CPI: A Collaborative Partial Indexing Design for Large-Scale Deduplication Systems

Yixun Wei, *Student Member, IEEE*, Zhichao Cao, *Member, IEEE*, and David H. C. Du, *Fellow, IEEE*

***Abstract*—Data deduplication relies on a chunk index to identify the redundancy of incoming chunks. As backup data scales, it is impractical to maintain the entire chunk index in memory. Consequently, an index lookup needs to search the portion of the on-storage index, causing a dramatic regression of index lookup throughput. Existing studies propose to search a subset of the whole index (partial index) to limit the storage I/Os and guarantee a high index lookup throughput. However, several core factors of designing partial indexing are not fully exploited. In this paper, we first comprehensively investigate the trade-offs of using different meta-groups, sampling methods, and meta-group selection policies for a partial index. We then propose a Collaborative Partial Index (CPI) which takes advantage of two meta-groups including recipe-segment and container-catalog to achieve more efficient and effective unique chunk identification. CPI further introduces a hook-entry sharing technology and a two-stage eviction policy to reduce memory usage without hurting the deduplication ratio. According to evaluation, with the same constraints of memory usage and storage I/O, CPI achieves a 1.21x-2.17x higher deduplication ratio than the state-of-the-art partial indexing schemes. Alternatively, CPI achieves 1.8X-4.98x higher index lookup throughput than others when the same deduplication ratio is achieved. Compared with full indexing, CPI's maximum deduplication ratio is only 4.07% lower but its throughput is 37.1x - 122.2x of that of full indexing depending on different storage I/O constraints in our evaluation cases.**

***Index Terms*—Article submission, IEEE, IEEEtran, journal, LaTeX, paper, template, typesetting.**

I. Introduction

DATA deduplication is a widely used technique to reduce data storage cost, especially for data backup systems [1], [2], [3]. Data deduplication partitions a data stream into multiple data chunks by a chunking algorithm, identifies the redundant data chunks, and only stores the unique ones (i.e., the ones that have not been processed and stored). The identification of redundant chunks relies on a chunk indexing that records the existence of all the unique data chunks that have been stored in the system. The indexing maintains a **ChunkID** (or called fingerprint in some studies) of each data chunk (i.e., a signature generated by using hash functions like SHA1 or MD5 based on the content of a data chunk) and its stored location. By checking the existence of a ChunkID in the chunk index, the deduplication system determines whether an incoming chunk is a duplicate or a unique chunk. While deduplicating the data stream, a **recipe** is continuously created to record the sequence of data chunks in the original data stream and their corresponding stored locations. The **recipe** will be used to restore the data stream when needed [4], [5], [6], [7], [8], [9]. To save the storage I/Os, the deduplication systems also gather a set of the newly identified unique chunks into a roughly fixed size group (called a **container**) and use the container as the basic I/O unit for storing and restoring [10], [11], [12], [13], [14] data chunks. Like **recipe**, the meta-data information of the data chunks in a container is created together with the container content which is called a **container-catalog** [15]. The effectiveness of deduplication can be evaluated by the **deduplication ratio**, which is defined as original data size divided by the total size of all stored unique chunks.

Initially, a chunk index maintains the ChunkIDs of all unique chunks (considered as **full index** [5], [14], [16], [17]), and all newly arrived data chunks are checked with **full index**. Thus, all duplicated chunks can be identified and the deduplication can achieve the upper bound of the deduplication ratio. A full indexing design (e.g., as an in-memory hash table) has good index lookup throughput initially since the whole chunk indexing can fit into memory. However, as the number of existing unique chunks continuously accumulates, the size of chunk indexing becomes larger than the allocated memory capacity, and thus a large portion of the indexing data must be maintained in storage. As a result, one ChunkID lookup may trigger several more storage reads to fetch a portion of the indexing data and thus cause a severe regression of indexing lookup throughput [14], [18]. Since backup services have a high deduplication throughput requirement to minimize the performance influence of other primary services and to complete the operations within a given backup time window, full indexing is no longer suitable for large-scale deduplication systems in the current big data era.

Partial indexing is introduced to address the aforementioned issue of the full indexing [13], [16], [17], [18] in large scale backup systems. Instead of maintaining the ChunkIDs of all data chunks in memory, partial indexing only stores a subset of ChunkIDs in memory, and the rest is in storage. It

Received 7 December 2023; revised 25 August 2024; accepted 29 September 2024. Date of publication 8 November 2024; date of current version 20 January 2025. This work was supported in part by the NSF CNS Awards: 2412436 and 2412437. Recommended for acceptance by J. Shu. *(Corresponding author: Yixun Wei.)*

Yixun Wei and David H. C. Du are with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA (e-mail: wei00161@umn.edu; du@umn.edu).

Zhichao Cao is with the School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ 85281 USA (e-mail: zhichao.cao@asu.edu).

Digital Object Identifier 10.1109/TC.2024.3485238

0018-9340 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

typically selects or samples a portion of ChunkIDs from the chunk index in memory. The sampled chunks are called **hooks** and will be referenced to different portions in-storage meta-groups (i.e., a collection of ChunkIDs). A meta-group can be either a segment of the recipe or a container-catalog of the unique data chunks consisting of a container since the recipe is created to restore the data stream and a container-catalog can be easily created at the time a container of unique data chunks is written to storage. The index lookup process, after creating a set of incoming data chunks, with the partial indexing follows these steps: 1) Calculate the ChunkIDs of incoming chunks and search against hooks in memory, 2) Based on the number of hook hits after comparing incoming set of ChunkIDs with hooks in memory, fetch the associated in-storage meta-groups related to hook hits to memory for further comparisons, and 3) Compare the incoming chunks against the meta-groups to identify unique or duplicated chunks. The partial indexing design is based on Broder's Theorem [19]: the similarity of two groups (e.g., the group of incoming chunks and an in-storage meta-groups) could be estimated by the probability of whether the minimum selected elements from two groups are equal if the elements from two groups are randomly picked. By restricting the number of storage reads and only searching a small portion of meta-groups, partial indexing can potentially achieve a desirable index lookup throughput and the throughput can be independent of the scale of the chunk indexing. When the meta-groups identified by hooks and fetched from storage are highly relevant to the incoming data chunks, the deduplication ratio of the partial indexing may be sufficiently high. However, it is a challenge to select the hooks to be included in memory since memory capacity is limited.

The goal and challenge of a partial indexing design is to use as few as possible meta-group reads from storage (i.e., achieving a high throughput) to identify as many as possible duplicated data chunks (i.e., high deduplication ratio). The following design factors can affect the throughput and deduplication ratio of partial indexing: 1) The type and organization of meta-groups: Existing studies Sparse Indexing [18], Extreme Binning [20], SiLo [21], and DeFrame [22] partition the recipe into small pieces as meta-groups (called **recipe-segments**). While Progressive Dedup [13] uses the chunk metadata of a container as a meta-group (**container-catalog**). 2) Given the number of unique data chunks continuously increasing, a hook selection policy is required to determine which and how many chunks should be sampled as hooks and stored in the memory with limited capacity. The more hooks selected, the higher the possibility that we can fetch meta-groups that have many overlaps with the incoming data chunks under checkup. However, this will increase the size of the in-memory chunk indexing structure. 3) Given a number of meta-groups having hook hits that can be fetched for further comparison, a meta-group selection policy is required to determine which and how many meta-groups to be read in from storage. This can affect both the deduplication ratio and the throughput.

Existing works use a single type of meta-group (either recipe-segment or container-catalog) without exploring the potential combination of the above three design factors. In this work, we propose a **C**ollaborative **P**artial **I**ndex scheme (called **CPI**), which combines recipe-segments and container-catalogs with dedicated hook sampling method and meta-group selection method to take advantages of their complementary strengths. We comprehensively investigate the influence of different hook sampling methods, hook sampling ratios (i.e., the ratio of hooks sampled from all chunks), and meta-group selection policies to achieve a better trade-off between the deduplication ratio and indexing search throughput with a limited memory capacity.

The proposed design is based on the **following observations and insights**: 1) Container-catalog can achieve a higher deduplication ratio with a high storage read budget and a high hook sampling ratio, while the recipe-segment is more tolerable with limited storage reads (i.e., more I/O efficient) and low hook sampling ratio (i.e., more memory efficient). recipe-segments preserve good inter-version locality and a few reads of recipe-segments can already achieve a considerable deduplication ratio. 2) Position-based sampling (i.e., choose samples based on their orders in a meta-group) collaborates with the container-catalog better, while the recipe-segment does not have an explicit preference for the sampling method. 3) Hook hit frequency-based selection policy for meta-groups performs better on the container-catalog, while the recipe-segment can adopt both frequency or recency-based meta-group selection policies to achieve a similar deduplication ratio. We do not consider re-organizing the meta-groups in other ways (e.g., clustering, sorting, and partitioning the unique ChunkIDs based on some criteria) because these re-organizations bring in extra computing and storage overheads. recipe-segments and container-catalogs are originally generated in the deduplication system. Using them in the partial indexing does not cause any extra overhead. ChunkID-based sampling is used for recipe-segments to exploit the similarity among multiple backup versions. Position-based sampling is used for container-catalogs to better explore the potential duplicates. A higher hook sampling ratio is applied to the container-catalogs as it is more sensitive to sampling ratio which recipe-segments are more tolerant to lower sampling ratio.

The overall memory usage for recipe-segments and container-catalogs is balanced and a two-stage hook eviction policy is applied to ensure the memory budget is not exceeded. The hooks from the old recipe-segments will be first evicted as there are alternative new hooks associated with similar more recent recipe-segments. To further optimize the memory usage of the hook index, a fine-grained meta-group sharing technique is proposed. Hooks share the references to multiple meta-groups if these meta-groups are associated with the same hooks. This sharing method can save up to 41% of hook indexing memory usage in our experiments and should save more memory in a pure recipe-segment partial indexing as the majority of sharing occurs in hooks of the recipe-segments. Moreover, since we adopt both recipe-segments and container-catalogs as meta-groups, we design a customized meta-group selection policy that loads meta-groups with more unique hook hits. This helps to achieve a higher deduplication ratio with fewer I/Os because each selected meta-group is similar

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

to the incoming data chunks but dissimilar to other selected meta-groups. Since recipe-segments and container-catalogs use different sampling methods, an extra hook adjustment is applied when encountering similar recipe-segments or container-catalogs.

According to our evaluation with real-world traces, CPI can always achieve a higher deduplication ratio and higher index lookup throughput than other partial indexing schemes when memory usage is constrained in the same way. Specifically, if the number of meta-group storage reads is capped at 4, 6, or 8, the average deduplication ratio of CPI is 1.21x and 2.17x of that of Sparse Indexing and Progressive Dedup. To achieve the same deduplication ratio (we configured 15 in the evaluation), CPI's index lookup throughput is 1.8x and 4.98x that of Sparse Indexing and Progressive Dedup. Compared with Bloomstore [5], an efficient full deduplication indexing, CPI's maximum deduplication ratio is only 4.07% lower but its throughput is 37.1x - 122.2x of that of Bloomstore depending on different read caps.

II. Workflow of Partial Index Design and Existing Work

The general workflow of a partial indexing design that we considered in this paper includes 1) selecting meta-groups, 2) choosing hooks from the selected meta-groups to be indexed in memory for fast search, 3) searching a batch of in-coming data chunks against the hook index, and 4) selecting a number of meta-groups based on the hook index searching result and deduplicating against them.

As we indicated, we consider both recipe-segments and container-catalogs as meta-groups. From these two meta-groups, a subset of hooks is to be selected in several different ways. In a partial indexing design, to save the I/O accesses required, a batch of incoming new data chunks (considered as a deduplication window) will be searched together against the hook indexing stored in memory to find out all hooks that also appeared in the new batch of chunks (called hook hits). Then the corresponding meta-groups (i.e., the recipe-segments or container-catalogs that include hit hooks) are considered to be accessed from storage. The meta-groups selected and loaded into memory are based on the read I/O budget and memory capacity limitation. The new batch of data chunks then are searched against the ChunkIDs in the selected meta-groups (just accessed from storage) to identify duplicates and "unique" data chunks. After that, "unique" data chunks will be stored in a new container, and a new portion of the recipe is also created.

Once a container is full, it will be flushed to storage, and a container-catalog will be generated at the same time. Once the portion of the newly created recipe reaches a certain size (i.e., a recipe-segment), it is also flushed to storage. During the flush of the container and recipe, some ChunkIDs will be sampled as hooks based on a specific hook selection method. The hooks are inserted into the hook index in memory. Some existing hooks may have to be deleted due to the limited capacity of memory. An upper bound of the deduplication ratio of a partial indexing scheme is achieved when all meta-groups that have at least one hook hit are read from the storage system to apply the deduplication search. This will also determine a lower bound of the lookup throughput of a partial indexing design (i.e., the maximum storage reads per batch). To maintain a high lookup throughput, partial indexing schemes usually restrict the number of meta-groups to be read in from storage.

There are several existing partial indexing schemes [13], [18], [20], [21], [23]. Among them, Sparse Indexing [18] and Progressive Dedup [13] are designed for stream-based deduplication systems, and others are for individual-file-based backup. Sparse Indexing [18], Extreme Binning [20], SiLo [21], DeFrame [22], and LIPA [23] use recipe-segments (i.e., small pieces of recipe) as meta-groups. Sparse Indexing [18] counts the number of hook hits of each meta-group and selects the ones with the most hook hits (i.e., **frequency-based selection**). When deduplicating a new file, Extreme Binning [20] selects the recipe from an already deduplicated file that is most similar to the new file and compares the data chunks of the new file against the data chunks in the selected file. SiLo [21] further extends the Extreme Binning by considering file similarity and chunk locality at the same time. SiLo groups correlated small similar files together as a block and partitioned large files into several blocks. DeFrame [22] investigates the design trade-offs of locality, segmenting, sampling, and selection policies of partial indexing. LIPA [23] uses a reinforcement learning-based algorithm to decide the similarity between recipe-segments and the new data chunks to achieve a higher deduplication ratio.

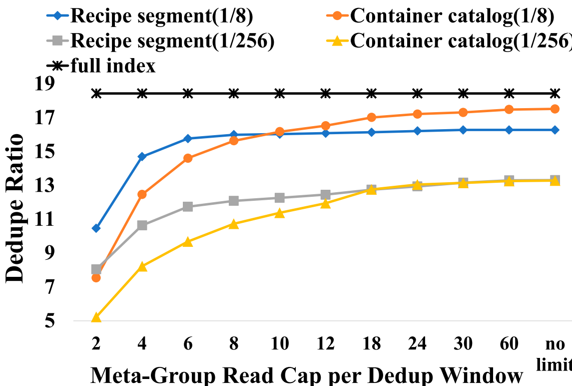
Differently, Progressive Dedup [13] logs the meta-data of chunks in a container as its container-catalog and uses each container-catalog as a meta-group. Only unique data chunks are logged to containers during deduplication, so the container-catalog also preserves some data chunk locality. However, data chunks in a container cover a bigger range in the recipe since the container-catalog only includes "unique" data chunks. Progressive Dedup reads out the container-catalogs with hook hits to an LRU cache without selection (or selecting based on LRU policy) and compares new data chunks against the chunk meta-data in the cache.

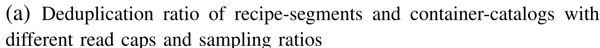
III. Trade-Off Analysis and Motivations

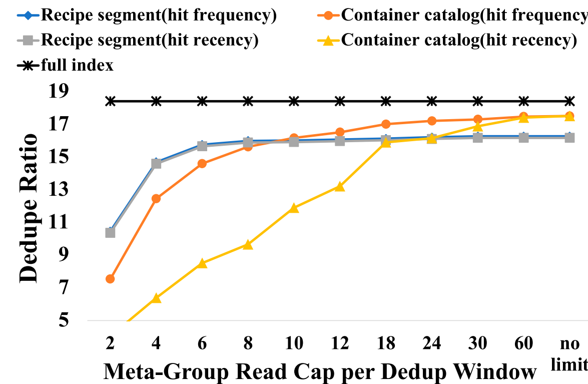
In this section, we explore the achievable deduplication ratio that may be influenced by four critical main design factors of a partial indexing design: 1) meta-group selection and its organization, 2) hook selecting and sampling ratio, 3) meta-group read cap (i.e., the number of I/O reads allowed to access relevant meta-groups from storage), and 4) meta-group selection policies after matching with the incoming data chunks. These observations and insights motivated our novel partial indexing CPI introduced in Section IV.

We first design a set of experiments to explore and validate the effect on the deduplication ratio of each factor. We use the FSL1 trace (described in Section V-A) in the experiments and each trace has 10 backup versions. Every experiment runs 3 times and we present the average results. We use the final deduplication ratio (after 10 versions are deduplicated) as a measurement metric which is also platform-independent. Note

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.




(a) Deduplication ratio of recipe-segments and container-catalogs with different read caps and sampling ratios



(b) The deduplication ratio of different meta-groups, read caps, and selection policy

Fig. 1. Meta-group, read Cap, sampling ratio, and selection policy influence on data deduplication ratio in partial indexing.

that each container-catalog and each recipe-segment have the meta-data information of 1024 data chunks on average. We deduplicate 4 container-size of data chunks together as one batch (4096 chunks per batch). The deduplication cache is set to hold the size of 64 meta-groups (i.e., 64×1024 chunks). **Meta-group read cap** is the maximum number of meta-groups that can be read from storage to the deduplication cache to deduplicate one batch of data chunks. Therefore, the Meta-group read cap almost decides the index lookup throughput. The experiment results are shown in Fig. 1.

A. Recipe-Segment vs. Container-Catalog

recipe-segment and container-catalog are two types of meta-groups used in the existing partial indexing schemes. A recipe-segment is created after 1024 data chunks are deduplicated. Recipe preserves the sequence of each data chunk that appeared in the data stream and includes the metadata of both unique and duplicated data chunks. It will be used for the purpose of restoring the data stream. The size of a recipe-segment is 1024 since the metadata of 1024 data chunks fits into one I/O operation. A container-catalog is generated when a container is full and flushed to the storage. Since only "unique" data chunks are stored, container-catalogs include only the meta-data of "unique" data chunks. Therefore, we have to accumulate the metadata of enough number of "unique" data chunks (i.e., 1024) to write out to storage as one storage I/O. We call this a container-catalog. That is, in this paper, we assume that a recipe-segment and a container-catalog have roughly the same number of metadata entries of 1024 data chunks and contain the essential metadata information of each data chunk which includes both ChunkID and a pointer to its location in storage. In partial indexing, we expect that with limited meta-groups being read into memory, the system can still identify most of the duplicated chunks.

When creating a new backup version, there may be a small portion of data chunks (or files) changed compared with the previous backup version. Therefore, most of the data chunks are the same as in the previous versions. If we use recipe-segments as meta-groups, the recipe-segments from the previous versions have a very high similarity to the batch of data chunks from the current backup version. When limited meta-groups are allowed to be read out to search (i.e., the read cap is small), using recipe-segments with a relatively small number of reads can achieve a sufficiently high deduplication ratio when deduplicating the mostly unchanged data chunks. However, recipe-segments fail to efficiently deduplicate the portions of modified data. Differently, container-catalogs only keep the unique data chunks and can cover a wider range of data chunks than recipe-segments. When the read cap is big, container-catalogs are more useful for exploring the potential redundancy.

The aforementioned impact of different types of meta-groups is also validated in our experiments. As shown in Fig. 1(a), if we use the same hook sampling ratio (1/8) and meta-group selection policy (hit frequency-based), the recipe-segment can achieve a higher deduplication ratio when the read cap is small (i.e., smaller than 10). As the read cap increases (i.e., index lookup throughput decreases), the container-catalog can achieve a higher deduplication ratio.

Using recipe-segments as meta-groups also has its limitations. In a deduplication system, the number of recipe-segments is much larger than that of container-catalogs (as originally defined). For example, if the deduplication ratio of the current data set is R ($R > 1$), the total number of generated recipe-segments is about R times the number of container-catalogs. If we use the same amount of memory for the hook index, the number of hooks from each recipe-segment is only about $1/R$ of that of a container-catalog.

B. Hook Sampling

In the existing partial indexing schemes, position-based sampling (i.e., choosing based on a fixed distance and is considered as uniform), ChunkID-based sampling (considered as random), and minimal ChunkID sampling (minimal) are three major sampling methods being used [22]. Suppose T is the total number of entries of chunk metadata in a meta-group. Position-based sampling selects one chunk out of every N consecutive chunks in a meta-group as a hook. To achieve the same sampling ratio, ChunkID-based sampling selects the chunks whose ChunkIDs

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

mode N = 0. Minimal ChunkID selects $\frac{T}{N}$ ChunkIDs that are the smallest ones in a meta-group. The effectiveness of ChunkID-based sampling and minimal ChunkID sampling is almost the same and verified in the previous study [22].

We explore the effectiveness of different sampling methods for the recipe-segments and container-catalogs. recipe-segments of the same data files usually share high similarity in different backup versions but may have the chunk shifting issue due to small modifications of data. The position-based sampling fails to sample the identical hooks after the shifting of chunks, while ChunkID-based sampling and minimal ChunkID sampling are resistant to chunk-shifting. On the other hand, chunks in real workloads are not exactly uniformly distributed. Merely sampling based on a certain ChunkID family (e.g., ChunkID mod 7 == 0) loses the generality. Position-based sampling can detect the potential useful hooks among all chunks and can help identify the similarity of meta-groups with different versions and types. In general, using position-based sampling for both recipe-segments and container-catalogs can achieve a better deduplication ratio. However, ChunkID-based sampling and minimal ChunkID sampling are also essential for the recipe-segments to detect redundancy in the case of chunk-shifting.

The sampling ratio also influences both the deduplication throughput and the deduplication ratio. With a high sampling ratio (i.e., more hooks are selected from each meta-group), we have more information to compare the similarity between the data chunks to be deduplicated and the meta-groups. Therefore, there is a higher probability of identifying more meta-groups that contain duplicates. However, it consumes more memory space for the hook indexing and may lead to a higher indexing searching and updating overheads. To cache all the hook index in memory, the sampling ratio is forced to be lowered as more meta-groups are created unless we can skip or ignore some meta-groups. As shown in Fig. 1(a), with a higher sampling ratio (1/8), the achieved deduplication ratio is higher (about 30-40%) than the low sampling ratio scenario (1/256).

C. Meta-Group Selection Policies

To achieve a higher lookup throughput, we need to limit the number of meta-group reads per batch data chunk deduplication. Therefore, a selection policy is essential to decide the priority of reading meta-groups from storage to memory per batch chunk deduplication. Sparse Indexing uses the number of hook hits to decide the priority (**hit frequency**). It is based on the observation that if a meta-group has more sampled data chunks appearing in the batched data chunks, other data chunks from the same meta-group may also have a higher probability of finding matches. To exclude the redundancy, a hook hit that appears in multiple meta-groups is only counted once.

Differently, Progressive Dedup selects the meta-groups (container-catalogs) based on the ordering of hook hits (**hit recency**). It is based on the observation that if one hook chunk from a container-catalog has a match with the deduplicated batch, there is a high probability that some of the consecutive chunks from the same container-catalog may also appear in the deduplicated batch due to the log fashion of data

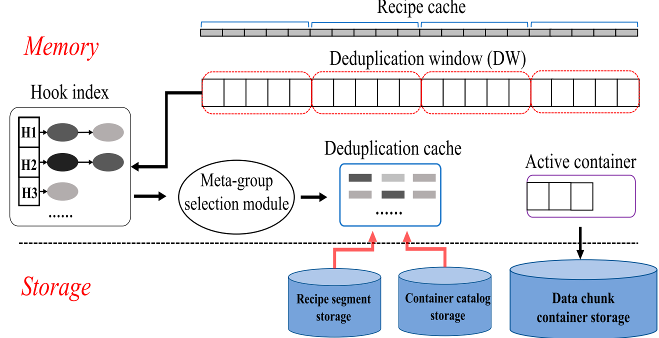


Fig. 2. The architecture overview of CPI.

chunks created in a container which aligns with the appearance sequence of data chunks in the original data stream.

For recipe-segments, hit frequency- and recency-based selection can always identify the most similar recipe-segments (i.e., the corresponding recipe-segments from the previous backup versions). It is also validated in our experiment, as shown in Fig. 1(b). No matter how big the read cap is, the recipe-segment with two different selection policies generates almost the same deduplication ratio. Differently, hit recency cannot cover other data chunk localities behind the first hook. Therefore, the deduplication ratio is always lower than selecting the containers with top N hook hits, while N is the read cap. As shown in Fig 1(b), the deduplication ratio for container-catalog with hit frequency-based selection is always higher. Only when the read cap is unlimited, the selection results from both policies will be the same, and the deduplication ratio will be the same too.

IV. Collaborative Partial Indexing Design

As explored and analyzed in Section III, each factor has a different impact on the deduplication ratio and index lookup throughput. Based on these observations and insights, we propose a **C**ollaborative **P**artial **I**ndexing (**CPI**) that uses both recipe-segments and container-catalogs effectively. By taking advantage of each mega-group's strengths and offsetting their weaknesses, CPI achieves a memory-efficient partial indexing design. We further optimize CPI with a hook-entry sharing method and a two-stage eviction policy to mitigate memory usage without reducing the deduplication ratio when the indexing scales up.

A. Design Overview

Fig. 2 overviews the architecture of CPI. The general workflow of a deduplication system with CPI carries out the following steps: ① A deduplication window (i.e., a batch of incoming chunks being created by a content-based chunking algorithm) with certain memory size is applied to buffer new coming data chunks and to calculate their ChunkIDs until the window is full; note that the deduplication window has a fixed capacity but variable size in terms of the number of chunks. When approaching the window capacity, we avoid cutting objects at

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

midpoints. Instead, we close the window earlier and open up a new window. ② The ChunkIDs of these new data chunks in the deduplication window are searched with the hook indexing that is implemented as an in-memory hash table. The hook index stores all the sampled hooks and their references to the corresponding meta-groups on storage. If a hook has a hit, the meta-groups referenced by this hook are added to a candidate pool maintained by the meta-group selection module; If the incoming chunk does not match any hook or chunks in the deduplication cache, it is considered a unique chunk and stored. ③ After all ChunkIDs in the deduplication window are searched, a meta-group selection policy is applied to the meta-groups in the candidate pool to further select proper meta-groups to load into memory. ④ We maintain a small hash table in memory called a deduplication cache to temporally hold the chunk metadata information from the loaded meta-groups (either recipe-segments or container-catalogs). ⑤ All the data chunks in the deduplication window are deduplicated (i.e., to identify duplicated or unique chunks) against the data chunks cached in the deduplication cache. If a data chunk does not find any match in the cache, it is treated as a "unique" chunk and will be stored in the active container. Otherwise, it is a duplicated data chunk. ⑥ A recipe based on the ordering of each data chunk that appeared in the data stream is generated in the recipe cache; When the active container is full, it is written to storage and the metadata of these chunks are temporarily stayed in the cache. When enough metadata of containers (i.e., 1024 chunks) are accumulated, a corresponding container-catalog is written to storage. ⑦ Once either the recipe cache or the space-holding container-catalogs are full, we sample hooks from them and insert the hooks into the hook index. ⑧ The filled-up recipe-segments or container-catalogs are persisted in storage.

Note that, the deduplication window size is determined by the available in-memory buffer size, which has a fixed capacity. Since content-based chunking is used, data chunks can have different sizes and thus one batch of deduplication may have a slightly different number of chunks. However, in this paper, the recipe-segment and container-catalog have the same number of metadata entries in order to benefit from the large block I/O size (i.e., we try to match the size of recipe-segment and container-catalog with the block size 4KB or larger).

B. Meta-Groups and Hook Selection

We propose to use a combination of recipe-segments and container-catalogs as meta-groups in CPI. The design philosophy is using recipe-segments to exploit the locality of unchanged data from one backup version to the next version, and using container-catalogs to explore the duplicates of data chunks with low localities such as in a new backup version, incremental backups, and updated data chunks. There is no extra storage space overhead as the two types of meta-groups are always generated and stored in a deduplication system. To simplify the discussion, we assume that both recipe-segments and container-catalogs have a fixed number of chunks that are the same size as the deduplication window (we use 1024 chunks in this study).

Once all data chunks in the deduplication window are deduplicated, a recipe-segment is also completed. We apply **ChunkID-based sampling (CBS)** method to extract hooks from the recipe-segments. Importantly, CBS helps identify the recipe-segments with high similarity. If the candidate pool has multiple recipe-segments waiting to be selected to the deduplication cache and they share an explicitly high number of identical hooks, we can avoid the redundant reads. Therefore, only one of them will be considered by the selection policy to improve the meta-group efficiency (Section IV-D).

When enough metadata of containers is accumulated, a catalog is also created and stored separately. Meanwhile, we apply **Position-based sampling (PBS)** method to select hooks for container-catalogs. PBS provides a significantly higher deduplication ratio than CBS for the container-catalog. Since containers only store unique chunks, we do not need to use CBS to identify highly similar container-catalogs.

Based on the observations in Section III, we use 1/128 as the sampling ratio of CBS for new recipe-segments and 1/8 as the sampling ratio of PBS for new container-catalogs to maintain a roughly similar memory usage between recipe-segments and container-catalogs since the number of container-catalogs is roughly 1/16 of the number of recipe-segments. Besides, a recipe-segment contains the metadata of all unique and duplicated data chunks, but a container-catalog contains only metadata from unique data chunks. Therefore, a higher sampling ratio for container-catalogs is required. Also, the deduplication ratio is observed around 16 in our test setting (Section V), we thus initially have a 1/16 sampling ratio of recipe-segments compared to that of container-catalogs. The sampling ratio will be adjusted lower when CPI is under memory pressure (Section IV-C).

C. Hook Indexing Design and Its Adjustment

The job of the hook indexing is to quickly identify whether the ChunkID of a data chunk in the deduplication window matches any hooks and provides the mapping from the hit hooks to their corresponding meta-groups. Please note that the same hook may be included in a number of meta-groups. The hook indexing is implemented as a hash table in memory. The key is the ChunkID of a hook and the value is a pointer pointing to a list of meta-groups that are associated with a hook. When inserting a hook, if the hook does not exist, a new key-value pair in the form of <ChunkID, pointer to meta-group list> is added to the hook index. If the hook has already been sampled, the new location of the current meta-group is appended to the meta-group list as its value part. With the location, we can access a new corresponding meta-group from storage.

Since hooks and meta-groups from both recipe-segments and container-catalogs are maintained, memory usage can be higher than using only one of them. To cope with the increasing memory usage pressure, we adopt two memory-saving methods: 1) **Meta-group list sharing** and 2) **Two-stage hook indexing eviction**.

1) Meta-Group List Sharing: When multiple hooks are sampled from the same set of meta-groups, Meta-group list

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

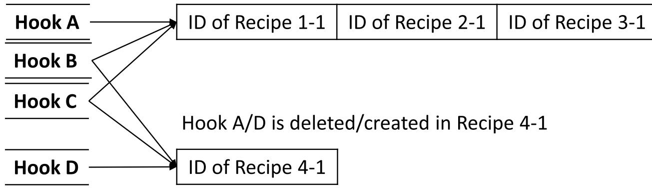


Fig. 3. An example of meta-group list sharing.

Algorithm 1: Meta-group Sharing

Notion : Hook indexing I, meta-group list L, hook set H sampled from current meta-group C

1 **Function** `NewSharing`(H,C):
2
3 create L_{new}
4 append (L_{new}, C)
5 **foreach** hook i in H **do**
6 create $i._{newPointer}$
7 $i._{newPointer} -> L_{new}$
8 $L_{new}.counter++$
9 **end**
10 **Function** `Main`():
11 while generating a new meta-group
12 **if** $\exists$ hook $i \in H$ and $\notin I$
13 `NewSharing`(H,C)
14 **else**
15 go over hooks in H, find the sharing list L
16 **if** $|H| < L.counter$
17 `NewSharing`(H,C)
18 **else**
19 append(L,C)
20 $L.counter++$

sharing is used to exploit the redundancy via sharing one common meta-group list among multiple hooks. The redundancy usually happens to hooks from recipe-segments: unchanged or less changed recipe-segments are generated from backup version after version, and the CBS selects the same set of hooks again and again. Therefore, one hook will have a pointer list reference to a number of recipe-segments and several of the hooks will have the same recipe-segment list. For hooks of container-catalogs, since only hooks sampled from the same container-catalog can share the meta-group list, hooks from different container-catalogs are unlikely to be the same.

An example of meta-group list sharing is shown in Fig. 3. Hooks A & B & C were initially sampled from the first recipe-segment of the first backup version (i.e., recipe 1-1). They then appeared again in the first recipe-segment of the following two backup versions (i.e., recipe 2-1 and recipe 3-1). The IDs of the three recipe-segments form a meta-group list and the list is shared among hooks A, B, and C.

It is possible that the same recipe-segments of different backup versions have created or deleted some hooks. In this case, a new meta-group list is created, and the existing hooks (and the newly created hooks if they exist) will have a new pointer pointing to the new meta-group list. The meta-group list-sharing procedure is described in Algorithm 1.

Compared with the typical hook index, meta-group list sharing allows each hook to have multiple pointers. The memory saving from meta-group list sharing can offset the overhead of extra pointers as long as the shared list has more than one meta-group[1]. In case a hook has too many extra pointers pointing to multiple shared lists, and each list has only one meta-group, we run a back-end thread to revoke the extra pointers of the hook. The hook will maintain a unique list recording all its associated meta-groups.

Since the meta-group sharing mainly comes from recipe-segments, the memory saving is determined by the sampling ratio of the recipe-segment. The sharing technology can save up to 41% of hook indexing memory usage in our design (i.e., $\frac{1}{128}$ recipe-segment hook sampling ratio) and can save more memory in a pure recipe-segment based partial index.

2) Two-Stage Hook Index Eviction: As more and more meta-groups are generated, more hooks and meta-group lists are created in the hook index. The hook index will inevitably exceed its memory capacity budget. We periodically check memory usage, evict hooks, and their associated meta-group lists from the hook indexing once the memory usage exceeds 95% of the memory capacity budget. The eviction will stop until the usage is lower than 90%. The eviction consists of two stages: 1) partially evict ChunkIDs of old recipe-segments from certain meta-group lists that are large or wildly shared, and 2) randomly evict hooks to decrease the overall sampling ratio.

In the first stage, we find top $(\frac{\textit{current mem usage}}{\textit{mem budget}} - 0.9) \times 100\%$ meta-group lists that are either large in terms of the number of meta-groups contained or mostly shared by hooks. We then evict all old recipe-segments (except the recipe-segments of the latest version) from the selected meta-group lists. The eviction has less influence on the deduplication ratio because the evicted recipe-segments in the selected meta-group lists are very similar to each other. If the evicted recipe-segments vary significantly from each other, the meta-group list is unlikely to be large or wildly shared. We find that the latest backup version usually contributes the most to the deduplication ratio, thus we keep the latest one if it is possible.

We run Stage 1 eviction until all recipe-segments of older versions have been evicted. If the memory usage is still at a high watermark (e.g., exceeds 90% of the memory budget), we have to lower the sampling ratio of the remaining meta-groups. Therefore, in Stage 2, we randomly evict the hooks and also delete the meta-group list if its associated hooks are fully removed. This process reduces the sampling ratio of both recipe-segments and container-catalogs. With the same meta-group read cap for the deduplication window, the deduplication ratio can be slightly lower.

D. Meta-Group Selection Policy

In the Meta-group selection stage, we search each chunk against the hook indexing and get a number of meta-groups that have at least one hook hit. Reading all those meta-groups to apply the deduplication can achieve a better deduplication ratio. However, to achieve a high index lookup throughput, the deduplication system needs to limit the number of meta-group

[1]The pointer and the location of a meta-group are both 4-byte integers.

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

reads per batch deduplication to N (i.e., at most N meta-group reads per batch deduplication). Therefore, CPI can only select at most N meta-groups that can achieve the highest deduplication ratio than other selection policies. Based on Broder's Theorem, we select the N meta-groups with the highest numbers of hook hits.

There will be at most N round of meta-group selection: ① At round $i (i \leq N)$, select the meta-group from all remaining candidates with the highest number of hook hits. ② Remove the meta-group from the candidate pool. ③ Reduce the hook hit number from the remaining meta-group candidates if they share the same hook hit with the selected meta-group. At step ③, it eliminates the probability of loading similar meta-groups since one hook hit will only be counted once. After we select one meta-group, the hook hit number of its similar peers will be adjusted to a low level and are unlikely to be loaded again.

However, this method can only identify the similarity between recipe-recipe or container-container scenarios. It cannot successfully identify the similarity between a recipe-segment and a container-catalog. Recipe-segments and container-catalogs use different sampling methods, their similarity cannot be simply identified based on hooks. Instead of identifying and deselecting similar recipes and containers during meta-group selection, we decide to ignore this similarity during hook sampling. A recipe-segment can share lots of chunks with a container-catalog only when a set of completely new data chunks comes. Therefore, each time we encounter a new backup (e.g., an explicit drop of deduplication ratio or hint from users), we only index the hooks from recipe-segments for future meta-group selection. We index recipe-segments rather than container-catalogs because the reserved recipe-segment can help identify the similarity with recipe-segments appearing in later backup versions.

E. Workload Adaptation and Optimizations

We further optimize CPI by adjusting the meta-group read cap N dynamically based on the current characteristics of the workload during deduplication, while keeping the same average number of meta-group reads. To ensure a stable hook index lookup throughput, existing studies [13], [18], [20] usually set a fixed meta-group read cap C (limiting the number of meta-group reads) per deduplication window. The actual meta-group read cap for different deduplication windows varies, which can improve the deduplication ratio without increasing the number of storage reads and lowering the average index lookup throughput.

We design a heuristic algorithm for this purpose. We monitor H_{ave}, which is the average number of meta-groups that have hook hits per deduplication window. A higher H_{ave} indicates that duplicated chunks are scattered to more meta-groups (i.e., low chunk locality), and thus a higher meta-group read cap is needed. Suppose the pre-defined meta-group read cap is C per deduplication window. Consider the current deduplication window i (DW^i) and assume the average number of meta-groups that have hook hits per deduplication window is H^i_{ave}, and the accumulated total number of actual meta-group reads is R^i. Therefore, the maximal number of meta-group reads that is allowed for DW^{i+1} is $C \times (i+1) - R^i$. Suppose the number of meta-groups with hook hits in DW^{i+1} is H^{i+1}. If $H^{i+1} \leq H^i_{ave}$, we use C as the meta-group read cap for DW^{i+1}. In this situation, the actual meta-group reads may be smaller than C. The "unused" meta-group reads can be accumulated for future deduplication windows as credits. If $H^{i+1} > H^i_{ave}$, the new read cap is: $min(C \times (i+1) - R^i, C \times \frac{H^{i+1}}{H^i_{ave}})$.

V. Implementation and Evaluation

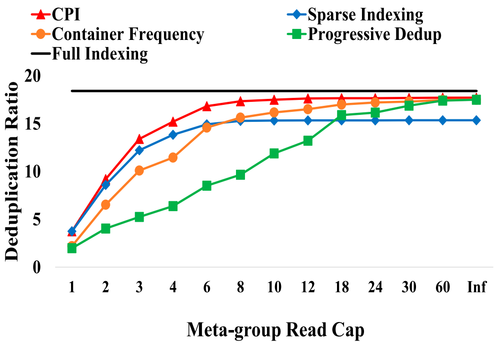
In this section, we present the performance comparisons between CPI and other deduplication indexing schemes. Specifically, we implement BloomStore [5] as the baseline of the full indexing scheme. We also implement HP Sparse Indexing [18] and Symantec Progressive Dedup [13] as state-of-the-art partial indexing schemes. Sparse Indexing is designed based on recipe-segments, ChunkID-based sampling, and hit-frequency-based meta-group selection policy. Progressive Dedup uses container-catalogs, position-based sampling, and hit-recency-based meta-group selection policy. To achieve a comprehensive comparison, we further implement another partial indexing baseline (i.e., Container Frequency) that uses container-catalogs, position-based sampling, and hit-frequency-based meta-group selection. There are other possible choices (e.g., recipe-segment + position-based sampling + hit-recency selection), but these combinations are inferior to the above partial indexing schemes in terms of deduplication ratio as we have seen in Section III. We maintain the same memory usage across all the schemes to fairly compare the overall deduplication ratio (i.e., deduplication effectiveness) and index lookup throughput (i.e., deduplication efficiency) of these schemes.

A. Experimental Setup and Data Sets

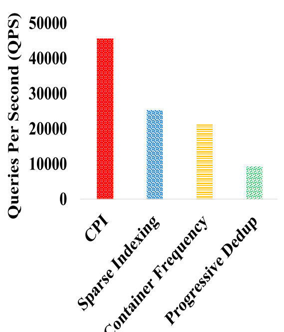
The prototype is deployed on a Dell PowerEdge R430 server with two six-core 2.4GHz Intel Xeon E5-2620 v3 CPUs (24 total system hyperthreads) and 64GB of memory. The storage used is an Intel 660p NVME SSD. In our experiments, the size of the container-catalog, recipe-segment, or deduplication window is all configured as 1024 chunks. We control the index lookup throughput by limiting the meta-group read cap per deduplication window. Memory usage is hard to maintain exactly the same because the number of hooks in ChunkID-based sampling is content-related. We assign the partial indexing schemes with different initial sampling ratios (i.e., $\frac{1}{64}$ for Sparse Indexing, $\frac{1}{8}$ for Progressive Dedup and Container Frequency) so that their initial memory usage can be roughly the same. When the deduplication system processed a large data stream, very soon the schemes will exceed a predefined memory budget (32 GB in our test case) and trigger their own eviction policies (random for Container Frequency). The memory usage is then maintained at the same level all the time.

In the experiments, we use six real-world deduplication traces. Traces MAC1, MAC2, FSL1, FSL2 are from the File System and Storage Lab (FSL) [24], [25], [26]. The MAC1 and MAC2 traces are the full backups of a Mac OS X Snow Leopard server running in an academic computer lab. FSL1 and

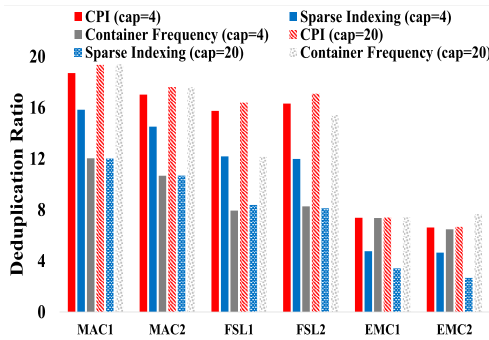
Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.



(a) The deduplication ratio comparison between four partial indexing schemes when meta-group read cap varies



(b) The lookup throughput (queries per second) comparison of four schemes (deduplication ratio = 15)



(c) The deduplication ratio comparison of four partial indexing in six traces

Fig. 4. The deduplication evaluation results.

FSL2 traces are the full backups of students' home directories from a shared network file system. Traces EMC1 and EMC2 are the weekly full-backup traces from EMC [27]. Note that we use a different maximum chunk size for each trace. That is, 16 KB average chunk size for FSL traces and 8 KB average chunk size for EMC traces because the scenario that needs partial indexing typically has a very large backup dataset and 4 KB chunk is too small to be efficient. Note that, the total number of storage reads of one experiment equals the number of read cap times the total number of deduplication batches. The total number of writes equals to the sum of the number of recipe-segments, the number of container-catalogs, and the total number of container writes. Thanks for the suggestion. We add a discussion about the comparison of a deployed system and our prototype in Section V-A. In a deployed storage system, the deduplication process is expected to complete quickly within a backup window with less interference to other background processes. Therefore, it is common to constrain the IO bandwidth that is allocated to data deduplication to complete its task. IO bandwidth consumed by deduplication process is dominated by the reads of container catalogs and recipe segments as they are loaded every time a batch of new data comes. Therefore, we adjust the read cap to explicitly limit the IO bandwidth. Although the consumption of IO bandwidth is also influenced by other factors like deduplication cache, restricting IO time can essentially show the deduplication ratio of a deduplication system with different IO bandwidth quota.

B. Deduplication Effectiveness and Efficiency Comparison

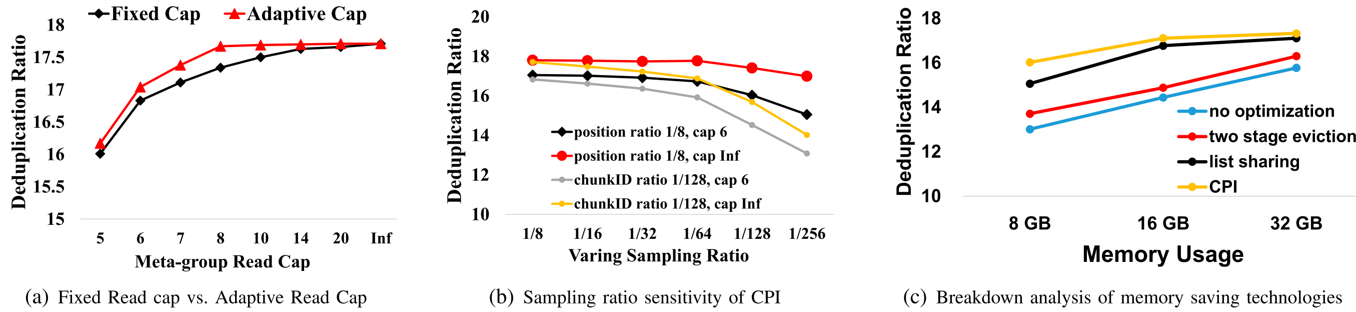
In this subsection, we first compare the deduplication ratio of the four partial indexing designs. Then, we compare the index lookup throughput (queries per second) when they achieve the same deduplication ratio. Finally, we compare CPI with BloomStore (an implementation of the full index).

We increase the read cap per 4096 data chunk deduplication from 1 to ∞ and compare their achievable deduplication ratios. The results are shown in Fig. 4(a). When the read cap is small (e.g., $\leq$8), Sparse Indexing that uses recipe-segments can harvest more data locality than other schemes that only use container-catalogs. When the read cap is large, container-catalog based indexing can further explore potential chunk redundancy and achieve a higher deduplication ratio. CPI, combining the advantages of both recipe-segments and container-catalogs, can achieve a higher deduplication ratio in both low and high read caps. In particular, when the read cap is between 4-8 (a typical read cap range that can achieve fast and sufficient deduplication), the deduplication ratio of CPI is on average 1.21x, 1.28x, and 2.17x of that of Sparse Index, Container Frequency, and Progressive Dedup respectively. When the read cap > 10, the deduplication ratio of CPI is very close to that of the full indexing which has the deduplication ratio upper bound (i.e., black line). When the read cap ≤ 60, CPI can always achieve the highest deduplication ratio. If the read cap is ∞, CPI, Container Sort, and Progressive Dedup can achieve the same deduplication ratio since all of them use container-catalogs to explore the duplicates, which can achieve a higher deduplication ratio than that of Sparse Indexing.

We also compare the index lookup throughput of the four partial indexing designs. The results are shown in Fig. 4(b). CPI can achieve about 45,500 lookups per second, which is 1.8x, 2.15x, and 4.98x that of Sparse Indexing, Container Frequency, and Progressive Dedup, respectively. Integrating recipe-segments and container-catalogs in the partial indexing design to better explore the similarity and uniqueness, CPI can select only a few meta-groups to achieve the same deduplication ratio as pure recipe-segments or pure container-catalog based schemes. The saving in the loading of meta-groups greatly improves the QPS of CPI indexing.

To better understand CPI's deduplication efficiency and effectiveness, we compare it with BloomStore, an efficient BloomFilter-based full deduplication indexing. To achieve a fair comparison, we run CPI first and measure its total memory usage. Then, we configure the Bloom Filter buffer of BloomStore to occupy the same amount of memory. BloomStore is configured with 65 instance partitions, false positive rate = 0.001, and 1024 chunk metadata KV pairs per flash page which is the same as one meta-group read in CPI. For the six traces, the average deduplication ratio of CPI is only 8.59%

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.



(a) Fixed Read cap vs. Adaptive Read Cap (b) Sampling ratio sensitivity of CPI (c) Breakdown analysis of memory saving technologies

Fig. 5. The evaluation of the (a) adaptive read cap; (b) varying sampling ratios (fixed position based sampling ratio = 1/8 changing ChunkID based sampling ratio from 1/8 to 1/128) or (fixed ChunkID based sampling ratio = 1/128 changing position based sampling ratio from 1/8 to 1/128); (c) restrict the memory usage from 8 GB to 32 GB, measure the deduplication ratio of CPI with and without memory saving technologies including two stage eviction and list sharing.

(cap = 6) and 4.07% (cap = 24) lower than those of the full indexing implementation BloomStore (deduplication ratio upper bound), respectively. However, the number of storage reads of BloomStore is about **122.2x** (cap = 6) and **37.1x** (cap = 24) higher than those of CPI, respectively. In general, with a small deduplication ratio reduction, CPI can achieve a much higher index lookup throughput than that of a full index.

C. Breakdown Analysis of Traces

We further analyze the deduplication ratios of six traces when the read cap is set to 4 and 20. Since the outcome of Progressive Dedup is always lower than that of Container Sort when the read cap is small, we do not plot its results in Fig. 4(c). With both low and high read caps, CPI always performs best in all traces except EMC2. In EMC traces, the ratio of changes between the two versions is much higher and has a low data chunk locality. This leads to a low overall deduplication ratio, especially for indexing schemes that use recipe-segments (i.e., Sparse Indexing). Although CPI also uses recipe-segments, it successfully excludes the recipe-segments that have low similarity and relies more on the container-catalogs to explore the duplicates, which leads to a similar deduplication ratio as that of Container Frequency. In general, for different workloads with different data chunk localities, CPI can achieve a higher deduplication ratio especially when the meta-group read cap is small (i.e., when a high index lookup throughput is achieved).

D. Adaptive Meta-Group Read Caps

We optimize CPI with an adaptive meta-group read cap mechanism. To evaluate the effectiveness of the adaptive meta-group read cap, we compare the deduplication ratio of two CPI implementations: fixed cap vs. adaptive cap. We vary the meta-group read cap from 1 to ∞ and evaluate the overall deduplication ratio of the six traces. When the cap is smaller than 5, the adaptive read cap does not achieve explicit improvement because each batch of deduplication has fully utilized its read caps. Therefore, the deduplication ratios of the two schemes are the same, therefore, we do not plot them out in the figure. The deduplication ratio comparison is shown in Fig. 5(c) as the meta-group read cap increases from 5 to ∞. The adaptive cap is always better than the fixed cap in terms of the achieved deduplication ratio. By utilizing the adaptive read cap, we can approach CPI's maximum deduplication ratio with fewer reads. For example, fixed read cap CPI achieves the maximum deduplication ratio when the read cap > 20, while the adaptive cap CPI achieves it when the read cap > 10.

E. Sensitivity Analysis of Sampling Ratio

Since we use two sampling methods with two different sampling ratios in CPI (i.e., 1/8 for Position-based and 1/128 for ChunkID-based), we separately evaluate the influence of different sampling ratios in this subsection. First, we fix the sampling ratio of Position-based to 1/8 and vary the sampling ratio of ChunkID-based hooks from 1/8 to 1/256. Then, we fix the ChunkID-based hook sampling ratio to 1/128 and vary the Position-based hook sampling ratio from 1/8 to 1/256. The results of the overall six traces are shown in Fig. 5(b).

Compared with pure recipe-segment or pure container-catalog (Section III) schemes, CPI shows a higher resistance to the decrease of sampling ratio. The deduplication ratio of CPI drops slightly when the sampling ratio $\leq$1/64 (e.g., 1.87% and 5.41% for ChunkID-based and Position-based CPI with cap=6). When the sampling ratio $>$ 1/64, the deduplication ratio drops relatively quicker. However, the overall drop is not as significant as other existing schemes. The decrease in Position-based sampling still has more influence on the deduplication ratio as it is originally more sensitive to the sampling ratio. The low read cap (i.e., 6) is more sensitive to the sampling ratio mainly because the low read cap relies more on high-quality reads while the decrease in sampling ratio would cause a problem in identifying similar meta-groups.

F. Breakdown Analysis of Memory Saving Technologies

Without the read cap constraint, we test the deduplication ratio of CPI under different memory usage with and without memory-saving technologies including two-stage eviction and list sharing. The default position-based sampling ratio is 1/8 and the default ChunkID-based sampling ratio is 1/128. For CPI without memory-saving technology (i.e., "no optimization" in Fig. 5c), a random eviction is adopted when the memory is full. Overall, the memory-saving technologies contribute 11%-20% more deduplication ratio when memory usage ranges

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

from 32 GB to 8 GB. List-sharing technology contributes more deduplication effectiveness than two-stage eviction. That is because recipe-segments consume the most available memory and list sharing is especially useful to reduce the memory usage of recipe-segments. However, the list-sharing scheme cannot purely be effective without two-stage eviction as the deduplication ratio of the pure list-sharing scheme decreases significantly when the memory usage decreases to 8 GB.

VI. Conclusion and Future Work

In this paper, we first analyze and discuss the limitations of partial indexing that is purely based on either recipe-segment or container-catalog. The observations of their shortcomings motivate us to design a new partial indexing scheme called CPI. CPI combines both recipe-segments and container-catalogs to achieve a higher deduplication ratio with a target deduplication throughput than the existing methods including Sparse Indexing and Progressive Dedup with the same number of meta-group reads. We further propose a meta-group list-sharing algorithm and a two-stage hook indexing eviction algorithm to save the memory usage of CPI and to improve its performance.

Note that CPI is currently implemented as a prototype of a centralized deduplication system in which backup data streams coming from multiple clients and being processed in a single server. This implementation allows us to check and demonstrate the benefits of our design. We noticed there is another trend of global deduplication which distributes data and indexing to multiple servers to increase the parallelism of deduplication [28], [29], [30], [31], [32]. It may be essential to extend the CPI into a distributed scenario. We plan to do so in our future work. Meanwhile, the adoption of CPI may also trigger modifications of other parts of data deduplication systems including garbage collection [33], [34], and restore [35], [36], and migration [37], [38]. In our future work, we also plan to investigate the potential of integrating CPI with the deduplication rewrite to optimize both deduplication and restore performance.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable feedback and revision suggestions. We thank all the members of CRIS at UMN and ASU-IDI at ASU for providing useful comments to improve our design, evaluation, and paper writing.

References

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

[2] S. Singhal, P. Sharma, R. K. Aggarwal, and V. Passricha, "A global survey on data deduplication," *Int. J. Grid High Perform. Comput.*, vol. 10, no. 4, pp. 43–66, 2018.

[3] A. Godavari and C. Sudhakar, "A survey on deduplication systems," *Int. J. Grid Utility Comput.*, vol. 15, no. 2, pp. 143–159, 2024.

[4] Z. Cao, H. Wen, F. Wu, and D. H. Du, "ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching," in *Proc.16th USENIX Conf. File Storage Technol. (FAST 18)*, Oakland, CA, USA: USENIX Association, 2018, pp. 309–323.

[5] G. Lu, Y. J. Nam, and D. H. Du, "BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol. (MSST)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 1–11.

[6] D. Park, Z. Fan, Y. J. Nam, and D. H. Du, "A lookahead read cache: Improving read performance for deduplication backup storage," *J. Comput. Sci. Technol.*, vol. 32, no. 1, pp. 26–40, 2017.

[7] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. Du, "Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, pp. 129–142.

[8] Y. Nam, G. Lu, and D. H. Du, "Reliability-aware deduplication storage: Assuring chunk reliability and chunk loss severity," in *Proc. Int. Green Comput. Conf. Workshops*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 1–6.

[9] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 581–586.

[10] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang, "Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 121–128.

[11] W. Xia, H. Jiang, D. Feng, and L. Tian, "Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets," in *Proc. Data Compression Conf.*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 203–212.

[12] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN-optimized replication of backup datasets using stream-informed delta compression," *ACM Trans. Storage*, vol. 8, no. 4, pp. 1–26, 2012.

[13] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2011, pp. 271–294.

[14] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST)*, 2008, pp. 1–14.

[15] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 271–294.

[16] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA: ACM, 2011, pp. 25–36.

[17] B. K. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2010, pp. 68–81.

[18] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th USENIX Conf. File Storage Technol. (FAST)*, 2009, pp. 111–123.

[19] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," in *Proc. 30th Annu. ACM Symp. Theory Comput.*, 1998, pp. 327–336.

[20] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proc. IEEE 17th Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst. (MASCOTS)*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 1–9.

[21] 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. USENIX Annual Technical Conference (USENIX ATC)*, 2011, pp. 26–30.

[22] M. Fu et al., "Design tradeoffs for data deduplication performance in backup workloads," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 331–344.

[23] G. Xu, B. Tang, H. Lu, Q. Yu, and C. W. Sung, "LIPA: A learning-based indexing and prefetching approach for data deduplication," in *Proc. IEEE 35th Symp. Mass Storage Syst. Technol. (MSST)*, 2019, pp. 1–14.

[24] A. M. Vasily Tarasov, "Traces and snapshots public archive," filesystems.org. Accessed Jun. 5, 2024. [Online]. Available: http://tracer.filesystems.org/

[25] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2012, pp. 261–272.

[26] Z. Sun et al., "A long-term user-centric analysis of deduplication patterns," in *Proc. IEEE 32nd Symp. Mass Storage Syst. Technol. (MSST)*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–7.

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.

[27] N. Park and D. J. Lilja, "Characterizing datasets for data deduplication in backup applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 1–10.

[28] T. Wong, S. Thakkar, K.-F. Hsieh, Z. Tom, H. Saraiya, and P. Shilane, "Dataset similarity detection for global deduplication in the DD file system," in *Proc. IEEE 39th Int. Conf. Data Eng. (ICDE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 3322–3335.

[29] M. Oh et al., "Design of global data deduplication for a scale-out distributed storage system," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 1063–1073.

[30] S. Luo, G. Zhang, C. Wu, S. U. Khan, and K. Li, "Boafft: Distributed deduplication for big data storage in the cloud," *IEEE Trans. Cloud Comput.*, vol. 8, no. 4, pp. 1199–1211, Oct.–Dec. 2015.

[31] M. Oh et al., "{TiDedup}: A new distributed deduplication architecture for CEPH," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2023, pp. 117–131.

[32] Y. Long and Y. Fu, "A fast deduplication scheme for stored data in distributed storage systems," in *Proc. 8th Int. Symp. Adv. Elect., Electron., Comput. Eng. (ISAEECE)*, vol. 12704, Hangzhou, China: SPIE, 2023, pp. 681–686.

[33] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. Botelho, "The logic of physical garbage collection in deduplicating storage," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 29–44.

[34] J. Yuan et al., "A focused garbage collection approach for primary deduplicated storage with low memory overhead," in *Proc. IEEE 40th Int. Conf. Comput. Des. (ICCD)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 315–323.

[35] Y. Zhang et al., "Improving restore performance for in-line backup system combining deduplication and delta compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2302–2314, Oct. 2020.

[36] L. Lin, Y. Deng, and Y. Zhou, "Improving restore performance of deduplication systems via a greedy rewriting scheme," in *Proc. IEEE 27th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 291–298.

[37] G. Cheng, L. Luo, J. Xia, D. Guo, and Y. Sun, "When deduplication meets migration: An efficient and adaptive strategy in distributed storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 10, pp. 2749–2766, Oct. 2023.

[38] R. Kisous, A. Kolikant, A. Duggal, S. Sheinvald, and G. Yadgar, "The what, the from, and the to: The migration games in deduplicated systems," *ACM Trans. Storage*, vol. 18, no. 4, pp. 1–29, 2022.

Yixun Wei (Student Member, IEEE) received the bachelor's degree in computer science from Shandong University, Qingdao, China. He is currently working toward the Ph.D. degree with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA. His research interests include data storage systems and data intensive applications.

Zhichao Cao (Member, IEEE) received the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, MN, USA. He is an Assistant Professor with the School of Computing and Augmented Intelligence, Arizona State University. His research interests cover areas of database systems, storage systems, and next-generation data infrastructure. His research interests also lie in the design and development of data management systems for new storage technologies, query engines for large-scale scientific computing in HPC, and storage solutions for AI/ML platforms.

David H. C. Du (Fellow, IEEE) received the Ph.D. degree in computer science from the University of Washington, Seattle, WA, USA. He is the Qwest Chair Professor at the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA. His research interests include intelligent storage systems, vehicular networks, and cyber-physical systems.

Authorized licensed use limited to: Arizona State University. Downloaded on July 02,2025 at 23:55:54 UTC from IEEE Xplore. Restrictions apply.