Scientific Research Publishing

Journal of Software Engineering and Applications, 2019, 12, 460-490
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

dCACH: Content Aware Clustered and Hierarchical Distributed Deduplication

Girum Dagnaw, Ke Zhou, Hua Wang

Huazhong University of Science and Technology, Wuhan National Laboratory for Optoelectronics, Wuhan, China
Email: girumdagnaw@yahoo.com, k.zhou@hust.edu.cn, hwang@hust.edu.cn

How to cite this paper: Dagnaw, G., Zhou, K. and Wang, H. (2019) dCACH: Content Aware Clustered and Hierarchical Distributed Deduplication. *Journal of Software Engineering and Applications*, **12**, 460-490.
https://doi.org/10.4236/jsea.2019.1211029

Received: October 17, 2019
Accepted: November 24, 2019
Published: November 27, 2019

Abstract

In deduplication, index-lookup disk bottleneck is a major obstacle which limits the throughput of backup processes. One way to minimize the effect of this issue and boost speed is to use very high course-grained chunks for deduplication at a cost of low storage saving and limited scalability. Another way is to distribute the deduplication process among multiple nodes but this approach introduces storage node island effect and also incurs high communication cost. In this paper, we explore dCACH, a content-aware clustered and hierarchical deduplication system, which implements a hybrid of inline course grained and offline fine-grained distributed deduplication where routing decisions are made for a set of files instead of single files. It utilizes bloom filters for detecting similarity between a data stream and previous data streams and performs stateful routing which solves the storage node island problem. Moreover, it exploits the negligibly small amount of content shared among chunks from different file types to create groups of files and deduplicate each group in their own fingerprint index space. It implements hierarchical deduplication to reduce the size of fingerprint indexes at the global level, where only files and big sized segments are deduplicated. Locality is created and exploited first using the big sized segments deduplicated at the global level and second by routing a set of consecutive files together to one storage node. Furthermore, the use of bloom filter for similarity detection between streams has low communication and computation cost while it enables to achieve duplicate elimination performance comparable to single node deduplication. dCACH is evaluated using a prototype deployed on a server environment distributed over four separate machines. It is shown to have 10× the speed of Extreme_Binn with a minimal communication overhead, while its duplicate elimination effectiveness is on a par with a single node deduplication system.

Keywords

Clustered Deduplication, Content Aware Grouping, Hierarchical

Deduplication, Stateful Routing, Similarity Bloom Filters

1. Introduction

By minimizing the amount of data stored on disks and transferred over networks, data deduplication makes it possible to backup huge sets of data with a minimal cost of storage and network bandwidth. It is a technique which keeps one copy of two or more identical data by substituting duplicates with links pointing to a stored copy [1] [2]. Deduplication works by representing files and their parts (chunks) with hash values and comparing these values against hashes of previously processed files and chunks. If a matching hash entry exists then the data the hash represents is a duplicate, otherwise, it is considered unique. Duplicates are replaced with links to old data while data belonging to unique hashes are stored. Fingerprint-based data deduplication involves two major activities. First the file to be deduplicated needs to be chunked and fingerprints representing these chunks need to be generated. And then these fingerprints are compared with other previously stored fingerprints in order to discard duplicates and replace them with links to just one copy. As explained in [1] [3] there are four methods in performing chunking, namely; *Static Chunking* (*SC*), *Content Defined Chunking* (*CDC*), *Whole-File Chunking* (*WFC*) *and Application-specific Chunking*. While SC is used in [4] and Aplication-specific chunking is used in [5] [6] in some form, most works use CDC for the deduplicaiton task.

Often the index structure used to store hashes is too big to be stored in RAM. Hence, it is stored in an on-disk structure leading to a popular problem known as disk index-lookup bottleneck [2]. This issue exasperates furthermore when fine-grained chunking is utilized [7]. For example, for a 1 PB data set, considering an average chunk size of 4 KB and an SHA-1 hashing, there will be around 2.5×10^{11} (250 billion) chunk fingerprints which require 5 TB of storage space. 5 TB is too big to keep in the RAM whereas storing this fingerprint index on disk and accessing it in the deduplication process will introduce the disk index-lookup bottleneck problem. This problem is alleviated using sampling techniques in [8] [9] at a cost of low duplicate elimination performance. While the work in [9] is a D2D scheme and hence is not much scalable, the design in [8] is a distributed deduplication scheme where similarity between files is exploited. In [7] [10] DHT based distributed deduplication is used to reduce the effect of this bottleneck and parallelize as well as scale up deduplication. The level of scalability in both [7] [10] is rigid because they do not offer a mechanism to grow beyond the initial system size as a result of the prefix based mapping being done at the time of system initialization. The work in [8]also faces the same issue because the bin-to-node mapping is done at system initialization and is not flexible.

In addition to the disk index-lookup bottleneck problem mentioned above, deduplication systems might lose their duplicate elimination performance as a result of *storage node island effect* [7] [11] [12]. This issue is peculiar to distributed deduplication systems where multiple nodes perform deduplication but each node is oblivious to data found in other nodes in the system and hence copies of the same data can be found in multiple nodes. The magnitude of storage node island effect on duplicate detection efficiency is minimized using careful allocation of data stream to nodes. In recent distributed deduplication systems stateful routing is used to route data stream to nodes which previously had been assigned data with a similar content [11] [13].

The chance of two files from two different types of applications having similar content or sharing identical parts is shown to be very slim in [5]. This implies that a text file does not share much content with an audio file, or an executable file has a very limited probability of sharing content with an image file. As a result, the duplicate identification and removal effectiveness will not decrease if independent index structures are utilized for each of the file types in a system or if some file types are grouped into distinct groups and the index structure is divided into these groups.

In works like [5] [13] [14] [15] [16] [17], it is observed that much of the storage space in a system is occupied by a very small number of files. This creates an opportunity for deduplication throughput improvement where-by multiple small files are packed into segments and deduplication is performed on these segments rather than the individual files.

Deduplication works like [7] [16] [18] [19] [20] [21] use a mix of file and chunk level deduplication. While this approach eases the disk index-lookup bottleneck problem, its advantage is still questionable if scalability is considered and/or if only one node performs the chunk level deduplication.

dCACH, a content aware clustered and hierarchical deduplication system, creates and exploits data stream locality as well as similarity for better deduplication performance. It achieves better throughput through a combination of hierarchical deduplication, grouping of small files and use of non-monolithic index structure. Its distributed duplicate removal effectiveness is enhanced by stateful routing while it exhibits unlimited scalability as a result of a novel bucket-node-mapped scaling approach. dCACH exploits the negligibly small amount of content shared among chunks from different file types to create groups of files and independent fingerprint index for these groups. The index space space is partitioned into non-overlapping and independent spaces and only one group of file types will be deduplicated using each index space. It also utilizes hierarchical deduplication to ease the disk index-lookup bottleneck problem. The hierarchical deduplication approach first deduplicates files, then segments belonging to unique files are deduplicated. Finally chunks belonging to unique segments are deduplicated. The first two levels of deduplication are distributed exact deduplications while chunk level deduplication is local to each storage node.

dCACH is evaluated in a networked environment where storage nodes are deployed on virtual machines distributed over four separate servers. The duplicate elimination effectiveness of dCACH's stateful routing is compared against three approaches namely; a single node deduplication system, a prototype of stateless distributed deduplication scheme and extreme-binning. For one of the datasets the single node deduplication scheme has a deduplication ratio of 100:5.3 while Extreme_binn and dCACH have 100:7.8 and 100:8.2 deduplication ratios respectively. Throughput wise, dCACH outperforms Extreme_binn by more than 10 times. For one of the datasets, Extreme_binn exhibits a little over 30,000 chunk fingerprints per second while dCACH managed to process over 312,000 chunk fingerprints per second. Considering an average chunk size of 4kb dCACH's throughput translates to ≈ 1.2 GB/s while Extreme_binn's throughput translates to ≈ 118.4 MB/s. We chose to compare our design against the performance of Extreme_binn for two reasons. First, Extreme_binn is a distributed system where a bin-to-node mapping dictates its stateless routing decision. dCACH, on the other hand, utilizes stateful content-based routing. While stateless routing approaches fare well in terms of throughput they are prone to storage node island effect. We wanted to see how well our statefull routing design performs against a stateless routing design. Second, Extreme_binn utilizes sampling to choose a representative chunk ID from chunks of a file for its routing and hence is a similarity based deduplication system. Likewise dCACH exploits similarity using chunk IDs to construct similarity bloom filter for routing decision. While Extreme_binn chooses one chunk ID for each file as a representative, dCACH selects a set of chunk IDs from chunk IDs of multiple files and computes the similarity bloom filter. We want to see how well our design identifies similarities between data streams and removes duplicates compared to Extreme_binns similarity detection and duplicate removal.

Our Contribution

dCACH is in-part inspired by the insignificance of data content shared between files of different application types [5]. We exploit this characteristic to partition flat index structures into independent smaller structures which improves disk index-lookup. We also use bloom filter based stateful routing for our distributed deduplication which yields better deduplication efficiency. Our contributions are:

1) **Content aware grouping**—Sets of groups are created using file's extension when the system is started. Once a file type is assigned to a group it will always be deduplicated in that set of group.

2) **SBFs for stateful data routing**—The use of SBFs (Similarity Bloom Filters) which are constructed from representative chunk fingerprints which themselves are selected from a few number of segments in a data stream.

3) **Seamlessly expandable decentralized storage node management**—Cluster Controller Nodes (CCNs) are introduced into the system anytime the need arises, taking over management of half of storage nodes from an overloaded CCN.

4) **Scalable distributed file and segment global exact deduplication**—File and segment level global exact deduplication is performed at CCNs. Addition of a new CCN does not alter the exact deduplication scheme.

Moreover, we implement the following additional techniques to achieve our goal of high deduplication performance:

1) **Fine grained chunk level deduplication**—Fine grained chunk level deduplication is performed at storage nodes in an inline or offline manner depending on backup window requirement.

2) **Small File Segmenting**—Consecutively appearing small files are packed into segments so that the number of indexes at CCNs is reduced significantly.

The rest of this paper is organized as follows. In the next section, background of deduplication aspects which motivated our work is discussed. dCACH's architecture and design are discussed in Section 3 and Section 4 discusses its implementation. Section 5 presents the result of experiments and analysis as well as comparison of these results. Section 6 assesses researches that are of interest to dCACH. Section 7 concludes the paper.

2. Background and Motivation

In this section, we discuss deduplication approaches with respect to number of nodes involved, method of chunking utilized and techniques used to assign deduplication loads to nodes. We also discuss how each of these issues motivated the design components of dCACH.

2.1. Distributed Deduplication

Data deduplication is performed in distributed setting in [7] [8] [10] [12] [19] while it is implemented on a single node in works like [2] [5] [6] [15] [18]. In distributed deduplication load balance, duplicate elimination effectiveness and network overhead are influenced by routing decisions. In *stateless routing*, a feature value representing the data to be routed is extracted and the application of a simple function (like mod) on this value is used to determine the destination node. This approach is oblivious to the history of previous workloads. In contrast *stateful routing* uses information about the location of previously deduplicated data (chunks) to decide where to put the new data (chunks) but with a higher cost of computation as well as memory and/or communication. In [7] [8] [10] [12] stateless routing is used while [11] [17] [22] [23] implement stateful routing.

Single node deduplication guarantees the removal of all duplicate data. But it suffers from limited scalability and very low throughput as a result of disk index-lookup bottleneck problem [2] [7]. Distributed deduplication, on the other hand, solves these drawbacks while it is susceptible to low duplicate elimination performance as a result of deduplication node information island [11] [12] and also incurs high communication overhead [11] [12] [17]. Moreover, it is susceptible to load imbalance among storage nodes [5] [11] [12].

2.2. Hierarchical Deduplication

In chunk level deduplication, the use of smaller chunk sizes has been proven to produce better storage efficiency but it has high memory cost [2] [24] [25] [26]. This problem exasperates as the size of data being deduplicated grows because the fingerprint index representing this data also grows.

There are many approaches utilized to minimize the size of the fingerprint index while maintaining an acceptable deduplication performance. Some implementations use sampling methods where just a fraction of chunk fingerprints are carefully selected as representatives and are kept in memory [8] [9] [11] [17]. Most systems use two-level deduplication where the first level is file level (course grained) and the second is chunk level (fine-grained) [7] [18] [19] [20] [21]. In the second approach, memory is used to store the entirety of the first level index and just a part of the chunk level index. The second level index as a whole is stored in HDD.

The process of fingerprint lookup is speedup using bloom filters in [2] [7] [27] while [28] [29] use solid state drives to cache and/or store fingerprint indexes.

Two-level hierarchical deduplication where files are first deduplicated and then chunks of only unique files are deduplicated significantly reduces the on-disk index-lookup frequency as it avoids the need for deduplication of chunks belonging to duplicate files. However, it does not tackle the issue where there are files which are considered unique only because a very small fraction of them is modified. In such cases, most part of the file is unchanged from the previous version, yet all of the chunks will be deduplicated wasting precious CPU time and also disk I/O. dCACH solves this problem by using three-level fingerprinting and consequentially, three-level deduplication. First files are deduplicated which is followed by deduplication of segments (highly course grained chunks) belonging to unique files. Finally, fine grained deduplication will be performed only on unique segments.

2.3. Segmenting Small Files

In [5] [14] [15] [16] [17], it is stated that a disproportionately small number of files occupies much of the storage space. This fact is strengthened by our observation in our previous work in [13]. In our work in [13], in one of the data sets, of the 77.3 million files just a little over 15 million of them(20%) occupies 98% of the storage space while 61.8 million files(80%) occupies a mere 2% of the storage space. The 2% logical storage space in that dataset amounts to ≈ 150 GB. This led us to adopt a mechanism where we can reduce the number of file fingerprints sent to the master node by representing n consecutive small file fingerprints with one MD5 value. This is done by inserting the n consecutive small file fingerprints into a character array and then computing the MD5 value of this array. This approach reduces the file fingerprint index at the nodes performing file level deduplication by a significant amount while it adds a little load on the client for computing the representative MD5. If there are a total of t files of

which p% of them are small files then the size of file fingerprint index processed at these nodes will be sum of the number of big files and an n fold fraction of the total number of small files (see Equation (1) below). The drawback of this approach is it results in the transfer of a lot more number of duplicate small files to storage nodes because the representative MD5 will be different even when just one of the n consecutive fingerprints is different or is out of order in its appearance. But, the cumulative effect of this issue will be minor on both throughput as well as efficiency, because the total transferred size of these files is negligibly small and the storage nodes will eliminate these duplicate files.

$$total\_fp\_count = t \times (1 - p) + \frac{t \times p}{n} \tag{1}$$

where, total_fp_count is total file fingerprints processed at nodes, t is total number of fingerprints before small file fingerprints are segmented, p is the percentage of count of small files against all files and n is the number of small files grouped into one segment.

2.4. Content Aware Grouping

Works like [5] [6] [15] [22] [30] perform deduplication using the content of the data (or the type of data) as a means to decide on what type of chunking to apply and where to store chunks. All except [22] and [30] of these works are single node deduplication systems and none of them except [30] utilize grouping to enhance deduplication speed.

3. dCACH Design

In dCACH, deduplication is performed in three stages. The first stage is the deduplication at the Backup Agent (BA) where filtering is performed to make sure that no two identical files or segments are candidates for the deduplication in the second stage. The second stage is a distributed and parallel deduplication scheme which involves multiple Cluster Controller Nodes (CCNs). In this stage, a global exact deduplication is performed on files and segments. The third stage is a fine grained chunk level deduplication which occurs at Storage Nodes (SNs). This stage of deduplication is local to SNs meaning the copy of a chunk might appear in multiple nodes because each node is unaware of the chunks found in other nodes. Hence in our system, we have three-level hierarchical deduplication (File, segment and chunk level) which involves three types of nodes (BAs, CCNs and SNs). Since the global deduplication is performed against all previously seen files and segments, it is considered exact deduplication.

Our system implements inline deduplication at the global level and offline deduplication in storage nodes. Because the global level deduplication removes all duplicate files and segments, the amount of data which are transferred to storage nodes is minimal. As a result, the storage space at storage nodes which is required to buffer incoming data stream prior to its deduplication is very small.

3.1. System Architecture

dCACH is designed in such a way that the nodes performing fingerprint deduplication are not on the path of actual data transfer from backup agents to storage nodes. Figure 1 shows the system layout of dCACH while Figure 2 shows control and data flow. As shown in these two figures it has four major components. It should be noted that the Master Node(MN) and CCNs reside on nodes hosting SNs and hence does not require their own physical nodes even though they can be run on their own if the resource is available.

3.1.1. Backup Agent (BA)

BA is a lightweight application which performs three separate operations. Figure 2 shows the steps followed and the communication procedure between the BA and other types of nodes. BA first performs file, segment and chunk fingerprint generation on the input directory or file using content defined chunking and Rabin fingerprinting algorithm. Second, it uses file extensions to identify the group a file belongs to and inserts its fingerprint into a bucket using the fingerprints prefix. Note that the set of file fingerprints in all of these buckets creates a tanker which represents a batch of data stream whose data content will be routed to storage nodes as a unit. When any of the buckets are full, BA stops inserting into all of them and sorts each bucket independently. It then removes duplicate entries from each bucket and sends the file fingerprints in each bucket to a CCN according to the buckets mapping. Upon receiving the unique files fingerprint list from all CCNs, the agent now inserts segment fingerprints of these unique files into segment buckets which have similar characteristics as that of file buckets and sends them to their respective CCNs. The procedure of deduplicating segment fingerprints is identical to that of file fingerprints both at the backup agent and all CCNs. Once BA receives the list of unique segment fingerprints from all CCNs, it generates SBF from chunks of segments and broadcasts the

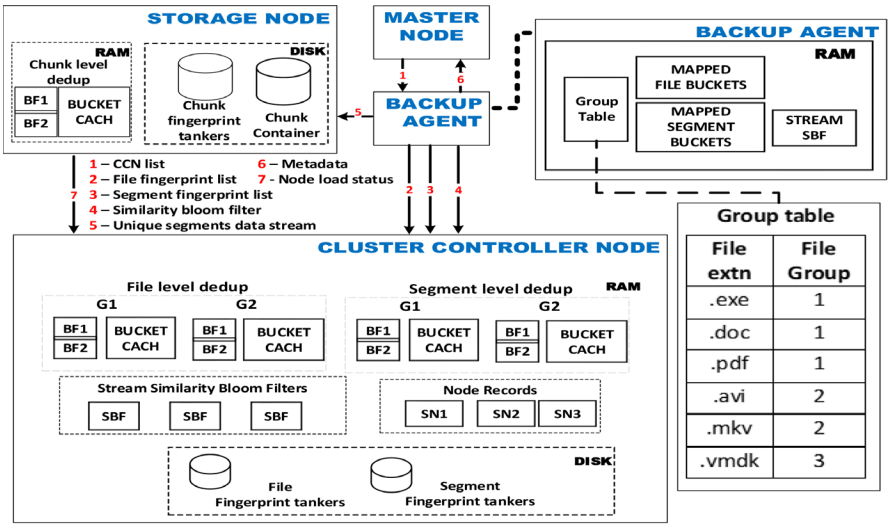

Figure 1. Detailed structure and data flow between backup agent, a single cluster controller and a single storage node under the controller.

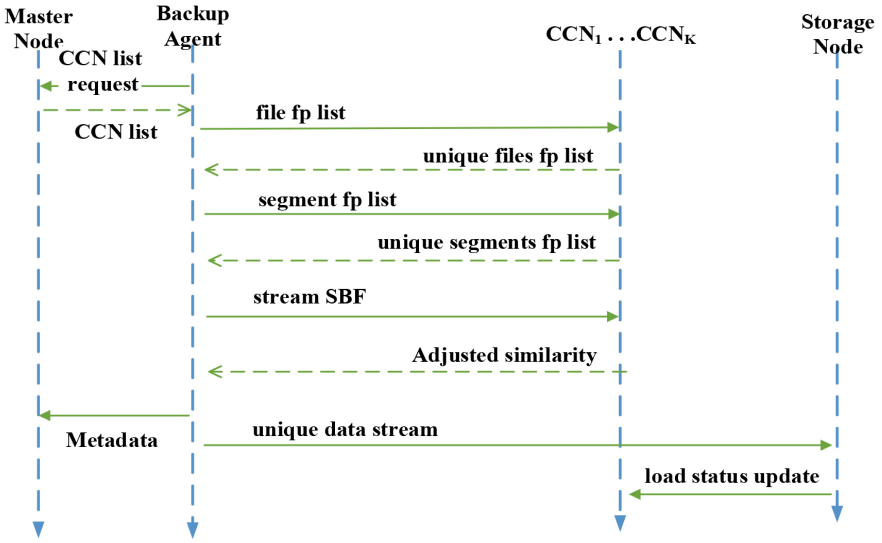

Figure 2. dCACH workflow.

SBF to all CCNs. Next, global adjusted similarity (GAS) is calculated using the response from CCNs. Finally, the agent sends the SBF to the CCN controlling the storage node with the highest GAS for future reference, sends the unique segments data stream to the chosen node and metadata to the MN. Section 4.3 and 4.4 present detailed discussion of how SBF is generated and GAS is calculated respectively.

3.1.2. Cluster Controller Node (CCN)

dCACH contains one or more CCNs which perform global exact file and segment deduplication, SBF comparison and control storage nodes. They can run on a dedicated physical node or on any of the nodes hosting a storage node. The file, segment and chunk level deduplication performed throughout the system is sped up using bloom filters (designated as boxes labeled BF1, BF2 and so on in Figure 1 which supports fast lookup for fingerprints. BF queries which resulted in positive are further checked to rule out false positives by reading buckets of respective tankers from on-disk tankers (file fingerprint tankers, segment fingerprint tankers or chunk fingerprint tankers) into the set-associative buckets designated as BUCKET CACH in the figure. Furthermore, content based deduplication is achieved by allocating independent bloom filter as well as bucket cache for each group. Further readings into how bloom filter array and the bucket cache are designed and implemented can be found in [2] [7]. CCNs compare an incoming SBF against previously recorded SBFs under each node they are managing. The result of this comparison and the load of the nodes is used to compute Adjusted Similarity (AS) and choose the candidate for stream routing. Once a stream is routed to an SN the CCN managing this particular SN will store the streams SBF for future similarity detection and routing. Each SNs load status record is updated occasionally and is used in computing adjusted similarity. The CCN is also responsible to decide if a CCN node split needs to happen based on the availability of computational resources. The number of

CCNs in dCACH can grow as high as the number of file buckets in the backup agent. It is worth noting that the number of storage nodes a CCN controls is independent of the bucket mapping used in the system. Hence, a CCN can control a cluster constituted of any number of storage nodes as long as it has the required RAM and computational resources.

3.1.3. Storage Node (SN)

One or more of SNs form a cluster and are managed by a CCN. Storage nodes perform fine grained chunk level exact deduplication on data stream they receive from the backup agent. Similar to the file and segment level deduplication at CCNs, the chunk level deduplication uses bloom filters to speed up index look up while also utilizing set-associative cache (BUCKET CACH in **Figure 1**) for buckets read from on-disk tankers. Full chunk fingerprint tankers and containers used to store unique chunks data are stored on disk. Moreover, SNs inform their cluster controller of their load status when certain conditions are met. These conditions can be requests from other nodes or every time the size of unique data stored in the node reaches some level. The CCN managing an SN might change when a CCN becomes a bottleneck and triggers a CCN split.

The chunk level deduplication performed at storage nodes uses the same grouping used in the backup agent and CCNs. Hence, each group of data has separate fingerprint index and its associated bloom filter in RAM. The tankers and containers on disk used to store the full fingerprint index and the actual content of data respectively are independent for each group.

3.1.4. Master Node (MN)

MN is a lightweight application responsible for maintaining metadata of files as well as creating and maintaining bucket-CCN mapping.

3.1.5. Workflow

As shown in **Figure 2**, the communication between components of the system starts with the BA requesting for the list of CCNs and the bucket-CCN mapping from the master node (MN). Once the BA receives this list it uses the mapping information to distribute the file fingerprint buckets to their mapped CCNs. It then receives the unique files fingerprint lists and sends segment fingerprint buckets to CCNs using the same mapping as the file fingerprints. After receiving the unique segments fingerprint list it generates SBF and broadcasts it to all CCNs which in turn will respond with adjusted similarity. The BA now computes Global Adjusted Similarity (GAS) and routes the unique data stream to the node with the highest GAS. It also sends metadata to MN. The storage nodes send updates of their load status to their CCNs occasionally which will be used to compute adjusted similarity.

4. System Implementation

4.1. File Grouping

Grouping of file types is implemented in the backup agent to create a batch of

files whose file and segment fingerprints will be queried at CCNs together and content of their unique segments is routed to a single storage node for further chunk level deduplication and storage. This approach has two benefits. First, the batch of file fingerprints and batch of segment fingerprints from unique files are formed from files which are close to each other on the physical storage. Furthermore, each batch constitutes fingerprints from files which are in the same group. This improves the speed of fingerprint lookup because the membership query on the bloom filters is done only on one of the group of bloom filters, the positive bloom filter query results are further checked only on that specific groups' fingerprint index which narrows down the look up from one gigantic index space to just a portion of it. Second, it helps maintain and exploit locality where catching yields better hit rate. It should be noted that the grouping of file types is performed by inspecting all of the files application types and each file types aggregate data sizes from the first full backup request and then using these two parameters to establish a balanced group of file types.

4.2. Data Chunking

dCACH uses content defined chunking and Rabin fingerprinting algorithm [31] to chunk files into course-grained segments with 1 MB, 2 MB and 4 MB as minimum, average and maximum chunk lengths respectively. The segments are further chunked into fine-grained chunks using 2 KB, 4 KB and 8 KB as minimum, average and maximum chunk lengths respectively. MD5 is proven to provide highly collision resistance hashes [32] and is utilized to generate cryptographic hash values of the chunks, segments and files. Throughout this paper we refer to these hash values as fingerprints. It should be noted that other hashing techniques like SHA-1 and SHA-2 can also be utilized.

4.3. SBF Generation and Comparison

As has been stated in previous sections, two prominent challenges in deduplication are that of disk index-lookup bottleneck and communication overhead. Many indigenous methods have been utilized to overcome these challenges. Some used techniques choose a minimal hash from all chunks of a file and use this hash to represent all chunks of the file. This is based on what is commonly known as the *Broder's Theorem*, which states that if two minimal hashes are identical then the files these two hashes belong to are similar [33]. Others used sampling techniques which exploit locality which is the likelihood that if an incoming chunk **C1** is the same as an existing chunk **E1** from previous backups then the next incoming chunks **C2, C3, ⋯ Cn** are highly probably the same as chunks neighbouring **E1**. The first approach is utilized in [8] while the second method is used in [7] [11]. Works like [17]exploit the benefits of both locality based and similarity based solutions.

Broder's Theorem—Consider two sets S1 and S2, with $H(S1) = \{h(xi) | xi \in Si\}$, where h is chosen uniformly and at random from a min-wise independent fami-

ly of permutations.

Let min($H(Si)$) denote the smallest element of $H(Si)$, then

$$\Pr\left[\min\left(H\left(S1\right)\right)=\min\left(H\left(S2\right)\right)\right]=\frac{\left|S1\cap S2\right|}{\left|S1\cup S2\right|} \tag{2}$$

This theory implies that if two representative chunk fingerprints from two different files or sets are identical, then there is a very high probability that the two sets share a lot more fingerprints. As shown in [22] it is possible to have a better estimate of similarity between sets if we increase the number of representative fingerprints from sets but at a cost of more memory and communication overhead. It is with this concept and challenge in mind that dCACH introduces ***SBF***.

In Section 3.1.1 it is described that dCACH uses a batch composed of buckets as a basis for routing. Hence, a batch, depending on the number of buckets and the depth of these buckets, could hold from hundreds to thousands of fingerprints. This poses two big challenges in selecting destination nodes when the conventional use of representative fingerprints are used for stateful routing. Challenge one is the space required to store these representative fingerprints at CCNs. If there are a million batches of streams to be deduplicated and each stream contains a thousand file fingerprints, then assuming an MD5 hash value of 128 bits length (16 bytes), this approach will need 16 × 1000 × 1,000,000 bytes (14.9 GB) of space at these nodes. This requirement grows significantly if representative fingerprints are selected for super-chunks instead of files because a file could contain two or more super-chunks. The second challenge is the communication overhead incurred for transferring the representative fingerprints to the CCNs. Our approach solves these two issues using SBFs. The SBF we generate for routing purpose has similar characteristics as bloom filters used in [2] [7] [20] and it is based on the original idea by [27]. In addition to their use for deduplication, they are also utilized in networking as discussed in [34].

In broad terms bloom filters are probabilistic data structures which provide a fast way to determine if an item is present in a set or not. An important characteristic of bloom filters is that they might falsely say an item is present. The chance of this occurrence is defined as false positive probability.

False positive probability of bloom filters is influenced by factors like their size, number of elements inserted into them, as well as the number of hash functions used to insert elements. Given the size m and member element count n, optimal number of hash functions k can be determined with $k=\left(\frac{m}{n}\right)\times\ln\left(2\right)$. Given a minimal false positive rate f and n we can calculate the required bloom filter space m using Equation (3)

$$m\geq n\times\lg e\times\lg\left(1/f\right) \tag{3}$$

And given lowest false positive rate f, number of hash functions k and size m the optimal number of elements that a bloom filter represents can be determined

using Equation (4)

$$n \le \left(\frac{m}{k}\right) \times \ln(2) \tag{4}$$

The size of bloom filters used to determine similarity in dCACH is accurate enough to tell similarities while it is optimal for both its memory consumption at CCNs and its communication overhead. The SBF generated at the backup agent is representations of chunk fingerprints rather than file or segment fingerprints. It is generated using chunks from a few segments in a data stream batch which is composed of segments of unique files. These segments are first sorted and then chunks belonging to the segments are put into a chunk tanker. Once any one bucket of this chunk tanker is full or chunks of all segments have been put into the chunk tanker, representative chunk fingerprints are chosen and hashed into the SBF. Here, the number of representative chunk fingerprints is set according to Equation (4). If the number of buckets of the chunk tanker is b then each bucket is first sorted and the first $\frac{r}{b}$ chunks are inserted into the SBF. The sorting of both the segment fingerprint list as well as chunk tanker buckets ensures that the generated SBF will capture similarity of two data streams if they contain data streams from similar files. **Algorithm 1** and **Algorithm 2** show the steps followed to select representative chunk fingerprints and SBF generation respectively. At each CCN an incoming SBF is compared with existing SBFs for similarity using jaccard similarity coefficient which measures the distance between them. For two equal sized SBFs SBF1 and SBF2 representing two data streams, their similarity is measured using the number of bit positions which are set to 1 using Equation (5). Since bloom filters are always set to 0 at the time of initialization our SBF comparison approach discards bit positions which are not set to 1. Hence what our comparison does is count the number of bit positions in both SBFs which are set to 1 and divide it by the size of the SBF. The greater the number of common bit positions set to 1 is, the higher their similarity is. This method is also used in [35].

$$Sim_{(SBF1,SBF2)} = \frac{\textbf{SetBitsCount}\left(\textbf{AND}\left(SBF1, SBF2\right)\right)}{\left|SBF1\right|} \tag{5}$$

Algorithm 1 Select Chunks for SBF

Input: Segment fingerprint list **seg_list**
Output: Sorted chunk tanker

```
1: while chunk tanker not full do
2:   for each segment in seg_list do
3:     for each chunk do
4:       Insert chunk fingerprint into chunk_tanker
5:       if chunk_tanker is full then
6:         chunk_tanker=full;
7:       end if
8:     end for
9:   end for
10: end while
11: for each bucket in chunk_tanker do
12:   qsort(bucket)
13: end for
```

Algorithm 2 Generate Stream SBF

Input: sorted chunk_tanker and representative chunks rate
Output: stream SBF

```
1: for each bucket b in chunk_tanker do
2:   while inserted element count n is less than r do
3:     if bucket[b][n].chunk_fp ≠ bucket[b][n+1].chunk_fp then
4:       insert_into_SBF(bucket[b][n].chunk_fp);
5:     else
6:       continue;
7:     end if
8:   end while
9: end for
10: reset(chunk_tanker);
```

where **AND**$(SBF1, SBF2)$ applies a logical AND on SBF1 and SBF2 and **SetBitsCount**( ) counts the number of set bits in the result of the AND operation.

4.4. Stateful Routing Using SBF

A backup stream is composed of unique segments from parts of a single large file or segments from multiple files which belong to the same group but appear close to each other on the physical disk at the source. The backup agent will select chunks from the segments sequentially after the segment list of the stream is sorted. Then the chunks are themselves sorted and the first r of them is inserted into the streams SBF. The value of r is decided when the system is started and is according to Equation (4). Once the SBF is generated it is broadcast to all the CCNs which in turn will compute its similarity with previous SBFs which were used to route streams to the storage nodes. The comparison of SBFs is performed using Equation (5).

SBF similarities above a certain value are considered as actual similarity degrees because even very dissimilar bloom filters can share similar bit positions. In our case we considered SBF similarities above 0.5 (50%) to represent potentially similar streams. In addition to the similarity of the SBFs the CCNs also take the load of nodes into consideration for choosing the candidate storage node. The reason to use the load of nodes for routing decision is because of what we call node popularity issue where a node which has a high number of data streams will be assigned more number of streams because the chance of stream similarity increases with the number of previously assigned streams. Node popularity is also observed and discussed in [11]. Ignoring this issue obviously will lead to better duplicate elimination performance but the system will suffer from great load imbalance. So, in order to create a compromise between SBF matching and load balance, similarity degrees above 0.8 (80%) will be taken as they are while those above 0.5 but below 0.8 will be used to calculate an adjusted similarity (AS) at CCNs using Equation (6). The reason why we use 0.5 and 0.8 as cut off points for degree of similarity can be found in section 1. If no storage node in the CCNs record contains an SBF similar to the current SBF (all similarity levels are below 0.5) a storage node will be chosen randomly and its similarity level is adjusted using its load in a similar fashion as above. Once the CCNs calculate the adjusted similarity, they all send it to the BA accompanying it with the load of

the matching node as well as the average load of all the storage nodes which they are responsible for managing. The BA then uses these values to compute a global adjusted similarity (GAS) using Equation (8) for the candidate storage nodes from each CCN and chooses the one with the highest GAS. If all of them have the same value of GAS, one of the nodes will be chosen randomly.

$$AS = W_S \times S_i + W_L \times \left(1 - \frac{L_i}{CML}\right) \tag{6}$$

$$G_{ML} = \frac{1}{n}\sum_{k=1}^{n} CML_k \tag{7}$$

$$GAS = W_S \times S_i + W_L \times \left(1 - \frac{L_i}{G_{ML}}\right) \tag{8}$$

where: $W_S + W_L = 1$ *and* $0 \le W_S \le 1$ *and* $0 \le W_L \le 1$. W_S represents Similarity Weight, W_L represents Load Weight, S_i represents degree of similarity from node *i*, CML represents Mean Load of nodes in cluster and L_i is the Load of node *i*, *k* is number of CCNs, CML_k is Mean load of cluster k, G_{ML} is global mean load,

4.5. Expandable Cluster Controllers

Segment level deduplication (can be considered an equivalent to super-chunk deduplication) and file level deduplication are distributed over a number of CCNs. These two levels of deduplications are performed across all the CCNs and exhibit a couple of major and unique characteristics. First, they are performed in parallel across all the CCNs which can significantly boost throughput. Second, prefix based indexing makes it possible to make all the CCNs independent from each other while not suffering from deduplication node island effect because they are mapped to non-overlapping buckets in the BA. Third, when a new CCN is introduced into the system it only affects one CCN whose indexes and other deduplication data structures as well as records are split with the new CCN. The introduction of a new CCN also does not introduce dependency between the new CCN and the currently split CCN. This ensures that we can make the system grow seamlessly from a mere one CCN to up to K number of CCNs (where K remains constant and is the number of buckets in the BA). **Figure 3** shows the mapping between buckets in BA and CCNs at different stages of CCN splits. **Algorithm 3** describes the steps used to split a CCN when it becomes the bottleneck.

Deciding which half of a full CCN's data to keep in the parent (full) CCN and which one to assign to the new CCN is straight-forward (**Algorithm 4**). The full CCN always takes the first half of all file buckets and segments of all the currently updateable open tankers in RAM and the full tankers which reside on disk while the second half is assigned to the new CCN. At MN this node split is handled by easily keeping the first half of the original buckets mapped to the currently split CCN while the second half of it is mapped to the new CCN. This

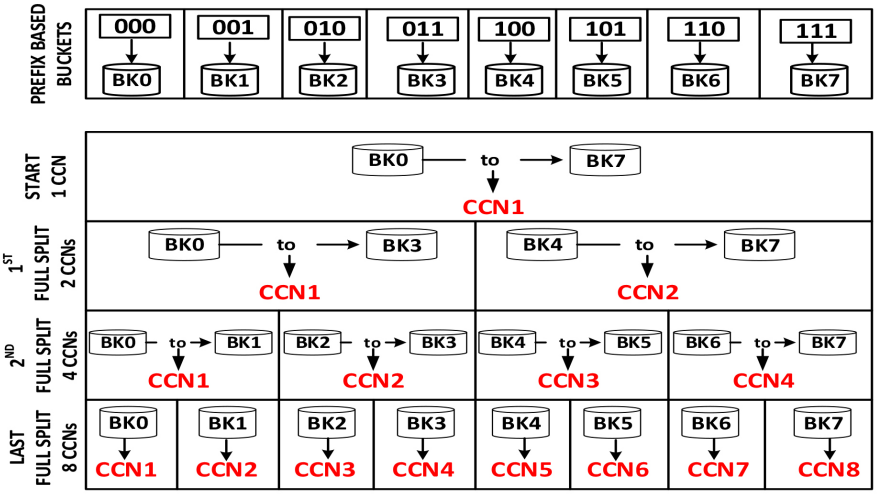

Figure 3. Bucket mapping and CCN expansion.

Algorithm 3 Split full CCN

Input: Full CCN
Output: Split CCN

1: Start new CCN on one of nodes in the cluster which currently have no running CCN
2: Copy all indexes, and node records from the full CCN node to the new CCN(Background task)
3: Start a daemon task to re-distribute half the index entries into a new index structure
4: Re-create a new bloom filter(by the daemon in step 3 and in parallel with step 3) as each index entry is re-arranged
5: Wait until new CCN is finished with its own construction of the index structure and its bloom filter
6: Notify MN of node split
7: Free old index structure, bloom filter and node records
8: Use the new index structure to perform future deduplications

Algorithm 4 Initialize New CCN

Input: new CCN
Output: Initialized CCN

1: Receive data from full CCN
2: Construct index structure and its associated bloom filter from data received from full CCN
3: Notify parent CCN when complete and be ready to accept deduplication requests

guarantees that no matter how many CCN node splits happen, the bucket to CCN mapping will always be correct. At the two CCNs involved in the split the fingerprint entries from half of the old buckets of both of the nodes are redistributed to their respective new buckets by taking n left most bits from fingerprints starting from position $n-1$ where 2^n is the number of CCNs. To demonstrate how this approach works, let's assume that the maximum expected number of CCNs is 8 (2^3) and see in detail what happens in each of the participating nodes. Note that we use 8 buckets in all type of nodes except Storage Nodes[1] and that a CCN identifier after a full split might change in subsequent full splits.

Backup Agent: Since there are 8 buckets and maximum possible number of 2^3 CCNs we always use the first 3 bits of file and segment fingerprints to put them

[1]Backup Agent (BA), Master Node (MN) as well as Cluster Controller Nodes (CCNs) need to have the same number of buckets for files and segments while storage nodes can have a different number of buckets. This is because the bucket-CCN mapping is used only for mapping between buckets in the Backup Agent and CCNs.

in a bucket. Then bucket to CCN mapping is performed depending on the currently active number of CCNs. If there is only one CCN, all buckets will be mapped to it. If there are two CCNs, first half of buckets will be mapped to the first CCN and second half will be mapped to the second. But when there are three CCNs, more needs to be done for mapping because when there are three CCNs (or for that matter when number of CCNs is not in power of 2 (like 3, 5, 6 and 7) it means that one or more of the old CCNs have split but not all have split. In such cases the mapping is simplified by communicating the master node which keeps the record of whether a CCN has split or not. MN maintains which buckets are mapped to which CCN and updates this record every time a CCN node splits. Since we consistently use the left most 3 bits of fingerprints to place them in buckets, we can be assured that a duplicate fingerprint will be found and eliminated by CCNs even after a fingerprint index entry have moved from one CCN to another.

Cluster Controller Node: Deciding which half of a full CCN's data to keep in the parent (full) CCN and which one to assign to the new CCN is straight forward. The full CCN always takes the first half of all file buckets and segments of all the currently update-able open tankers in RAM and the full tankers which reside on disk, while the second half is assigned to the new CCN. The full CCN then transfers a copy of all of its records to the new CCN. This is followed by launching a background task which will reconstruct a new index and bloom filter from first half of its data in the old index. Here, depending on the number of active CCNs (it requests the MN for the number of CCNs after the next full split and uses this data as number of active CCNs), it redistributes its indexes and recreates the associated bloom filter. So, if this CCN is the only CCN in the system the next full split results in two CCNs. The indexes from the first half (the first 4 buckets) of the old buckets will be redistributed using 3 left most bits starting at position 0 (*i.e.* 2^0-1). The new CCN will do the same except that it will redistribute the second half (the last 4 buckets) of the bucket copy it received from the full CCN. On the other hand if the full CCN is one of two CCNs, the next full split will result in 4 CCNs. This means, when redistributing their respective bucket entries both the full and the new CCN will use 3 bits of fingerprints starting from position 1 (*i.e.* 2^1-1). In other words the first bit is ignored because it is the same for all fingerprints in both halves (all 4 buckets) of the old bucket. Node split can continue until maximum possible number of CCNs is arrived at (in this case 8 CCNs). It should be noted that the full CCN initiates background task to handle its node split because the normal deduplication task is not interrupted during node splits and the MN is notified of node split only after both the background task of the full CCN and the construction of indexes at the new CCN are complete. Algorithm 5 presents steps of splitting a full CCN while algorithm 6 shows the steps a new CCN follows after it is initiated.

Master Node: AT MN, node split is handled by easily keeping the first half of the buckets mapped to the currently split CCN while the second half of it is mapped to the new CCN. This guarantees that no matter how many CCN node

splits happen, the bucket to CCN mapping will always be correct. So, when the first CCN splits, the first 4 buckets (buckets 0, 1, 2, 3) are mapped to the full CCN and the last 4 (buckets 4, 5, 6, 7) are mapped to the new CCN. If one of these two nodes splits later, let's say only the second CCN is split, again the first half is kept in the full CCN and the last half mapped to the new one. But if we go one step back and see where the buckets in the original mapping are, we can see that buckets 4, 5, 6 and 7 are split where buckets 4 and 5 are still mapped to the original second CCN while buckets 6 and 7 are mapped to and moved to a new CCN. Note that buckets 0 through 3 are still mapped to original first bucket. Algorithm 7 explains the procedure of creating a new mapping when a node split happens.

Algorithm 5 Bucket-CCN mapping

Input: Number of CCNs N_c, number of buckets N_b, previous splits count P_{sc}, number of CCNs in the previous full split N_{pc}
Output: New bucket mapping

if $P_{sc} == 0$ **then**
  - Map the first N_b/N_c buckets to first CCN node and the next N_b/N_c buckets to the second CCN and so on;
else if $P_{sc}! = N_c$ **then**
  - Map the first half buckets of the node being split to itself and the other half to the new CCN;
else
  - Map the first N_b/N_{cc} buckets to first CCN and the next N_b/N_{cc} to the second and so on; where $N_{cc} = N_{pc} \times 2$;
  - Reset P_{sc} to 0;
end if

5. Evaluation

dCACH is evaluated using a prototype developed for a distributed storage environment. The prototype consists of three components, namely; clients, storage nodes and a master node. The following sections describe the physical experiment environment, the datasets used and the evaluation criteria as well as evaluation result.

5.1. Evaluation Environment

Four separate servers were used to host 17 virtual machines where 16 of these VMs were used to create two clusters of storage nodes with each cluster constituting eight storage nodes. Three of these four servers feature 32 GB RAM, 4 core 2.13 GHz Intel Xeon and 8 disk drives in a RAID-5 setup, while one server features 24 GB RAM, 2 × 8 core 2.40 GHz Intel Xeon and 5 disk drives in a RAID-5 setup. The master node and backup agent were deployed on one machine, while the storage nodes were deployed as guest OSs. Two cluster controller nodes were deployed on two of the 16 virtual machines. The VMs for the storage nodes were configured to have 4 cores of processors with 4 GB of RAM and 600 GB of hard disk.

5.2. Experiment Dataset and Setup

Two data sets are used to evaluate dCACH which are collected from [36]. The

first data set is snapshots of students' home directories (homedir set) from a shared network file system. They consist of 102 snapshots taken from five users over a period of six months from January 22, 2013 to June 04, 2013. The other data set is collection of snapshots taken from a Mac OS X Snow Leopard server running in an academic computer lab. It consists of traces from 44 weekly snapshots spanning 12 months from January 6, 2013 to December 31, 2013. Since these data sets provided pre-processed fingerprints, we were not able to use our CDC approach as described in Section 1 and hence we created segments using consecutively appearing chunk fingerprints by taking into consideration of the chunks length and count to make super-chunks. **Table 1** shows the properties of the datasets.

dCACH is evaluated for throughput, duplicate removal performance, load balancing and network overload. We used a single node deduplication scheme to identify the exact amount of duplicate content on our input data sets. A prototype of Extreme_Binn [8] and a stateless distributed deduplication scheme are used to compare performance against our approach. The stateless deduplication scheme performs inline file level global exact deduplication in a similar fashion as dCACH and offline chunk level deduplication at the storage nodes. The routing method used for this approach is stateless and does not consider storage node load for routing streams. Moreover, no grouping of files is utilized.

5.3. Optimal SBF Parameters

Accuracy of bloom filters is determined by the number of hash functions, number of elements, its size and bound of false positive probability. We tested the accuracy of SBFs representation of data streams by varying the lower bound of the false positive probability. We also checked if SBF representation is affected by the use of multiple buckets for the representative chunks. We set the size m of SBF at 5 KB, the number of hash functions k at 12 and tested the optimal number of representatives to be inserted into the SBF using Equations (3) and (4) by first setting the false positive bound f to 110^6 and then to 110^8. We compared the similarity degree of the bloom filters against the similarity degree of the actual representative fingerprints used to generate the SBFs. That is, if SBF of data stream 107 is 80% similar to SBF of stream 19, we check the number of common representative fingerprints for data streams 107 and 19 as a percentage and compute the gap between the two percentage results. We selected 50 data streams from the homedir data set which showed similarities with previous data streams. As shown in **Figure 4** the gap between actual similarity and SBF similarity goes as high as 39 when the false positive probability bound is set at 110^6 and only 1 bucket is used. The gap becomes as big as 59 when 4 buckets are used. When f is set to 110^8 the gap becomes very small showing better representative count r for the SBF. In this setting the maximum gap observed between the actual similarity degree and the SBF similarity degree is 10 when 1 bucket is used and 9.5 when 4 buckets are used. Thus the number of buckets we used does not

affect the effectiveness of the SBF when f is set to an appropriate value. We chose to use one bucket for representative chunks in order to avoid the computational cost of sorting multiple buckets. We have also observed that the gap between the similarity degrees is bigger for similarities below 0.5 while it becomes narrower as similarity degrees increase. Hence, SBFs with similarity degrees below 0.5 are considered not similar while those above 0.8 are very similar. Accordingly, streams with an SBF matching above 0.8 are routed to the storage node which has the stream of the matching SBF, whereas streams with SBFs with a similarity degree below 0.8 are routed to nodes with consideration of the load of nodes using adjusted similarity as in Equation (6).

5.4. Evaluation Result

5.4.1. Deduplication Throughput (DT)

In Figure 5, dCACH deduplication throughput is the number of chunk fingerprints processed per second. It is calculated using an estimation which considers

Table 1. Property of used datasets.

Data Set	Size (TB)	Num of Files (mlns)	Average Chunk Size (KB)	Average File Size (KB)
homedir	1.73	15.90	4	116.8
macOS	9.80	87.70	4	120

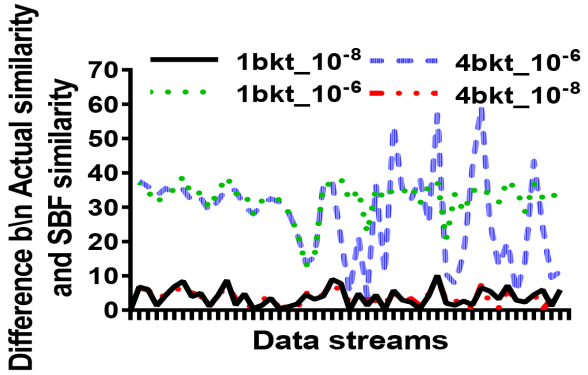

Figure 4. Absolute dierence between SBF Similarity and Representatives Similarity for selected data streams.

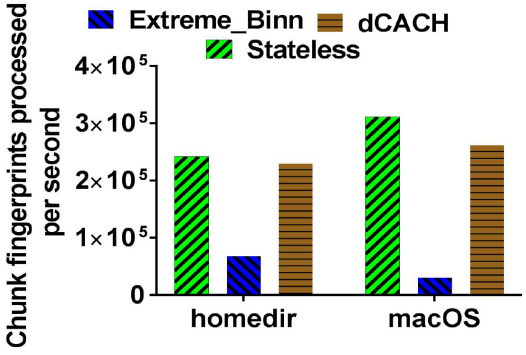

Figure 5. Deduplication Throughput.

the logical data size and Average Chunk Size (ACS) of each data set. Given the logical data size (LDS), deduplication duration time in seconds (D_{sec}) and ACS of a dataset, we use Equation (9) to calculate it.

$$DT = \frac{LDS}{ACS} \times \frac{1}{D_{\text{sec}}} \tag{9}$$

The highest deduplication throughput is observed for the stateless deduplication approach with 2.12×10^5 and 3.12×10^5 for the homedir and macOS datasets respectively. In dCACH, 2.3×10^5 and 2.62×10^5 were processed per second for homedir and macOS datasets respectively. Extreme_binn has the lowest deduplication throughput at 6.89×10^4 and 3.03×10^4 for homedir and macOS datasets respectively. Both dCAH and the stateless approach exhibited more than 10 times the throughput of Extreme_binn.

5.4.2. Load Balancing

In dCACH routing decision for a data stream is performed using similarity of the stream with previously routed streams and the load of nodes holding similar streams. For Extreme_binn routing decision is performed for each file using the file's representative chunk id. In the stateless scheme routing is performed in random fashion.

We measured the load balancing effectiveness of these three approaches using Mean Absolute Deviation (MAD), where the distance of the load of nodes in the system is measured against the average load of nodes and the difference of load between the highest loaded node and the least loaded node. Equation (10) is used to calculate MAD where x_i is load of node i and $m(X)$ is the average load of all n nodes in the system.

$$\frac{1}{n}\sum_{i=1}^{n}\left|x_i - m(X)\right| \tag{10}$$

Among the three distributed deduplication schemes dCACH has the best load distribution while Extreme_binn has the worst. This is evidenced by the gap between the node with the lowest load and the node with highest load. The mean absolute deviation of all nodes against average load also reveals their load distribution performance. For dCACH, the difference of load between the least loaded and highest loaded nodes for the homedir data set is 2.24 GB while its MAD is 0.36. For the same data set the stateless scheme exhibits max-min gap of 6.09 GB and MAD of 1.31 while Extreme_binn's max-min gap is 34.33 GB and its MAD is 4.51. For the macOS data set dCACH has 2.07 GB and 0.42 for max-min gap and MAD respectively while the stateless scheme showed gap of 10.30 GB and MAD of 2.10. Extreme_binn exhibited max-min difference of 9.38 GB and 1.63 MAD. **Figure 6** shows the mean absolute deviation of nodes for the three deduplication approaches.

5.4.3. Duplicate Detection Efficiency (DDE)

DDE is the ratio of size of the original logical data size LDS to the size of the physical data size after deduplication PDS. It is expressed in 11.

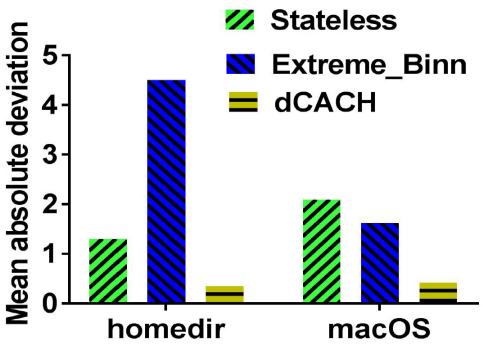

Figure 6. Mean Absolute Deviation of load among all nodes.

$$DDE = \frac{LDS}{PDS} \tag{11}$$

The duplicate removal performance of the various deduplication approaches is shown in **Table 2** as it lists the physical data after deduplication is performed using each approach on the same logical data for each data. **Figure 7** and **Figure 8** show the size of physical data after the four deduplication schemes are performed on homedir and macOS datasets respectively. As expected, the single node deduplication scheme exhibits the highest duplicate elimination performance. The least duplicate elimination performance is observed for the stateless deduplication scheme. From the 1713.8 logical data size of the homedir dataset, dCACH stored 141.1 GB of data as physical data while the single node, Extreme_binn and stateless schemes stored 91.02 GB, 133.83 GB and 159.33 GB respectively. From the 9821.6 GB logical data size of the macOS dataset, dCACH stored only 317 GB while the single node, Extreme_binn and the stateless approaches stored 209.2 GB, 239.66 GB and 373.47GB respectively. **Figure 9** and **Figure 10** show the duplicate removal efficiency in GB and in percent respectively for the homedir dataset while **Figure 11** and **Figure 12** show the duplicate removal efficiency in GB and in percent respectively for the macOS dataset.

5.4.4. Communication Overhead

One big challenge of deduplication over a distributed environment is communication overhead. When fingerprint lookup is performed in a distributed manner over many storage servers the communication cost of the lookup will increase linearly with number of storage servers and exponentially with the number of fingerprints queried. This affects the scalability of deduplication systems. Another form of communication cost is incurred when stateful routing is utilized. In [22] [23], similarity of a new super-chunk is checked with previously routed super-chunks by probing all nodes before it is routed to the node holding the most similar super-chunk. A file could own one to thousands of super-chunks making the communication cost even higher for bigger files. In [8] a single representative chunk fingerprint of a file is used to decide where to route the file for deduplication. This approach does not have cost of communication

Table 2. Deduplication Efficiency of dCACH and three other deduplication approaches.

Deduplication Scheme	Routing Scheme	Data Size in GB			
		Homedir Set		macOS Set	
		Logical	Physical	Logical	Physical
Single Node	None	1713.79	91.02	9821.6	209.2
Extreme Binn	Stateful		133.83		239.66
Stateless	Stateless		159.73		373.47
dCACH	Stateful		144.1		317.01

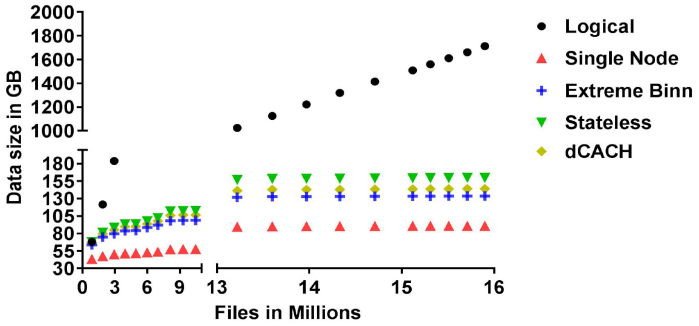

Figure 7. Physical Data Size After Deduplication for homedir dataset.

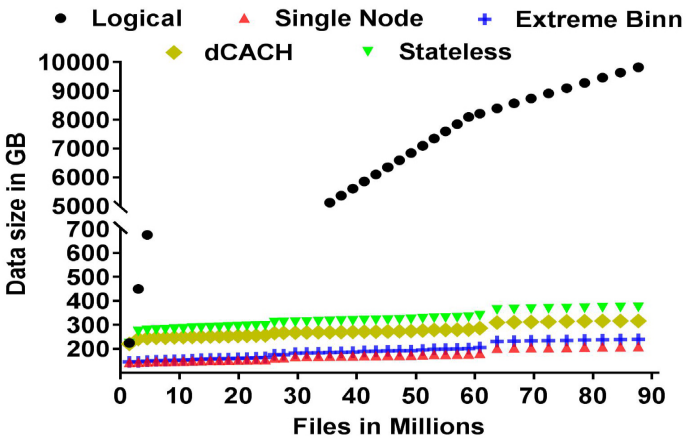

Figure 8. Physical Data Size After Deduplication for macOS dataset.

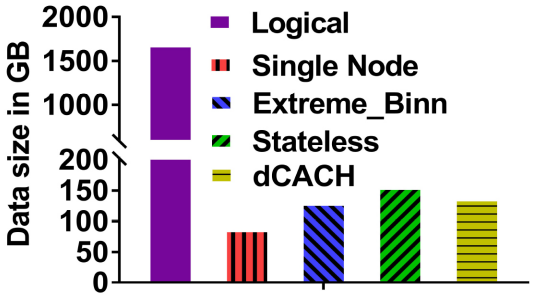

Figure 9. Detected Duplicates at storage nodes in GB for homedir dataset.

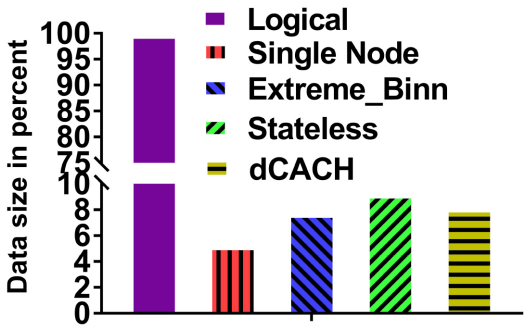

Figure 10. Detected Duplicates at storage nodes in percent for homedir dataset.

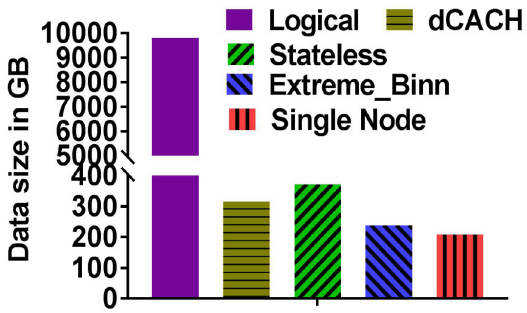

Figure 11. Detected Duplicates at storage nodes in GB for macOS dataset.

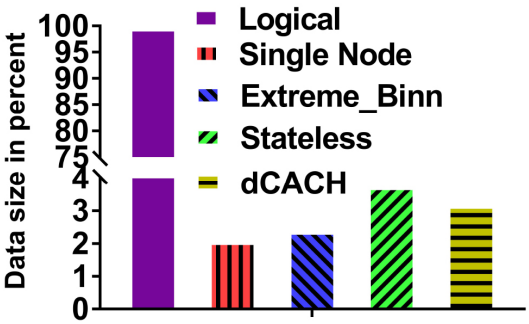

Figure 12. Detected Duplicates at storage nodes in percent for macOS dataset.

for deciding where to route a file but every file will be routed to a deduplicating node whether it is fully duplicate, slightly modified or new file. dCACH implements stateful routing for assigning unique data streams to storage nodes using similarity bloom filters. The SBF is broadcast to all cluster controller nodes and the unique data stream is routed to the storage node with the highest matching SBF. Communication overhead of dCACH's stateful routing is significantly low compared to other approaches because first it happens only between the backup agent and CCNs. Second, an SBF represents tens of thousands to hundreds of thousands of chunk fingerprints depending on the size and number of files and segments belonging to the batch of a data stream.

Because dCACH makes routing decisions on a batch of segments from a single large file or multiple files, the number of lookup is a multiple of these batches. These lookups involve three stages. First is the lookup for duplicate files where a set of file fingerprints are sent to CCNs, then segment lookup for segments belonging to unique files is performed in a similar fashion. Third, lookup for nodes holding a similar batch of data is performed by broadcasting the SBF to all CCNs. For [22] (AppDedupe) the number of lookup is a multiple of super-chunks and number of nodes since each lookup is broadcast to all nodes.

The communication cost of different approaches can be compared using a simple scenario where there are 10,000 files with an average file size of 100 KB and 16 KB average chunk size with 1MB super-chunk size. dCACH with two CCNs needs $(10{,}000/T) \times 3 \times 2$ lookups, where T is number of files in a batch at the backup agent. If T is set to 100, it requires 600 lookups for the deduplication of all the files. It should be noted that the actual number of lookup will be much fewer than this estimate because if all files in a batch are duplicate then there will be no subsequent lookup for segments and similarity comparison for that particular batch of files. For the same scenario [8] needs to communicate the server for each file which is 10,000 whereas in [22] the number of lookups is extremely high at 11,700 × n because all n nodes are involved in routing decision for every 10000*100 KB/1 MB ≈ 11700 super-chunks.

6. Related Works

In our previous work [13], we have used application type of files to create set of file groups and assign each set to specific cluster of nodes with each cluster exclusively deduplicating data streams only from its assigned file group. In single node deduplication schemes like ALG-Dedupe [15] and ADMAD [6], different types of chunking methods were utilized for files of different application types. In HPDV [30], global shared fingerprint index is divided into sub-indexes according to the type of operating systems a virtual machine is hosting. This approach is intended to reduce the scope of fingerprint index search. In AppDedupe [22], the type of files is used to decide where to route its chunks. dCACH creates groups of files based on their application type. Independent index space and its bloom filters are then used for each group throughout all nodes in the system. Moreover, dCACH uses file size to filter small sized files and create file-segments from these small files. The file-segments are used for deduplication instead of the small files. A similar technique of filtering files using their size is utilized in [22] but their intention is only to increase data transfer rate over the network and they completely ignore these small files from the deduplication process. In [9] [11] [12] [22] [37] [38] super-chunks are created from chunks to minimize disk index-lookup bottleneck and exploit data locality. dCACH also exploits locality and improves disk index-lookup performance by first chunking files into segments using CDC and then further chunking these segments into fine grained chunks. Our approach is different from the rest in that it uses CDC to create segments while the other approaches use chunks to create segments

(super-chunks) which does not use CDC.

DHT based distribution of fingerprints and load for deduplication has been implemented in works like [7] [10]. Even though these works have shown that such an approach guarantees exact deduplication and fair amount of work and storage load among nodes, it assumes that the number of storage nodes remain constant. For example in [7], each storage component deduplicates fingerprints with the same specific fingerprint prefix. This enables it to achieve parallel exact deduplication without the storage node island effect. But the number of storage components cannot change or grow because doing so will disrupt the prefix mapping and the system will not be able to identify duplicates against previously deduplicated data. Backup agent of dCACH also uses prefixes to put file and segment fingerprints into non overlapping buckets and these buckets are mapped to currently available cluster controller nodes. This makes it possible to achieve parallel deduplication while avoiding the storage node island effect. But unlike other systems, in dCACH the number of nodes involved in the distributed exact deduplication (CCNs to be explicit) can grow (and also shrink if a special need arrives to downsize). Extreme binning [8] supports the addition of nodes to its system but it results in indexes being reshuffled, and moved around between all the nodes. In this system, in some cases data might be moved around so much that, it might come back to its original host. On the other hand in dCACH, the addition of CCNs results in the movement of indexes and associated records between only two nodes. Our approach also makes it possible to add as any number of storage nodes as required into the system because node management is split between CCNs and will not be a bottleneck.

In [11] [12] [22], a stateful routing is utilized for routing data to minimize the effect of deduplication node information island. In [12] a set of representative handprints are selected from each super-chunk and the similrity of super-chunks is decided by comparing representative handprint chunks. In [11] super-chunks are routed as a unit. Counting bloom filters are used in every node to trace the number of times a fingerprint is stored in the nodes. This approach can achieve a considerable duplicate removal performance, but it is susceptible to high communication cost, high memory consumption and computational overhead. In [22], a two-tired inter-node routing and an application-aware intra-node deduplication are performed to tackle the issue of deduplication node information island effect. In this system, the director-node first selects a group of storage nodes for each file using file type as criteria, the client node then chooses a representative chunk for each super-chunk and broadcasts it to all the selected nodes with the node containing a matching chunk chosen as the destination. The communication cost of this system is high because it floods the network with representative chunk fingerprints for every super-chunk. dCACH introduces a drastic reduction of communication overhead because SBF is utilized. The SBF represents a whole data stream and is sent only to CCNs for similarity computation.

7. Conclusion and Future Work

In this paper, we presented dCACH, a content-aware clustered and hierarchical deduplication system. It employs non-monolithic fingerprint index structures where each partition is used exclusively for a set of file types. This approach boosts fingerprint lookups without loosing duplicate elimination performance because files from different applications share insignificant amount of content. Moreover, it employs decentralized and scalable global exact deduplication for files and segments. The number of nodes responsible for file and segment level deduplication and storage node management can grow without affecting duplicate removal ability. The batch based routing mechanism and use of big sized segments captures and exploits locality. In fact, the probability that the whole content of a file is assigned to different backup nodes is minimal. All parts (chunks) of a file are guaranteed to be assigned to only one node except when the file is very big and is segmented into super chunks where chunks of these super chunks are used for routing decision instead of chunks of the file as a whole. This results in better data management outcomes because a file whose chunks are distributed over multiple nodes is prone to less reliability because failure of any of the nodes hosting its parts will result in loss of the data. Moreover, similarity bloom filters are utilized for stateful routing which results in duplicate elimination rate on a par with single node deduplication with a minimal cost of computation and communication.

Deduplication is shown to cause chunk fragmentation which results in very low read throughput when data restoration is performed. Furthermore, most recent backup versions are more likely to be restored than older versions but their content is the most fragmented. There are two design strategies used so far to deal with fragmentation. The first one is re-writing, where duplicate chunks which are evaluated to cause fragmentation are written again along with unique data [30] [40] [41]. Another approach is the use of optimized caching mechanisms which utilize backup recipes. Since backup's read sequence is available from their backup recipe, works like [42] [43] utilize it to design algorithms which can minimize the number of disk accesses during restoration. Because the recipe of a backup cannot be read into RAM in its entirety, most approaches use a sliding window approach where just a fraction of the recipe is read and its corresponding data is restored and the next part is read and restored and so on. This restore window size influences the caching performance because it determines to what extent the caching algorithm can see the future access pattern. dCACH first divides data into groups according to its content and then routes data according to bucket-node mapping. As a result, a backup version will be split into multiple parts where each part is routed independently. Hence, at storage nodes we have backup recipes of parts which can well fit into RAM. In our future work we plan to design a restore scheme which exploits the small-sized recipes at storage nodes to improve caching mechanisms. We also would like to examine the effect of our similarity-based routing on fragmentation.

Acknowledgments

This work is supported in part by the National Key Research and Development Program of China under Grant No.2016YFB0800402 and the Innovation Group Project of the National Natural Science Foundation of China, No.61821003.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., Fu, M., Zhang, Y. and Zhou, Y. (2016) A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proceedings of the IEEE*, **104**, 1681-1710. https://doi.org/10.1109/JPROC.2016.2571298

[2] Zhu, B., Li, K. and Patterson, H. (2008) Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, USENIX Association, Berkeley, 18:1-18:14.

[3] Meister, D. and Brinkmann, A. (2009) Multi-Level Comparison of Data Deduplication in a Backup Scenario. *Proceedings of SYSTOR* 2009: *The Israeli Experimental Systems Conference*, Haifa, Article No. 8. https://doi.org/10.1145/1534530.1534541

[4] Quinlan, S. and Dorward, S. (2002) Venti: A New Approach to Archival Data Storage. *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, USENIX Association, Berkeley, 89-101.

[5] Fu, Y., Jiang, H., Xiao, N., Tian, L. and Liu, F. (2011) AA-Dedupe: An Application-Aware Source Deduplication Approach for Cloud Backup Services in the Personal Computing Environment. *IEEE International Conference on Cluster Computing*, Austin, 26-30 September 2011, 112-120. https://doi.org/10.1109/CLUSTER.2011.20

[6] Liu, C., Lu, Y., Shi, C., Lu, G., Du, D.H.C. and Wang, D.-S. (2008) ADMAD: Application-Driven Metadata Aware De-Duplication Archival Storage System. *5th IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, Computer Society Press, Baltimore, 29-35. https://doi.org/10.1109/SNAPI.2008.11

[7] Wei, J., Jiang, H., Zhou, K. and Feng, D. (2010) MAD2: A Scalable High-Throughput Exact Deduplication Approach for Network Backup Services. *IEEE 26th Symposium on Mass Storage Systems and Technologies*, Lake Tahoe, 6-7 May 2010, 1-14. https://doi.org/10.1109/MSST.2010.5496987

[8] Bhagwat, D., Eshghi, K., Long, D. and Lillibridge, M. (2009) Extreme Binning: Scalable, Parallel Deduplication for Chunk-Based File Backup. *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, London, 21-23 September 2009, 1-9. https://doi.org/10.1109/MASCOT.2009.5366623

[9] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G. and Camble, P. (2009) Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. *Proceedings of the 7th Conference on File and Storage Technologies*, USENIX Association, Berkeley, 111-123.

[10] Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Un-gureanu, C. and Welnicki, M. (2009) HYDRAstor: A Scalable Sec-

ondary Storage. *Proceedings of the 7th Conference on File and Storage Technologies*, USENIX Association, Berkeley, 197-210.

[11] Dong, W., Douglis, F., Li, K., Patterson, H., Reddy, S. and Shilane, P. (2011) Tradeoffs in Scalable Data Routing for Deduplication Clusters. *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, USENIX Association, Berkeley, 15-29.

[12] Fu, Y., Jiang, H. and Xiao, N. (2012) A Scalable Inline Cluster Deduplication Framework for Big Data Protection. *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, Montreal, 3-7 December 2012, 354-373. https://doi.org/10.1007/978-3-642-35170-9_18

[13] Dagnaw, G., Hua, W. and Zhou, K. (2018) CACH-Dedup: Content Aware Clustered and Hierarchical Deduplication. 24*th IEEE International Conference on Parallel and Distributed Systems*, Singapore, 11-13 December 2018, 399-407. https://doi.org/10.1109/PADSW.2018.8644884

[14] Agrawal, N., Bolosky, W.J., Douceur, J.R. and Lorch, J.R. (2007) A Five-Year Study of File-System Metadata. *ACM Transactions on Storage*, **3**, 9. https://doi.org/10.1145/1288783.1288788

[15] Fu, Y., Jiang, H., Xiao, N., Tian, L., Liu, F. and Xu, L. (2014) Application-Aware Local-Global Source Deduplication for Cloud Backup Services of Personal Storage. *IEEE Transactions on Parallel and Distributed Systems*, **25**, 1155-1165. https://doi.org/10.1109/TPDS.2013.167

[16] Tan, Y., Jiang, H., Feng, D., Tian, L., Yan, Z. and Zhou, G. (2010) Sam: A Semantic-Aware Multi-Tiered Source De-Duplication Framework for Cloud Backup. 39*th International Conference on Parallel Processing*, San Diego, 13-16 September 2010, 614-623. https://doi.org/10.1109/ICPP.2010.69

[17] Xia, W., Jiang, H., Feng, D. and Hua, Y. (2011) Silo: A Similarity-Locality Based Near-Exact Deduplication Scheme with Low Ram Overhead and High Throughput. *Proceedings of USENIX ATC*, Portland, 15 June 2011, 26-28.

[18] Li, Y.-K., Xu, M., Ng, C.-H. and Lee, P.P.C. (2014) Efficient Hybrid Inline and Out-of-Line Deduplication for Backup Storage. *ACM Transactions on Storage*, **11**, 1-21. https://doi.org/10.1145/2641572

[19] Yang, T., Jiang, H., Feng, D., Niu, Z., Zhou, K. and Wan, Y. (2010) DEBAR: A Scalable High-Performance Deduplication Storage System for Backup and Archiving. *IEEE International Symposium on Parallel & Distributed Processing*, Istanbul, 7-9 July 2010, 1-12. https://doi.org/10.1109/IPDPS.2010.5470468

[20] Zhang, J., Zhang, S., Lu, Y., Zhang, X. and Wu, S. (2013) Hierarchical Data Deduplication Technology Based on Bloom Filter Array. Lecture Notes in Electrical Engineering, *Proceedings of the International Conference on Information Engineering and Applications*, **1**, 725-732. https://doi.org/10.1007/978-1-4471-4856-2_88

[21] Zhou, Y., Feng, D., Xia, W., Fu, M., Huang, F., Zhang, Y. and Li, C. (2015) SecDep: A User-Aware Efficient Fine-Grained Secure Deduplication Scheme with Multi-Level Key Management. 31*st Symposium on Mass Storage Systems and Technologies*, Santa Clara, 1-5 June 2015, 1-14. https://doi.org/10.1109/MSST.2015.7208297

[22] Fu, Y., Xiao, N., Jiang, H., Hu, G. and Chen, W. (2017) Application-Aware Big Data Deduplication in Cloud Environment. *IEEE Transactions on Cloud Computing*, 1. https://doi.org/10.1109/TCC.2017.2710043

[23] Luo, S., Zhang, G., Wu, C., Khan, S. and Li, K. (2015) Boafft: Distributed Deduplli-

cation for Big Data Storage in the Cloud. *IEEE Transactions on Cloud Computing*, 1. https://doi.org/10.1109/TCC.2015.2511752

[24] Kulkarni, P., Douglis, F., LaVoie, J. and Tracey, J.M. (2004) Redundancy Elimination within Large Collections of Files. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, 59-72.

[25] Muthitacharoen, A., Chen, B., *et al.* (2001) A Low-Bandwidth Network File System. *Proceedings of the* 18*th ACM Symposium on Operating Systems Principles*, Banff, 21-24 October 2001, 174-187. https://doi.org/10.1145/502034.502052

[26] Policroniades, C. and Pratt, I. (2004) Alternatives for Detecting Redundancy in Storage Systems Data. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, 73-86.

[27] Bloom, B.H. (1970) Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, **13**, 422-426. https://doi.org/10.1145/362686.362692

[28] Lu, G., Debnath, B. and Du, D.H. (2011) A Forest-Structured Bloom Filter with Ash Memory. *IEEE* 27*th Symposium on Mass Storage Systems and Technologies*, Denver, 23-27 May 2011, 1-6.

[29] Wang, J., Zhao, Z., Xu, Z., Zhang, H., Li, L. and Guo, Y. (2015) I-Sieve: An Inline High Performance Deduplication System Used in Cloud Storage. *Tsinghua Science and Technology*, **20**, 17-27. https://doi.org/10.1109/TST.2015.7040510

[30] Lin, C., Cao, Q., Huang, J., Yao, J., Li, X. and Xie, C. (2018) HPDV: A Highly Parallel Deduplication Cluster for Virtual Machine Images. 18*th IEEE/ACM International Symposium on Cluster*, *Cloud and Grid Computing*, Washington DC, 1-4 May 2018, 472-481. https://doi.org/10.1109/CCGRID.2018.00074

[31] Rabin, M.O. (1981) Fingerprinting by Random Polynomials. Tech. Rep., Center of Research in Computer Technology, Technical Report.

[32] Rivest, R. (1992) The md5 Message-Digest Algorithm. https://doi.org/10.17487/rfc1321

[33] Broder, A.Z., Charikar, M., Frieze, A.M. and Mitzenmacher, M. (1998) Min-Wise Independent Permutations. *Journal of Computer and System Sciences*, **60**, 327-336.

[34] Broder, A. and Mitzenmacher, M. (2004) Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, **1**, 485-509. https://doi.org/10.1080/15427951.2004.10129096

[35] Donnet, B., Gueye, B. and Kaafar, M.A. (2012) Path Similarity Evaluation Using Bloom Filters. *Computer Networks*, **56**, 858-869. https://doi.org/10.1016/j.comnet.2011.11.003

[36] FSL (2014) Traces and Snapshots Public Archive. http://tracer._lesystems.org

[37] Xia, W., Jiang, H., Feng, D. and Tian, L. (2016) DARE: A Deduplication-Aware Resemblance Detection and Elimination Scheme for Data Reduction with Low Overheads. *IEEE Transactions on Computers*, **65**, 1692-1705. https://doi.org/10.1109/TC.2015.2456015

[38] Zhang, P., Huang, P., He, X., Wang, H., Yan, L. and Zhou, K. (2016) RMD: A Resemblance and Mergence Based Approach for High Performance Deduplication. 45*th International Conference on Parallel Processing*, Philadelphia, 16-19 August 2016, 536-541. https://doi.org/10.1109/ICPP.2016.68

[39] Fu, M., Feng, D., Hua, Y., He, X., Chen, Z., Xia, W., Huang, F. and Liu, Q. (2014) Accelerating Restore and Garbage Collection in Deduplication-Based Backup Systems via Exploiting Historical Information. *USENIX Annual Technical Conference*, Philadelphia, 19-20 June 2014, 181-192.

[40] Tan, Y., Wang, B., Wen, J., Yan, Z., Jiang, H. and Srisa-an, W. (2018) Improving Restore Performance in Deduplication-Based Backup Systems via a Fine-Grained Defragmentation Approach. *IEEE Transactions on Parallel and Distributed Systems*, **29**, 2254-2267. https://doi.org/10.1109/TPDS.2018.2828842

[41] Wu, J., Hua, Y., Zuo, P. and Sun, Y. (2018) Improving Restore Performance in Deduplication Systems via a Cost-Efficient Rewriting Scheme. *IEEE Transactions on Parallel and Distributed Systems*, **30**, 119-132. https://doi.org/10.1109/TPDS.2018.2852642

[42] Cao, Z., Wen, H., Wu, F. and Du, D.H. (2018) ALACC: Accelerating Restore Performance of Data Deduplication Systems Using Adaptive Look-Ahead Window Assisted Chunk Caching. 16*th fUSENIXg Conference on File and Storage Technologies*, Oakland, 12-15 February 2018, 309-324.

[43] Kaczmarczyk, M. and Dubnicki, C. (2015) Reducing Fragmentation Impact with Forward Knowledge in Backup Systems with Deduplication. In: *Proceedings of the* 8*th ACM International Systems and Storage Conference*, ACM, New York, 17. https://doi.org/10.1145/2757667.2757678

Appendix A

Table A1 presents most frequently used acronyms and terms used throughout the paper.

Table A1. Acronyms and frequently used terms.

Acronym (Term)	Definition
BA	Backup Agent
CCN	Cluster Controller Node
MN	Master Node
SN	Storage Node
SBF	Similarity Bloom Filer
AS	Adjusted Similarity—Generated by CCNs
GAS	Global Adjusted Similarity— Computed at BA using ASs from CCNs and node loads
G_{ML}	Global Mean Load—Average Load of all storage nodes in the system
CCN Split	Migration of half of load of a CCN to another new CCN
Data Stream	Stream of data content belonging to unique segments of unique files from a batch of files