rate of Bloom Filter, 10MB average file size, 3 average correlative files, and 50TB deleted files (5% of logical files size); and then it needs around 326GB Bloom Filter in original *Mark&Sweep* and 48GB Bloom Filter in FGC (14% of original Bloom Filter). The detailed experiment results are shown in §V-C. Thus, FGC saves Bloom Filter size but keeps the low false positive rate. ② When a chunk is determined as a false-positive case, FGC will not remove that chunk, and it causes space wastage. We could limit the false positive rate to tolerate wastage.

E. Distributed Deduplication with FGC

As introduced in §II-A, the explosive growth of network traffic is a major overhead in the distributed system. We adopt the Ceph decentralized metadata management architecture [10], [11] as the distributed deduplication framework and discuss utilizing FGC to reduce network traffic.

When running GC in the distributed system, the network traffic is used to query chunk liveness across nodes. We summarize two different approaches to clear stale chunks in one node. The first one is querying the liveness of physical chunks among the clusters, which is network-intensive. A node issues lots of queries to other nodes about the liveness of local physical chunks which are not needed in the local alive files. By using FGC, we could only issue queries to the specific nodes which contain the correlative files. Thus, FGC avoids querying all chunks liveness to all other nodes to save the network traffic.

However, querying billions of chunks takes large network traffic and gets vulnerable if an error occurs. Thus, as discussed in §II-A, the second approach is compressing queries in local node and exchanging with other nodes once, which is computational-intensive. In this way, a node could exchange the local Liveness Array (e.g., Bloom Filter) to conduct the global Liveness Array. And then, it could determine the local physical chunks one by one. By using FGC, we focus on building the Liveness Array about the deleted files and correlative files, which is similar to the stand-alone system in §IV-D. Thus, it firstly saves the local computation cost by reducing the traversing scope. Secondly, the smaller Liveness Array reduces the network traffic. Generally, FGC could be used in these two approaches to reduce the network overhead.

F. Discussion

Handling the large files. When FGC traverses correlative files of deleted files, the size of correlative files decides the traversing times and Liveness Array memory overhead. A large

file could result in an amplification problem, as few chunks sharing causes a large correlative file reading. To address this issue, we could set a predefined maximum file size (i.e., granularity) and split the large file into contiguous virtual files. Meanwhile, this consumes more memory to keep the generated virtual files in RefGraph. This granularity will be further studied in §V-C.

What if RefGraph gets crashed. If RefGraph is crashed in memory, we could read the saved RefGraph from the disk and reconstruct new relationships by redo all file changes from checkpoint time. If RefGraph is crashed in the disk, we should reconstruct the whole RefGraph based on all file recipes. The time to reconstruct a new RefGraph includes reading all files recipes and updating RefGraph. In that extreme case, the cost time is similar to the traditional Mark phase due to reading all files recipes.

What if common referenced data is deleted. If the common referenced data is deleted, it could result in a cascading problem that many referenced files will be traversed to rebuild the new relationship. The common referenced data comes from the similar file content (such as file header). We could set the threshold for the heavily referenced chunks and ignore them directly when deleting files. By controlling the number of heavily referenced chunks, it can avoid performance degradation when traversing files widely.

V. Performance Evaluation

A. Experimental Setup

Evaluation Platform. We implement FGC framework based on Destor [2] and perform our experiments on a workstation with Intel Xeon Gold 6130@2.10GHz, 128GB memory, Intel D3-S4610 SSDs, and 7200rpm HDDs.

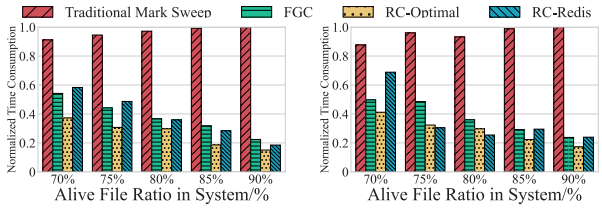
Experimental methods. Before running GC, we firstly run the backup and dedup process. After randomly choosing files to delete, then we run GC to determine the stale chunk. We compare our approach FGC with traditional Mark&Sweep method and two different Reference Count variants.

We implement the traditional Mark&Sweep by traversing all alive files recipes and sweeping stale chunks, representing those approaches that still need detecting all alive file recipes, such as LGC [7]. We implement two different Reference Count methods named Reference Count Optimal (denoted as RC-Optimal) and Reference Count Redis (denoted as RC-Redis). RC-Optimal tracks all chunks in the memory, while RC-Redis uses Redis Application to manage chunks, requiring the extra disk IO to read and write reference values. Thus, we consider RC-Optimal as an ideal implementation and RC-Redis as a real product implementation, such as iDedup [20]. As discussed in §II-B, GMS groups different backup session and CMA removes the entire container when all data in container become out-of-date. However, there is no explicit backup version in primary storage, which means GMS and CMA is unsuitable in primary storage. Thus, we do not compare with them.

Evaluation Datasets. We adopt one specific snapshot of the FSL datasets [27] as our benchmark data, which collect user data on different primary operating systems. As shown in Table II, the datasets cover different operating systems and deduplication ratio.

TABLE II
Statistics of four datasets in evaluation.

Dataset	Raw Size (GB)	#Files	DR (%)
MacOS	239.95	3, 214, 343	43.52%
USER011	311.24	2, 457, 630	36.01%
USER015	217.00	312, 351	49.60%
USER014	173.07	1, 368, 240	61.13%



(a) MacOS (b) USER011

Fig. 7. The normalized time consumption in the stand-alone system.

B. Time Consumption in a Stand-alone System

In this section, we study the FGC improvement with different alive files ratio from 70% to 90%. The actual time consumption is around millisecond-level, and we normalize it to show the improvement clearly. Since the experiment results on different datasets are similar, we only show two of them due to the space limit.

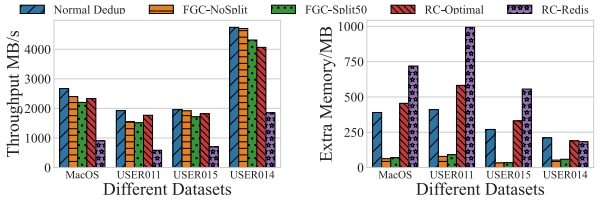
As shown in Fig. 7, when keeping 90% alive files in system, FGC could reduce time consumption by 4.2×-7.34× to the traditional Mark&Sweep. It performs similar to RC-Redis but worse than (around 30%) RC-Optimal. With deleting more files, the time consumption becomes longer, and the improvement begins to decline. When system maintains 70% alive files, FGC reduces the time consumption by about 1.5×-2.35× to traditional Mark&Sweep. However, it performs better than RC-Redis in USER011 dataset. It is because Redis generates much disk IO to update the involved chunk reference value, while FGC only changes the file relationship in memory. RC-Optimal still performs best since it avoids the disk IO to update reference values. The experiment results of USER014, USER015 are similar to USER011.

Since traversing time of correlative files is proportional to the deleted files, our experiments show that FGC performs better than traditional Mark&Sweep and it has the similar performance to the real implementation Reference Count.

C. Overhead of Different GC

In this section, we study the memory and throughput overhead in various GC approaches. We focus on the throughput of Index phase, in which Reference Count will maintain the chunks reference and FGC will maintain the RefGraph structure. Fig. 8 shows the comparison on Index throughput and memory usage in different approaches. The normal dedup approach does not use Reference Count or FGC technique, and it is considered as the benchmark with the highest throughput.

Fig. 8(a) shows the Index throughput results, and FGC does not decrease the Index throughput much since it updates the sparse Austere RefGraph in the file granularity. However, the throughput of RC-Redis declines largely, which is around 30% to 40% of the normal Index throughput. It is because Redis



(a) Throughput Comparison (b) Memory Overhead Comparison

Fig. 8. Index throughput and extra memory usage with different GC

continuously generates disk IO to persist the chunk reference value. RC-Optimal does not cause much throughput degradation since it manages the reference in memory. Besides, tracking all chunks is a memory-consuming process. Fig. 8(b) shows the memory usage of different approaches. FGC reduces memory by 3.96×-8.18× to the traditional Mark&Sweep and 7.63×-30× to Reference Count variants. The total memory overhead of FGC is about tens of MBs (63.9MB to MacOS, 79.1MB to USER011, 53.7MB to USER014, 33.7MB to USER015), which is no more than 0.05% to the datasets size. Generally, compared with traditional Mark&Sweep and Reference Count variants, FGC does not decrease the Index performance much and consumes lower memory overhead.

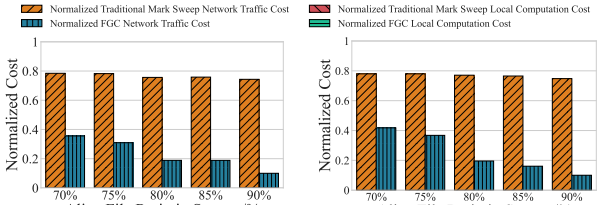
As discussed in §IV-F, FGC could split a large file into multiple contiguous logical virtual files to avoid the amplification problem, and the splitting granularity effects the throughput performance and memory overhead. Fig. 8 also compares the results when setting the splitting parameter as 50 (denoted as FGC-Split50) and no splitting parameter (denoted as FGC-NoSplit). The throughput of FGC-Split50 decreases about 5%-10%, while memory overhead increases about 10%-30% in different datasets. A smaller splitting parameter could generate more vertices in RefGraph, increasing memory consumption.

D. Working in Distributed Systems

In this subsection, we deploy our system in three physical nodes and route the physical chunks and file recipes onto these nodes. The way of experiments are similar to that of above subsections: we first randomly choose files to delete, and then run garbage collection. Here we mainly compare FGC with traditional Mark&Sweep, due to Reference Count is limited by performance issues (caused by complicated transactions to guarantee the consistency of reference count among nodes) and not usually used in the distributed system. We only show the results on two datasets due to the space limit.

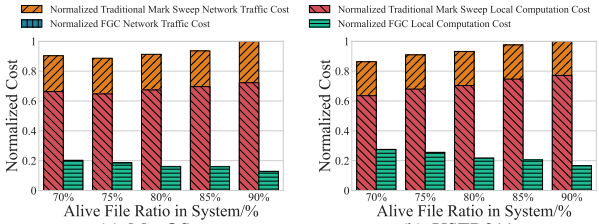
There are two GC strategies for distributed dedup as discussed in §IV-E: network-intensive GC and computational-intensive GC. Network-intensive GC sends queries for each chunk to check their liveness, while computational-intensive GC packs queries in a bloom filter when checking chunks' liveness.

For network-intensive GC, the network traffic is the major overhead and local computation is negligible. Fig. 9 shows the trend of the network/computation cost with different alive files. FGC could reduce network traffic across nodes by 85%-90% when keeping 90% files in system. However, the improvement decreases as deleting more data. It only saves 47%-55% network traffic when system keeps 70% files.



(a) MacOS (b) USER011

Fig. 9. The normalized cost of network-intensive GC in distributed system.



(a) MacOS (b) USER011

Fig. 10. The normalized cost of computational-intensive GC in distributed system.

For computational-intensive GC, the major cost is running local Mark&Sweep and network transmitting cost takes a few proportions. Fig. 10 demonstrates the local computation cost on Mark&Sweep is increasing in traditional Mark&Sweep but it decreases in FGC when keeping more alive files in the system. In the traditional approach, local computational cost takes the main part (around 60%-70%) and network traffic cost takes the other part (around 30%-40%). Compared with traditional Mark&Sweep, FGC could reduce around 45%-90% computational cost with different alive files and the network traffic cost on Bloom Filter is reduced by 128×-256×, which only takes about 1%-5% (hard to identify in barplot). It is because FGC only focuses on deleted files and correlative files.

VI. Conclusion and Future Work

In this paper, we propose a new garbage collection method, Focused Garbage Collection (FGC). Through studying correlative relationships of files, FGC limits the GC task to a local-scale task instead of a global task (traditional *Mark&Sweep*), and thus reduces GC's I/O, computation, and network overheads. Evaluations suggest that FGC reduces about 1.3× - 7.34× Mark time in a stand-alone system and 128×-256× network traffic cost in a distributed system. Meanwhile, the extra memory overhead is negligible $< 0.05\%$. In the future, we will focus on data migration problems by using Austere RefGraph after identifying cross-volume sharing content.

Acknowledgment

Thanks for anonymous reviewers' insightful comments and valuable suggestions. This research was partly supported by the National Key-Research and Development Program of China under Grant No. 2020YFB2104003, the National Natural Science Foundation of China under Grant no. 61972441; Shenzhen Science and Technology Innovation Program under Grant no. RCYX20210609104510007, no. JCYJ20200109113427092, and no. GXWD20201230155427003-20200821172511002; Guangdong Provincial Key Laboratory of Novel Security

Intelligence Technologies (2022B1212010005); Young Innovative Talents Project of General Colleges and Universities in Guangdong Province, 2022KQNCXl59. Wen Xia (wenxia@hit.edu.cn) is the corresponding author.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, 2016.
[2] M. Fu *et al.*, "Design tradeoffs for data deduplication performance in backup workloads," in *Proc. FAST*, 2015.
[3] F. Guo *et al.*, "Building a high-performance deduplication system," in *Proc. USENIX ATC*, 2011.
[4] X. Zhang *et al.*, "Exploiting data deduplication to accelerate live virtual machine migration," in *Proc. Cluster*, 2010.
[5] S. Wu *et al.*, "Ead: a collision-free and high performance deduplication scheme for flash storage systems," in *Proc. ICCD*, 2020.
[6] J. Wei *et al.*, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *Proc. MSST*, 2010.
[7] F. Douglis *et al.*, "The logic of physical garbage collection in deduplicating storage," in *Proc. FAST*, 2017.
[8] D. N. Simha *et al.*, "A scalable deduplication and garbage collection engine for incremental backup," in *Proc. SYSTOR*, 2013.
[9] A. Thomson *et al.*, "Calvin: fast distributed transactions for partitioned database systems," in *Proc. SIGMOD*, 2012.
[10] S. A. Weil *et al.*, "Ceph: A scalable, high-performance distributed file system," in *Proc. OSDI*, 2006.
[11] M. Oh *et al.*, "Design of global data deduplication for a scale-out distributed storage system," in *Proc. ICDCS*, 2018.
[12] A. El-Shimi *et al.*, "Primary data deduplication - large scale study and system design," in *Proc. USENIX ATC*, 2012.
[13] D. T. Meyer *et al.*, "A study of practical deduplication," *ACM Trans. Storage*, 2012.
[14] G. Wallace *et al.*, "Characteristics of backup workloads in production systems," in *Proc. FAST*, 2012.
[15] J. Min *et al.*, "Efficient deduplication techniques for modern backup operation," *IEEE Trans. Computers*, 2011.
[16] C. Zuo *et al.*, "Repec-duet: Ensure high reliability and performance for deduplicated and delta-compressed storage systems," in *Proc. ICCD*, 2019.
[17] J. Kim *et al.*, "Deduplication in ssds: Model and quantitative analysis," in *Proc. MSST*, 2012.
[18] J. Paulo *et al.*, "Efficient deduplication in a distributed primary storage infrastructure," *ACM Trans. Storage*, 2016.
[19] A. T. Clements *et al.*, "Decentralized deduplication in SAN cluster file systems," in *Proc. FAST*, 2009.
[20] K. Srinivasan *et al.*, "idedup: latency-aware, inline data deduplication for primary storage," in *Proc. FAST*, 2012.
[21] M. Fu *et al.*, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *Proc. USENIX ATC*, 2014.
[22] F. C. Botelho *et al.*, "Memory efficient sanitization of a deduplicated storage system," in *Proc. FAST*, 2013.
[23] P. Shilane *et al.*, "99 deduplication problems," in *Proc.HotStorage*, 2016.
[24] K. Shanmugasundaram *et al.*, "Payload attribution via hierarchical bloom filters," in *Proc. CCS*, 2004.
[25] M. Fu *et al.*, "A simulation analysis of redundancy and reliability in primary storage deduplication," *IEEE Trans. Computers*, 2018.
[26] V. Tarasov *et al.*, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX ATC*, 2012.
[27] Z. Sun *et al.*, "A long-term user-centric analysis of deduplication patterns," in *Proc. MSST*, 2016.