Boafft: Distributed Deduplication for Big Data Storage in the Cloud

Shengmei Luo, Guangyan Zhang, Chengwen Wu, Samee U. Khan, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—As data progressively grows within data centers, the cloud storage systems continuously facechallenges in saving storage capacity and providing capabilities necessary to move big data within an acceptable time frame. In this paper, we present the Boafft, a cloud storage system with distributed deduplication. The Boafft achieves scalable throughput and capacity usingmultiple data servers to deduplicate data in parallel, with a minimal loss of deduplication ratio. Firstly, the Boafft uses an efficient data routing algorithm based on data similarity that reduces the network overhead by quickly identifying the storage location. Secondly, the Boafft maintains an in-memory similarity indexing in each data server that helps avoid a large number of random disk reads and writes, which in turn accelerates local data deduplication. Thirdly, the Boafft constructs hot fingerprint cache in each data server based on access frequency, so as to improve the data deduplication ratio. Our comparative analysis with EMC's stateful routing algorithm reveals that the Boafft can provide a comparatively high deduplication ratio with a low network bandwidth overhead. Moreover, the Boafft makes better usage of the storage space, with higher read/write bandwidth and good load balance.

Index Terms—Big data, cloud storage, data deduplication, data routing, file system

1 Introduction

Currently, the enterprise data centers manage PB or even EB magnitude of data. The data in those cloud storage systems (e.g., GFS [1], HDFS [2], Ceph [3], Eucalyptus [4], and GlusterFS [5]) that provide a large number of users with storage services are even larger. The cost of enterprise data storage and management is increasing rapidly, and the improvement of storage resource utilization has become a grand challenge, which we face in the field of big data storage. According to a survey, about 75 percent data in the digital world are identical [6], and especially the data redundancy in backup and archival storage systems is greater than 90 percent [7]. The technique of *data deduplication* can identify and eliminate duplicate data in a storage system. Consequently, the introduction of data deduplication into cloud storage systems brings an ability to effectively reduce the storage requirement of big data and lower the cost of data storage.

Data deduplication replaces identical regions of data (files or portions of files) with references to data already stored on the disk. Compared with the traditional compression techniques, data deduplication can eliminate not only the data redundancy within a single file, but also the data redundancy among multiple files. However, to find redundant data blocks, deduplication has to make content comparison among a large amount of data. Deduplication is both computation-intensive and I/O-intensive, which easily has a negative impact on the performance of data servers. To reduce this negative impact of data deduplication, an attractive approach is to implement it in parallel by distributing the computational and I/O tasks to individual nodes in a storage cluster. This can utilize the computation capability and storage capacity of multiple nodes in cloud storage to solve the bottleneck of data deduplication.

One of technical challenges with regards to distributed data deduplication is to achieve scalable throughput and a system-wide data reduction ratio close to that of a centralized deduplication system. By querying and comparing the entire data globally, we can achieve the best data deduplication ratio (DR). However, it is required to maintain a global index library. Both index data updates and duplicate data detection will cause network transmission overheads. Therefore, such a global deduplication will have a severe performance degradation, especially in a cloud storage system with hundreds of nodes. An alternative approach is a combination of content-aware data routing and local deduplication. When using this approach, one will face the challenge of designing a data routing algorithm with low computing complexity and high deduplication ratio.

In this paper, we present Boafft,[1] a cloud storage system with distributed deduplication. Boafft creates super-chunks that represent consecutive smaller data chunks, then routes super-chunks to nodes according to data content,

- *S. Luo, G. Zhang, and C. Wu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: luo.shengmei@zte.com.cn, gyzh@tsinghua.edu.cn, wcw14@mails.tsinghua.edu.cn.*
- *S.U. Khan is with the Department of Electrical & Computer Engineering, Mississippi State University, Starkville, MS 39762 USA. E-mail: skhan@ece.msstate.edu.*
- *K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.*

Manuscript received 15 Jan. 2015; revised 10 Aug. 2015; accepted 10 Dec. 2015. Date of publication 23 Dec. 2015; date of current version 3 Dec. 2020. Recommended for acceptance by J. Chen, I. Stojmenovic, and I. Bojanova. Digital Object Identifier no. 10.1109/TCC.2015.2511752

1. an abbreviation of "Birds of a feather flock together".

and finally performs local deduplication at each node. Boafft uses an efficient data routing algorithm based on data similarity that reduces the overhead of network bandwidth and calculates the storage location of data quickly. For a data set, multiple representative fingerprints, instead of only one fingerprint, are used to find more similarity in datasets. The data routing algorithm leverages data similarity to enable superblocks with high similarity to co-locate in the same data server, where Boafft performs local data deduplication. This provides a minimal loss of deduplication ratio, while reducing the network overhead.

Moreover, Boafft accelerates data routing and local data deduplication by two key technologies. First, Boafft constructs hot fingerprint cache in each data server based on access frequency, which accelerates data routing and guarantees data deduplication ratio by leverage of data locality. Second, Boafft maintains an in-memory similarity index table in each data server that aids in avoiding a large number of random disk reads and writes, and in turn accelerates local data deduplication.

We implement the Boafft prototype by modifying the source code of the widely-used Hadoop distributed file system (HDFS). Our results from detailed experiments using data center traces show that Boafft achieves scalable I/O throughput using multiple storage nodes to deduplicate in parallel, with a minimal loss of deduplication ratio. Compared with the EMC's stateful routing algorithm, the Boafft can provide a comparatively high deduplication ratio with a low network bandwidth overhead. Moreover, the Boafft can achieve better storage space utilization, with higher read/write bandwidth and good load balance.

The rest of this paper is organized as follows. Section 2 discusses related work, then Section 3 describes an overview of our Boafft system. Section 4 focuses on data routing for distributed deduplication, and Section 5 depicts how local deduplication is performed within a single data server. We present our experimental methodology, datasets, and the corresponding experimental results in Section 6. Finally, conclusions are presented in Section 7.

2 Related Work

Commonly used lossless compression algorithms include Huffman coding [8], Lempel_ziv [9], and Range encoding [10]. For example, the design of DEFLATE [11] algorithm that was applied in the Gzip compression software, is based on the two lossless compression algorithms of Lempel and Huffman.

For data deduplication, the granularity is the key factor that determines the deduplication ratio. Currently, there are four main granularities in deduplication, namely: file, data block, byte, and bit. File granularity deduplication [12] systems, such as FarSite system [13] and EMC Center System [14] perform data deduplication by judging whether the whole file is identical. File granularity deduplication can only deduplicate when the files are identical, but cannot deduplicate redundant data blocks within a file. Although deduplication with data block, byte, or bit granularity can deduplicate within a file, it does utilize more system resources.

In terms of data block partitioning, there are two main techniques, namely: fixed-sized partitioning [15] and content-defined chunking [15]. OceanStore [16] and Venti [17] are based on fixed-sized partitioning. Content-defined chunking, which is a content based method, partitions data into data blocks with variable size by using sliding window and Rabin's fingerprint algorithm [18]. Consequently, it can achieve a higher deduplication ratio, which is the primary reason for its widespread adoption in many systems, such as LBFS [19], Pastiche [20], Extreme Binning [21], EMC Cluster Deduplication [22], and Deep Store [23]. Apart from the aforementioned algorithms, some researchers proposed many other partitioning algorithms based on the features of dataset, such as TTTD [24], ADMAD [25], Fingerdiff [26], and Bimodal CDC [27].

In the process of deduplication, the judgment of redundant data blocks is based on the search and match of fingerprints. Therefore, the optimization of indexing and querying is an effective way to improve the I/O performance and reduce the bottleneck of disk search in deduplication systems. There are three main methods to optimize the data block index. The first method is the optimization strategy based on data locality. For example, in the design of DDFS [28], Zhu et al. proposed summary vector, stream-informed segment layout, and locality-preserved caching, which are based on locality to reduce the number of disk I/O and improve cache hit ratio to optimize the process of data block index and query. However, with the increase of the scale of storage, a lot of system resources are required. Therefore, the methodology tends to be used in a one-node system. The second method is based on data similarity. For example, DDFS [28] uses the technique of *Bloom Filter* [29] to reduce the size of data index table. Lillibridge et al. proposed sparse indexing [30], which samples data blocks based on similarity to reduce the amount of data to be indexed and queried, and then deduplicates those data segments with higher similarity. The HP's extreme binning strategy [21] is also based on data similarity that eliminates the bottleneck of disk in the process of index querying. Based on Broder's theory of min-wise independent permutations [31], it determines weather two files are similar by comparing their minimum fingerprint. This method uses a two-level indexing, so each data block query only costs a disk access, which reduces the number of index query. The third method is based on the SSD's index. Due to better performance of the SSDs on random reads, storing the index of fingerprints and file's Metadata in the SSD can accelerate the query. For example, Microsoft Research proposed ChunkStash [32] that stores data block's index within the SSD to improve system throughput.

The construction of a large-scale, high performance, distributed deduplication system needs to take various factors into consideration, such as system's global deduplication, single node's throughput, data distribution, and scalability. It is noteworth to mention that the system overhead, deduplication ratio, and scalability are all interdependent as well. The EMC's Data Domain [33] global deduplication array has a good scalability in a small-scale cluster, but the deduplication ratio, I/O throughput, and communication overheads are the major drawbacks. Nippon Electric Company's (NEC) HYDRAstor [34] distributes data by distributed hash tables (DHT). The HYDRAstor distributes data blocks to different virtual super node according to the fingerprint. Thereafter, data deduplicates in each of the virtual super node. Consequently, it can scale out a storage

system quickly. However, it cannot maintain data locality due to the size of the 64 KB data block granularity.

Extreme binning utilizes file similarity to perform stateless routing. First, it selects the minimum fingerprint in the file as its characteristic fingerprint, according to Broder's theory of min-wise independent permutations. Thereafter, it routes the files that are similar to the same deduplication server to deduplicate. Although extreme binning can keep its system performance and scalability in a large-scale cluster, since its granularity is file, it can only perform well when data locality is good. Dong et al. [22] proposed a solution for high performance deduplication cluster, which takes super-chunk as its granularity to improve overall system routing efficiency. Besides, they presented stateless and stateful routing strategies based on super-chunk. Stateless strategy uses conventional DHT to route super-chunk, which has good load balance in a small cluster. But in a large-scale cluster, it is hard to keep load balance, and its deduplication ratio is comparatively low. While stateful strategy matches the super-chunk's fingerprint with stored fingerprints in all nodes according to the index table, with the consideration of load balance, it determines the route with a good deduplication ratio in the node. The stateful strategy can avoid imbalance and achieve a good deduplication performance, but it has an increased cost in computation, memory and communication, especially when the storage scale grows rapidly.

Frey et al. proposed a probabilistic similarity metric [35] that identifies the nodes holding the most chunks in common with the superblock being stored. By introducing this metric, the computational and memory overheads of stateful routing at the superblock granularity can be minimized. As some workloads express poor similarity and some others may have poor locality, some systems (e.g., DDFS, ChunkStash) can only perform well when workloads exhibit good locality, and others (e.g., Extreme Binning) can do well only when workloads have good similarity. Based on this observation, SoLi [36] exploits both similarity (by grouping strongly correlated small files into a segment and segmenting large files) and locality (by grouping contiguous segments into blocks) in backup streams to achieve near-exact deduplication. However, it only addresses the intra-node challenge of single deduplication server. $\sum$-Dedupe [37] leverages data similarity and locality to make a sensible tradeoff between high deduplication ratio and high performance scalability for cluster deduplication. Some other deduplication systems focus on offline deduplication, such as DEBAR [38] and ChunkFarm [39]. These two systems split data partitioning and signature calculations from the global index lookup, and update operations in parallel, and batch access to the index.

3 Overview of the Boafft

The Boafft is a cluster-based deduplication system that is built on a distributed storage system, where each data server has not only storage capacity but also certain computational capability. First, to better utilize the cluster's capability, data servers perform local data deduplication in parallel that guarantees overall system performance and storage bandwidth in cloud storage environments. Second, each client sends those data with high data similarity to the same data server by using an efficient routing algorithm based on data similarity, which ensures the high global deduplication ratio. Finally, we optimize the storage of fingerprint index, reduce the index query overhead, and achieve a good deduplication ratio in a single node, by implementing similarity index querying, storage container caching, and hot fingerprint cache.

3.1 Theoretical Basis

Boafft uses *MinHash* (or the min-wise independent permutations locality sensitive hashing scheme) [31], [41] to quickly estimate how similar two superblocks are. The *Jaccard similarity coefficient* [40] is a commonly used indicator of the similarity between two sets. As shown in Equation (1), the Jaccard similarity coefficient is defined as the size of the intersection divided by the size of the union of the sample sets.

$$sim(A,B) = J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}. \quad (1)$$

According to min-wise independent permutations, we get Equation (2). Here, Let h be a hash function that maps the members of A and B to distinct integers, and for any set S define $h_{min}(S)$ to be the member x of S with the minimum value of $h(x)$. That is, the probability that $h_{min}(A) = h_{min}(B)$ is true is equal to the similarity $J(A,B)$, assuming randomly chosen sets A and B.

$$Pr[h_{min}(A) = h_{min}(B)] = J(A,B). \quad (2)$$

Then, we have a result as expressed in Equation (3).

$$sim(A,B) = Pr[h_{min}(A) = h_{min}(B)]. \quad (3)$$

If r is the random variable that is one when $h_{min}(A) = h_{min}(B)$ and zero otherwise, then r is an unbiased estimator of $J(A,B)$. r is too high a variance to be a useful estimator for the Jaccard similarity on its own—it is always zero or one. The idea of the *MinHash* scheme is to reduce this variance by averaging together several variables constructed in the same way. As a result, we use the k representative fingerprints of a data set, by calculating minimal fingerprints of its k data subsets, to find more similarity in datasets. In Section 6.3.1, we discuss the effect of different k, and how to select k in reality.

3.2 System Architecture

Fig. 1 demonstrates the architecture design of Boafft. Logically, the system is composed of clients, a metadata server and data servers. A out-of-band distributed file system is built on those data servers and the metadata server. Clients interact to those servers via network connection to perform data deduplication.

The main function of clients is to provide interactive interfaces. Clients perform data preprocessing for data deduplication, e.g., data block partitioning, fingerprint extraction, organization of data into superblocks. Clients get the routing address of a superblock by interacting with the data servers in the distributed file system, and then send the data and the corresponding fingerprints to the selected data server.

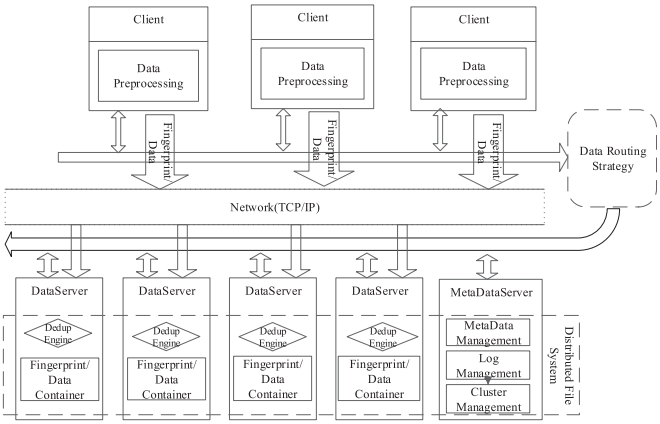


Fig. 1. Architecture of the Boafft system.

The metadata server mainly handles the storage of metadata, the management and maintenance of the cluster. More specifically, it manages the process of data storage, maintains and manages the whole metadata in the distributed file system and its storage status, guides the data routing and maintains system load balance.

A data server performs local data deduplication, and the storage and the management of data. A data server communicates with clients via network to update the status of data reception and node's storage asynchronously. When receiving a write request, a data server is responsible for receiving and deduplicating the data within the node, and constructing corresponding index association of the fingerprint. In addition to that, a data server needs to build a connection between fingerprint and data blocks, index data block's fingerprint, map the file and data blocks, and use the container to manage the storage of data and fingerprint.

The network communication module provides clients with a highly efficient communication to the nodes in the distributed file system. In detail, the communication exists among clients, the metadata server, and data servers. The main way to conduct those communications is Remote Procedure Call (RPC), by which the metadata and a small amount of control information can be exchanged. Moreover, the transmission of large quantity of data and fingerprints can be done by a stream socket.

3.3 Work Flow

Boafft's work procedure is shown in Fig. 2. Initially, a client partitions the write data stream into multiple chunks, calculates a fingerprint for each chunk, and organizes them into superblocks for data routing. To get the routing address for a superblock, Boafft selects a chunk fingerprint as the feature fingerprint of the superblock, and interacts with the data routing engine of the metadata server. Finally, the client sends the superblock to the corresponding data server for storage and processing.

The metadata server preserves the whole session, and manages the cluster. To assign a routing address for a superblock, it uses a local similarity routing algorithm to determine the best storage node. Meanwhile, we need to take a thorough consideration of storage status of each data server and query results in the process, which require us to select the target node dynamically to balance the storage utilization among data servers, so that the system's storage can be balanced.

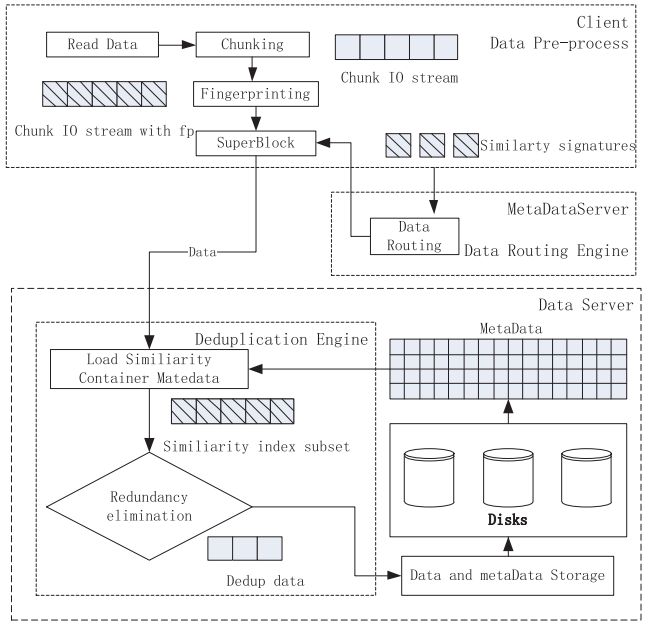


Fig. 2. Work flow of the Boafft system.

Boafft only performs local data deduplication within a single data server. On the basis of data similarity and data locality, Boafft uses self-describing containers to store and manage data and fingerprints. Containers are self-describing in that a metadata section includes the segment descriptors for the stored segments. When a superblock arrives at a data server, the data server loads the fingerprints of the best matched container according to the similarity index, and organizes them into a subset of the similarity index. The Boafft will then compare the superblock with the subset of similarity index for data deduplication, which is able to avoid large mount of I/O operations. Therefore, the overhead of fingerprint query can be decreased greatly. Although this will lower deduplication ratio to some extent, it can get a great promotion in terms of I/O throughput. In addition, to overcome the problem of lower deduplication ratio when only matching similar containers, we design a strategy of container cache and fingerprint index to optimize the index query, and improve the deduplication ratio from the aspect of data stream locality.

4 Data Routing for Distributed Dedup

The Boafft's data routing algorithm is based on the similarity of data, with which a superblock is sent to the corresponding data server according to their content. The goal of the data routing algorithm is to make superblocks with high similarity co-locate in the same data server, where Boafft performs local data deduplication. Boafft selects superblock's feature fingerprint by sampling, and completes the data routing by using stateful routing. In this manner, the improvement of deduplciation ratio can be achieved, while the performance of the storage cluster can be guaranteed.

4.1 Routing Granularity

A granularity of data routing determines a tradeoff between throughput and capacity reduction ratio. Taking small

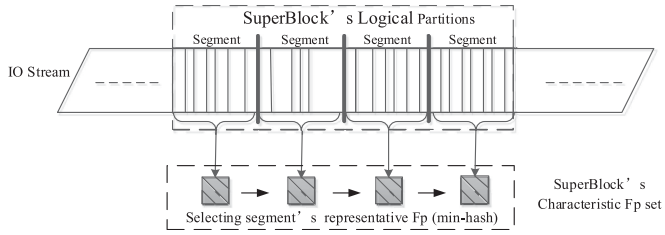


Fig. 3. The procedure of choosing feature fingerprint for a superblock.

chunk as routing granularity can improve the deduplication ratio but will decrease overall system throughput. While taking a file as routing granularity will make a great decrease in deduplication ration, and will cause imbalance of data distribution among data servers. Boafft takes superblock as its basic unit of data routing. When uploading a file, Boafft uses a partitioning algorithm to divide a file into many small chunks. A superblock is made up of a number of continuous chunks. In this way, a file is splitted into some continuous superblocks, and then take superblock as a basic routing unit, which is sent to the selected data server to deduplicate.

The selection of the superblock size is the key factor that affects deduplicaiotn ratio and system performance. Through the experiments on actual datasets, we found that when the superblock size varies between 4 and 16 M, the write efficiency of data stream and the system throughput can be improved, while the deduplication ratio of system can be ensured as well. Refer to the section of experimental evaluation for more details. Moreover, taking a superblock as the granularity of data routing will partition a data stream into some superblocks that will be sent to different data servers, by which we can utilize computing resources of a cluster efficiently and satisfy the applications' demands of parallel processing on big data in the cloud storage system. Besides, it can keep the data locality, improve the performance of read/write, reduce the overhead of network communication, and avoide the problem of data distribution skew when routing a whole file. Besides, we can achieve good deduplication ratio.

4.2 Choosing Feature Fingerprint

In the process of data routing, we assume the incoming data streams have been divided into chunks with a content-based chunking algorithm, and the fingerprint has been computed to identify each chunk uniquely. The main task of data routing is to quickly determine the target data server where an incoming chunk can be deduplicated efficiently. Sending all chunk fingerprints to all data servers for matching will cause huge network overhead. Moreover, it is unrealistic to load all fingerprints into memory in a data server, which will cause a large amount of disk I/Os, and degrading system performance severely. The Boafft takes feature fingerprints for data routing, which was sampled from the data to be routed. Therefore, the Boafft uses similarity matching instead of globally exact matching.

The process of selecting a feature fingerprint for a superblock is shown in Fig. 3. The Boafft divides a superblock into multiple segments based on the principle of fixed size or equal number of chunks. Every segment was composed by some continuous chunks, and the segment is taken as a unit when selecting a representative fingerprint. According to the Broder's theory [31], Boafft selects the Min-Hash as the representative fingerprint of the segment. After the completion of the sampling of all segments in a superblock, the Boafft organizes the selected representative fingerprints of all segments into a characteristic set, and takes the set as the feature fingerprint of the superblock.

The EMC's Cluster Deduplication determines the routing path by adopting simple stateless routing or global search, and system deduplication ratio and throughput will be best when the superblock size is 1MB. However, in cloud storage environments, for these online compression and deduplication systems, forwarding small data blocks consumes a large amount of network resources, and in turn increases systems response time and lowers I/O throughput seriously. Therefore, we redefine the size of a superblock, and sample according to the similarity-based theory to ensure highly efficient deduplication within the data server and satisfy cloud storage's fast response requirement better.

4.3 Stateful Data Routing

Boafft uses the stateful routing algorithm based on data similarity. When storing a data superblock, Boafft sends the feature fingerprint of this superblock to every data server, gets the similarity values of the superblock with the data stored in each data server, and chooses the best matched data server according to similarity values and storage utilization of data servers. Consequently, Boafft can ensure comparatively high deduplication ratio and achieve load balance among data servers.

Each data server maintains a similarity index table, which is used to store the characteristic fingerprints of the superblocks in data containers. When storing a superblock in an open container, the characteristic fingerprint of that superblock will be loaded into similarity index as the container's representative characteristic fingerprint. The similarity index can be loaded into memory. A client sends a superblock's characteristic fingerprint to all data servers. Each data server compares the received fingerprint with the similarity index table in memory to get the hit number of the characteristic fingerprints. According to Border theory, the equal Min-Hash of two sets means that the two sets have high similarity. Therefore, a superblock's hit number in a data server represents that the value of similarity the superblock with the corresponding data server. If the hit number in a data server is large, we believe that the data stored in the node have a high similarity with the superblock.

We determine the final routing address on the basis of a reference value $v_i (v_i = h_i/u_i)$. It is a comprehensive reflection of similar characteristic fingerprint's hit number and the storage utilization of a data server. In this equation, h_i is the hit number of characteristic fingerprints in data server i, and u_i is the percentage of the storage capacity of data server i that is be used. As we can see, when h_i is great, the reference value v_i is great as well, which means selecting such a node i will have a good effect on deduplication ratio. The large value u_i will lead to small reference value v_i, which indicates that the storage utilization of this server has been significantly higher than the average, so we should decrease the probability of selecting node i. Therefore, we can maintain the balance among nodes' storage utilization to some extent in the premise of not decaying the

deduplication ratio. It should be noted that it also works well for a heterogeneous system with data servers having different storage capabilities.

4.4 Description of Data Routing Algorithm

Algorithm 1 shows the pseudo code of local similarity routing algorithm. (1) First, a client partitions the incoming data stream into multiple superblocks, and selects representative fingerprints for it (lines 1-3). (2) The client sends the representative fingerprints of the superblock to all data servers, and collects the numbers of fingerprint hits h from data servers (lines 4-7). (3) The client calculates each data server's reference value v. The data server with the greatest reference value is the ideal one (lines13-18). (4) If reference values v of all data servers are zero, the client selects one randomly, from the ones whose storage utilization is the lowest, as the routing address of the superblock.

Algorithm 1. Local Similarity Routing Algorithm

Input:
s: the superblock to be routed.
Output:
id: the ID of data server selected.
1: split superblock s into k segments;
2: fingerprint $f_i \leftarrow$ min hash of Segment i, $(0 \leq i < k)$;
3: representative fingerprint set $S_f \leftarrow \{f_0, f_1, \ldots, f_{k-1}\}$;
4: **for** each data server D_j **do**
5: connect data server D_j through RPC;
6: $c_j \leftarrow$ hits_from_similarity_index(D_j, S_f);
7: **end for**
8: hit count set $C \leftarrow \{c_0, c_1, \ldots, c_{n-1}\}$;
9: **if** $\forall c_j \in C$, $c_j = 0$ **then**
10: $id \leftarrow$ min_used_node();
11: **return** id
12: **else**
13: **for** each data server D_j **do**
14: $u_j \leftarrow$ storage_usage(D_j);
15: $v_j \leftarrow$ compute_value (D_j, c_j, u_j);
16: **end for**
17: value set $V \leftarrow \{v_0, v_1, \ldots, v_{n-1}\}$;
18: $id \leftarrow m$, **if** v_m is max(V);
19: **return** id;
20: **end if**

Local similarity routing algorithm is based on data locality and data similarity, we take superblock as routing granularity and redefine superblock's organization to meet the demands of the performance of big data's storage in cloud storage. In our implementation, we do logic partition on superblock, select similar characteristic fingerprint locally, and get the data distribution of each node to select the best deduplication node according to stateful data routing algorithm. Meanwhile, to maintain the balance of system storage, we design the routing reference value of each node according to node's current storage status, and the best routing address is determined by the value of this reference value.

5 Local Deduplication within a Data Server

In this section, we describe how a data server works when a data read/write request arrives. Especially, we deliberate on how it performs data deduplication and data regeneration.

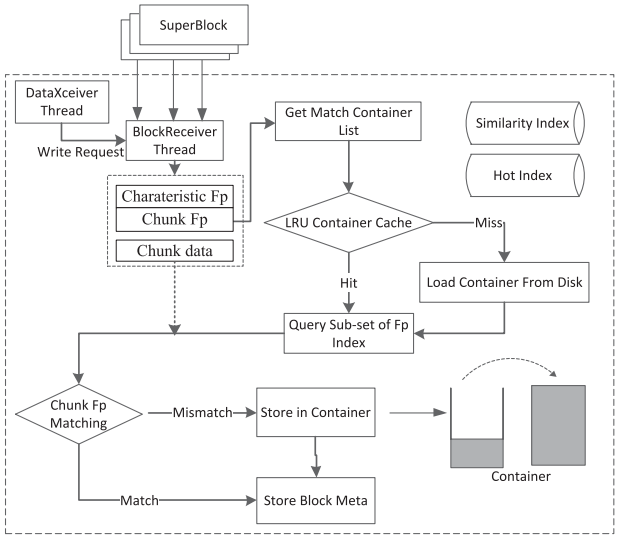


Fig. 4. The procedure of data deduplication in a data server.

5.1 The Procedure of Data Deduplication

A data server deduplicates locally for the incoming data. Fig. 4 shows the process of deduplication in a data server. In a data server, thread DataXceiver is responsible for receiving and processing the read/write requests from client. Once DataXceiver receive the write request, the data server will start thread BlockReceiver to receive and store data. The deduplication engine also works in the BlockReceiver thread.

First of all, a data server receives superblock's characteristic fingerprints and metadata, determines the matched data container by querying Hot Index and similarity index table. A I/O read is required since the container is stored in disk. We use the LRU container cache to take a direct match, which can decrease the number of disk I/O operations to some extent. After organizing the determined container into index subsets, Boafft can search the index for the superblock. If matched, there is no need to store the original data. Otherwise, Boafft selects an open container to store the remaining data. Finally, Boafft writes every chunk's storage information into the disk.

After deduplicating all the chunks in a superblock, Boafft then writes the remained data into the container at once. Compared with the method of writing deduplicated chunk one by one, it can reduce the number of I/O operations and improve the data server throughput when receiving data.

Another important aspect is the maintenance of index in the process of deduplication. Index update can be divided into three parts:

- Update of index in cache, the update of LRU container cache is based on data server's read/write requests and the update of cache is in a way of LRU.
- Update of similarity index table, after storing the container, we need select a characteristic fingerprint from superblocks in the container, and update it to the similarity index table for the later use of querying and matching of similar fingerprint.
- Update of Hot Index table, we update the Hot Index according to the access frequency of chunk's fingerprint, and the chunk stored in a container of the LRU container cache.

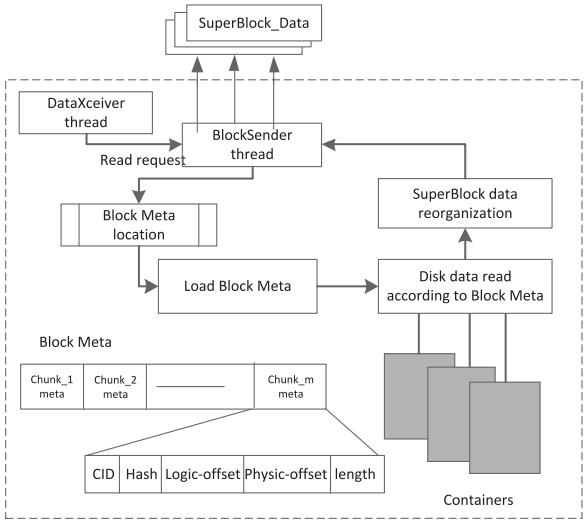


Fig. 5. The procedure of data reorganization in a data server.

5.2 The Procedure of Data Reorganization

When reading a file, a client sends a request to the metadata server for the file metadata. Then, the metadata server returns the data server address of the superblock and the superblock ID to the client. The client establishes a connection with the data server, by which the client reads the corresponding superblock. To handle the read request from a client, a data server reorganizes the data content of the superblock, and returns the whole data to the client.

Fig. 5 shows the process of superblock reorganization in data server. When the DataXceiver thread receives read requests, the system starts the BlockSender thread, which reads and sends the requested superblock. In more detail, BlockSender first positions the physical address of the superblock metadata, and loads the superblock metadata. BlockSender then reorganizes the superblock according to the superblock metadata. Finally, BlockSender sends the reorganized superblock to the client.

When a data server performs local read operations, if the chunk to be read next is not the data but the address of the data, it will cause random disk read. That is, the read of a chunk will across some files. To eliminate the bottleneck of random disk reads, Boafft performs data deduplication only among some similar containers. In this way, there will be not too many I/O operations and will not open many files for data reorganization, which will decrease the possibility of random reads greatly.

5.3 Updating Hot Data Index

On the basis of LRU cache, Boafft uses Hot Index based on the access frequency of fingerprints to improve the deduplication ratio within a single node. With the adaption of Hot Index, when newly coming superblock was not deduplicated in the similar containers, we can turn to the Hot Index for a second deduplication. In this cache, we set the frequency of the fingerprint in the container when match the fingerprint in the cache, update the Hot Index. Algorithm 2 shows the pseudo code of Hot Index's update.

Selecting the match container by similar fingerprint cannot get a good deduplication result, since the similar fingerprint is not a good representation of the container's data characteristics. And the feature of data locality will gather the duplicate data, which promise a good performance for system's deduplication. Consider the situation that some fingerprints of a container being continuously hit, combined with the data locality, there is no doubt that the possibility of the other data in the container that detected as duplication data will increase.

Algorithm 2. Hot Index Update Algorithm

Input:
c: the incoming chunk; l: data container cache list.
Output:
the HotIndex.

```
1: h ← lru_get_cache(l, c);
2: if h = NULL then
3:     delete_fingerprint(HotIndex, c);
4: else
5:     update_ref_count(l, c);
6:     T ← now();
7:   if T - lastupTime(h) > INTERVAL then
8:         lastupTime(h) ← T;
9:         sort(l);
10:        update(HotIndex, l);
11:  end if
12: end if
```

6 Experimental Evaluation

6.1 Implementation and System Settings

By revising the source code of widely-used Hadoop's distributed file system, we implemented the Boafft prototype. Here we conclude the main modifications on HDFS as follows.

- In a client, we revised the original data storage code, and implemented functions of data partition, fingerprint calculation, and data routing in the process of data storage.
- In the metadata server, we retained the metadata and cluster managements, and replaced the network distance perception routing algorithm with our similarity-based data routing algorithm.
- In a data server, we modified the organization of data storage, adding deduplication engine and data reorganization engine, LRU cache, and Hot Index based on data locality and access frequency. Use Berkeley DB for persistent storage, such as a fingerprint similarity index table (Similarity Index) and other metadata.

We used three servers to build up our cluster environment, each server's configurations is shown in Table 1. We test the effect of deduplication in multiple nodes by building virtual machines in physical machines to simulate it. In a simulated environment, we start multiple processes to simulate large scale cluster environment in each machine.

6.2 Evaluation Metrics and Data Sets

We use deduplication ratio, relative deduplication ratio, deduplicaiton performance, deduplicaiotn throughput, and cluster storage balance to analyze and evaluate cluster

TABLE 1
Configuration of the Machines

item	configuration
CPU	Intel Xeon E5-2620@2.00GHz
Memory	32 GB
Disk	1 TB × 2
Network	Intel Gigabit 100 Mbps
Operating system	Ubuntu 12.04 64 bit
Kernel	Linux 3.5.0-23-generic
Operating environment	Jdk1.7

deduplication system. Deduplication ratio is the ratio of logical to physical size of the data. Relative deduplication ratio (RDR) is the ratio of DR obtained by similarity matching within a node to that of DRs get by globally matching. Deduplication throughput (DT) is the ratio of the size of original data with the time for uploading data. Deduplicaition performance (DP) is the size of data that deleted by deduplication per unit time. Cluster storage balance (CSB) is the variance of nodes' storage resource utilization, which is used to test cluster storage balance.

Table 2 shows the datasets used by our experiments. Dataset 1 and Dataset 2 are real datasets, whose sources are the backup data from the web server of the department of computer science of FIU and its mail server [42]. Dataset 3 is the vdi images of five Linux virtual machines in our lab. The deduplication ratio in Table 2 is under the circumstance of 4 KB block size and global search in a single node. Among them, DRs of Dataset 3 are 2.06 and 2.36 by using Fixed-sized partitioning (FSP) [26] and Content-defined chunking (CDC) [15], respectively.

6.3 Micro Benchmarks

6.3.1 *Partitioning of SuperBlock*

In our experiments, we partition the data stream into chunks in the client, which is a widely used granularity today. The chunk size is 4 KB in FSP. While the average size of chunk is 4 KB, and the slide window is 48 bytes in CDC. Besides, we set the maximum size of chunk to 8 KB, the minimum to 2 KB. Our experiments show that different container sizes in data servers have little impact on the deduplication ratio. We set the container size to a fixed value 16 MB based on the three reasons.

- The fixed container size makes allocation and deallocation of containers easier.
- The implementation of Boafft is based on HDFS, which is friendly with large chunks.
- The large granularity of a container write achieves high I/O throughput.

The goal of our first set of Micro benchmarks is to test the size of superblock and the number of superblock's segments

TABLE 2
Description of Data Sets

#	Dataset	Size	DRs
1	Web Dataset	52 GB	2.25
2	Mail Dataset	247 GB	4.91
3	VM Dataset	159 GB	2.06/2.36

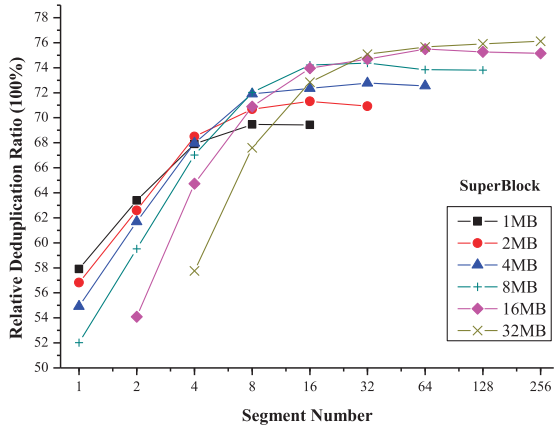


Fig. 6. The impact of superblock size and segment number on dedup ratio.

these two factors' effects on deplication ratio and deduplication throughput. Since there is no data routing overhead in a single node, the number of segment is the main factor that affects deduplication ratio. Henceforth, We test the deduplication ratio by setting different superblock sizes and different segment numbers in each case.

As shown in Fig. 6, for a certain size of superblock, the relative deduplication ratio is improving along with the increase of the number of segments. For different size of superblock, the deduplicaiotn ratio keeps steady when the segment number reaches to a certain value, which we call superblock's best number of segments. This is because the more the number of segments is, the higher the sample frequency of fingerprints is, and the number of containers that need to be compared will increase as well. Moreover, the similarity index of Hash Map will occupy more memory. That is, the increased number of segments do not improve the deduplication ratio but cost more memory. As we can see from the experiments, the best numbers of segments for superblocks with different sizes are—four for 1 MB, eight for 2 and 4 MB, 16 for 8 MB, 32 for 16 MB, and 64 for 32 MB, respectively. In other words, we can divide a superblock into some 512 KB segments on average.

Fig. 7 shows the effects of different superblock sizes on deduplication throughput. We divided a given size of superblock into its best number of segments. As shown in the figure, the deduplication throughput increases gradually along with the increase of the superblock size. The

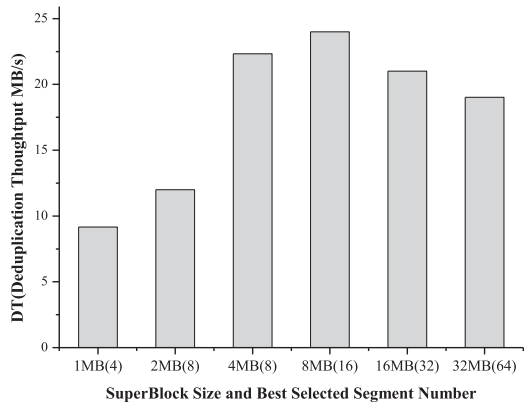


Fig. 7. The impact of superblock size on deduplication throughput.

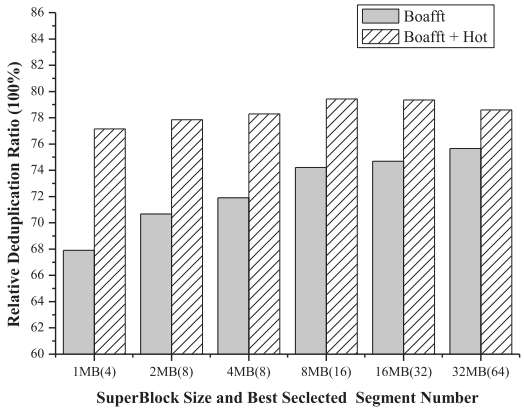


Fig. 8. The impact of hot fingerprint to dedup ratio.

reason for this increase is that the upload of larger superblocks needs to load and match fingerprints of similar containers for fewer times, which results in lower overheads. However, when the superblock size goes beyond a value, the number of characteristic fingerprints that a data server needs to compare is large. The data server receiving a superblock will performs a lot of disk I/Os, and in turn brings a negative impact on system performance. On the other hand, with the increase of the superblock size, although deduplication throughput will increase to some extent, the deduplication ratio in a large scale cluster is low. Therefore, we measure the deduplication performance in the following by choosing the superblock size between 4 and 16 MB.

6.3.2 *Impact of Hot Fingerprint Cache*

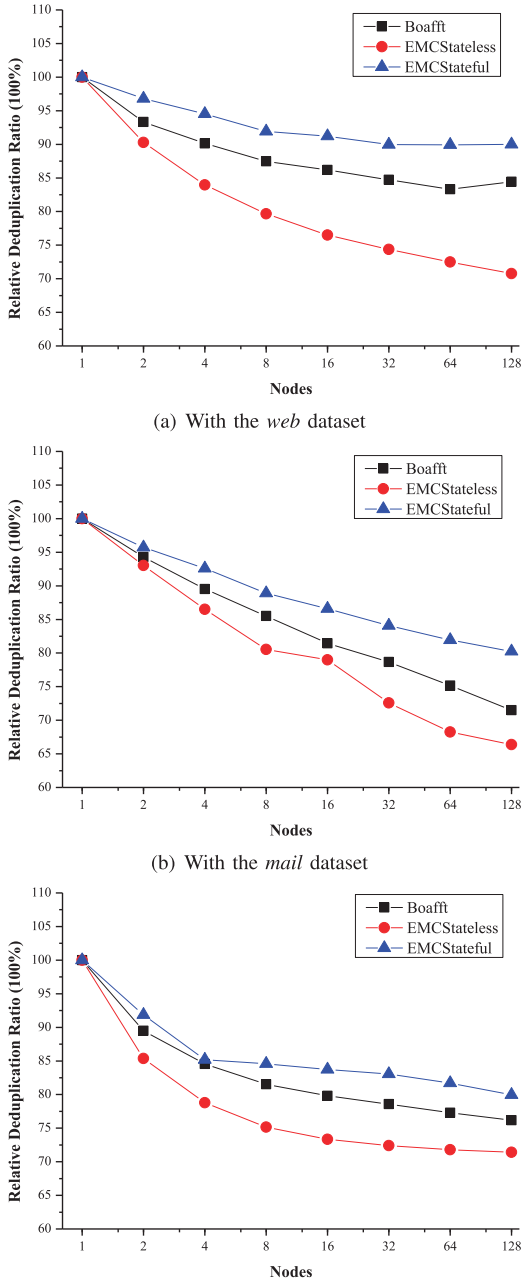
When a data server receives the routing data, it only loads the similar characteristic fingerprint of the container into the Hash to represent the storage information of the container. On the basis of data locality and access frequency, we designed Hot Index to further improve the deduplication ratio within a data server.

Fig. 6 shows the effects of the different size of superblock and the best number of segments on system deduplication ratio. In Fig. 8, we can see a 5-10 percent improvement of RDR after the adaption of Hot Index. The experiment result also show that there exits spatial locality and temporal locality among data.

6.4 Macro Benchmarks

6.4.1 *Deduplication Ratio*

We conducted our experiments in cluster by setting the size of superblock to 8 MB, and constituting the scale of the cluster to 1 128 respect, and compared with these systems based on EMC's Stateful and Stateless routing algorithms. As we can see from Fig. 9, the testing results of the three datasets have the same change rule, and Boafft's deduplication ratio can maintain between EMCStateful and EMCStateless. When the number of node is one, the RDR is always equal to 1. RDR of Boafft, EMCStateless, and EMCStateful is decreasing along with the increase of node. The above result must exits since the increase of available nodes for storage and the difference of data distribution. EMCStateless uses stateless routing algorithm, so its deduplication ratio decreases sharply. When the number of node in the cluster



(a) With the *web* dataset

(b) With the *mail* dataset

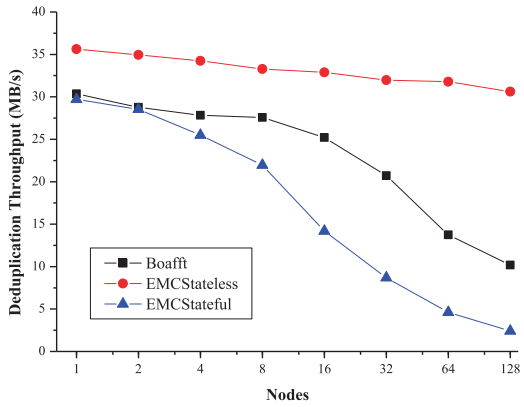
(c) Under the *vm* workload

Fig. 9. Relative deduplication ratio with different system sizes.

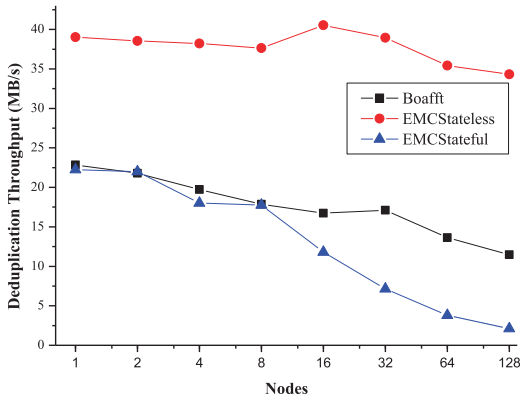
reached to 128, we can only get a about 65 percent RDR of that within a single node. Hence, this stateless algorithm is not feasible in large scale cluster.

For an EMCStateful cluster with 128 nodes, its RDR can reach to 80-90 percent. This way of global stateful data routing can guarantee system a relatively high deduplication ratio. Although Boafft do not have the high deduplication ratio as EMCStateful, its deduplication ratio did not fall fast as EMCStateless. And Boafft's dediplication ratio maintained at around 80 percent, when the cluster has 128 nodes. Above all, in large scale cluster of deduplication system that is based on cloud storage, the local similar routing algorithm applied in Boafft can get a good RDR to some extent.

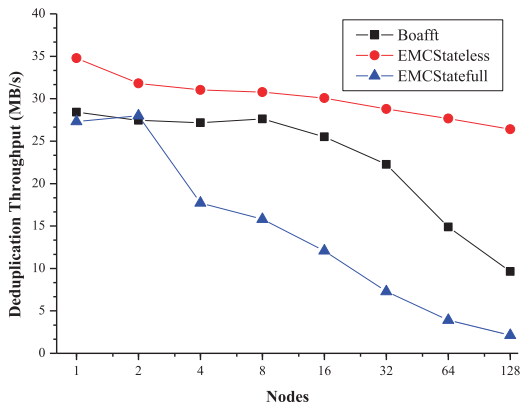
The EMCStateful has a good performance in deduplication ratio, but since the best storage node was decided by



(a) With the *web* dataset

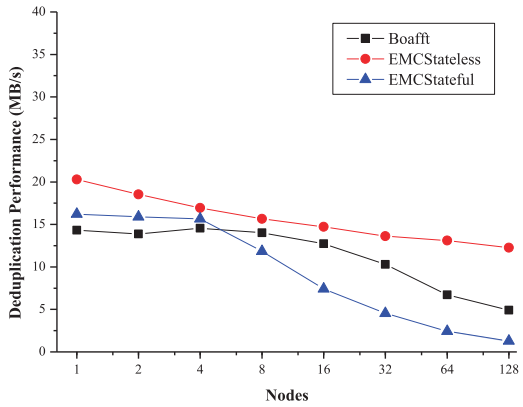


(b) With the *mail* dataset

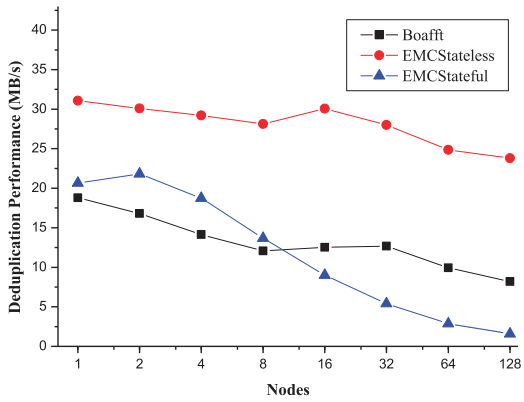


(c) Under the *vm* workload

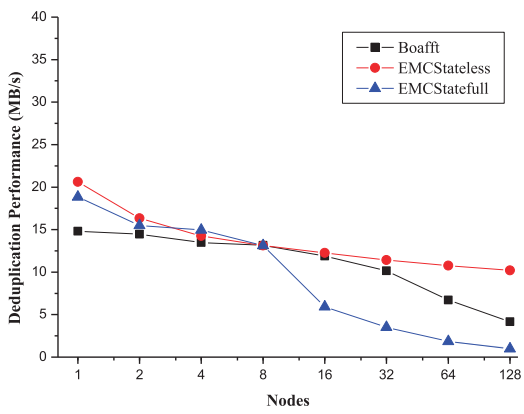
Fig. 10. System throughput with different system sizes.



(a) With the *web* dataset



(b) With the *mail* dataset



(c) Under the *vm* workload

Fig. 11. System performance with different system sizes.

matching all indexes of the node, the overhead of transmitting fingerprint is too large. In addition, the overhead of global match within a node is its main bottleneck.

6.4.2 System Overhead

Figs. 10 and 11 shows the experiments of the deduplication throughput and system performance of Boafft, EMCStateful and EMCStateless under different scale of cluster. EMCStateless has high deduplication throughput and system performance. The main reason lies in that it does not consume any network transmission overheads in the process of data routing, even if the scale of the cluster is large.

Boafft and EMCStateful implemented stateful data routing, so their storage performance decreases along with the increase of the number of data nodes. However, Boafft's storage performance is better than EMCStateful with the increase of nodes. This is because the amount of matched information transmitted by Boafft's similar match is 1/128 of that by EMCStateful, and the memory usage of fingerprint matching in data server is 1/128 of that in EMCStateful. Therefore, although Boafft does not achieve high deduplication ratio as EMCStateful, its storage performance is better and the consumed memory is less.

6.4.3 Load Balance among Data Servers

The balance of storage utilization is a main aspect of evaluating a distributed system. In different scale of cluster, we can get each node's disk usage after the upload of datasets and get the cluster's status of storage balance by the calculation of CSB.

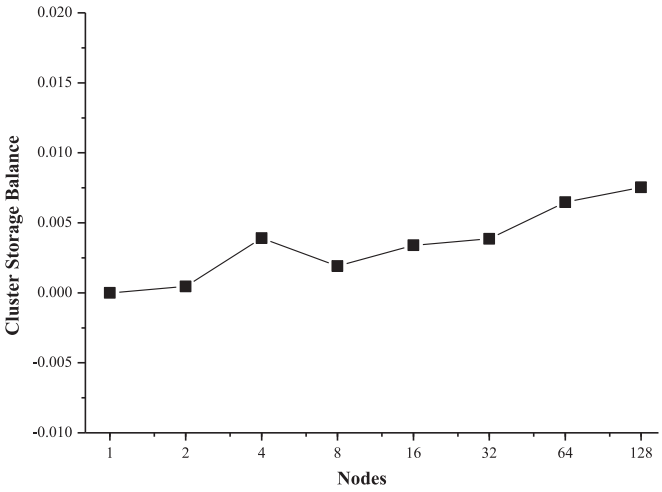


Fig. 12. The deviation of storage utilization with the increase of cluster nodes.

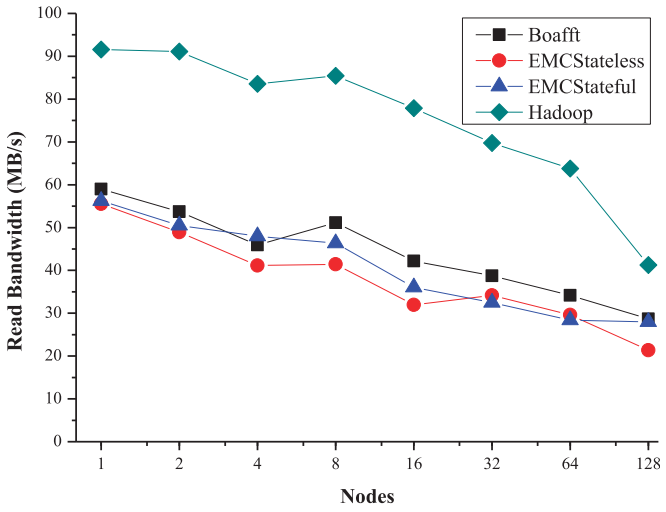


Fig. 14. The read bandwidth of cluster storage system.

Fig. 12 shows the CSB results in our experiments. With the increase of cluster's node, the deviation of storage increases gradually. From the above observation, we can see that the storage deviation of each node in Boafft is increased under the condition of the cluster's scale is large. However, CSB tend to increase slowly from the trend, which means the routing algorithm of Boafft did not have very negative impact on system storage balance. This is because our routing algorithm takes the status of storage into consideration and decides the data routing address dynamically online.

6.4.4 I/O Bandwith

Finally, we test the read/write bandwidth of Boafft, EMCStateless and EMCStateful to analyze their I/O Bandwiths in different scale of cluster. Fig. 13 shows the write bandwidth. The write performance of the system with deduplication was significantly lower than the original Hadoop system.

With the increase of the scale of cluster, system's write bandwidth decreases. The above phenomenon results from the errors caused by the simulation of large-scale clusters on a single physical machine. However, as we can see from the experimental results, it is obvious that the write bandwidth of Boafft is larger than EMCStateful. One of the reasons is that Boafft's memory cost is only 1/128 of that in EMCStateful.

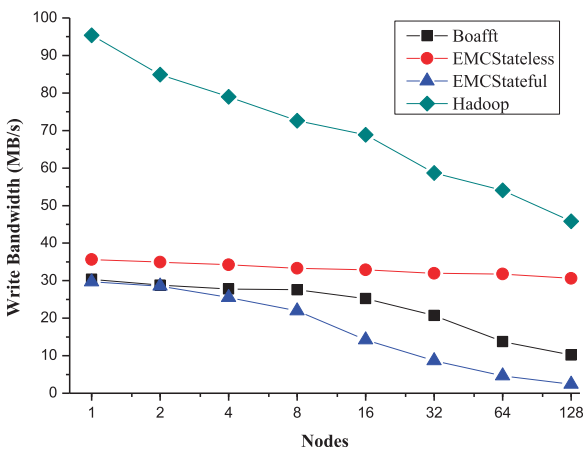


Fig. 13. The write bandwidth of cluster storage system.

Fig. 14 shows the read bandwidth of four storage cluster systems. First, Hodoop's read bandwidth is greater than the others with cluster deduplication. This is because Hadoop read file sequentially, while the addition of deduplication changes sequential storage of data on disk into random storage, there are random reads in data reads. Second, Boafft's read bandwidth is larger than EMCStateful and EMCstateless in different scale of cluster. Boafft's deduplication is based on a small amount of similar containers, which saves the overheads of comparing too much containers' fingerprints.

7 Conclusions

In this paper, we present Boafft, a cloud storage system with distributed deduplication. It achieves scalable throughput and capacity using multiple storage nodes to deduplicate in parallel, with a minimal loss of deduplication ratio. First, Boafft adopts efficient data routing algorithm based on data similarity, which not only reduce the overhead of network bandwidth, but can also calculate the storage location of data fast. Second, each data server maintains a similarity index table in memory, which can be used to deduplicate data partially, and a large number of disk random reads/writes can be avoided. Third, we improve the data deduplication ratio in single node with the help of cache container of hot fingerprint based on access frequency.

We implemented the prototype system of Boafft by revising the source code of widely-used Hadoop distributed file system. Experiment results show that Boafft can provide a relatively high deduplication ratio, and compared with EMCStateful, its network overheads is lower, memory usage can reduced to 1/128 of the EMCStateful, storage utilization and read/write bandwidth is higher, and load balance is also good.

Acknowledgments

We are grateful to Zhiran Li for providing helpful comments and assistance with our experimentation. This work was supported by the National Natural Science Foundation

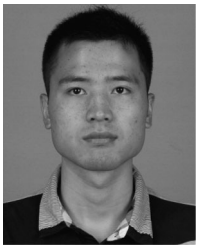
of China under Grants 61170008, and 61272055, the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402, and the National High Technology Research and Development Program of China under Grant 2013AA01A210. The work was also support in part by the National Science Foundation grant CNS 1229316. G. Zhang is the corresponding author of this paper.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
[2] K. Shvachko, H. Kuang, S. Radia, et al., "The Hadoop distributed file system ," in *Proc. IEEE 26th Symp.Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[3] S. A. Weil, S. A. Brandt, E. L. Miller, et al., "Ceph: A Scalable, High-Performance Distributed File System", in *Proc. 7th USENIX Symp. Operating Syst. Design Implementation*, 2006, pp. 307–320.
[4] D. Nurmi, R. Wolski, C. Grzegorczyk, et al., "The eucalyptus open-source cloud-computing system," in *Proc. 9th IEEE/ACM Int. Symp. Cluster Comput. Grid*, 2009, pp. 124–131.
[5] Gluster File System [Online]. Available: http://www.gluster.org/community/documentation/index.php, visited in 2015.
[6] J. Gantz, and D. Reinsel, "The digital universe decadełare you ready?" *IDC White Paper*, May 2010[J]. 2011.
[7] H. Biggar, "Experiencing data de-duplication: Improving efficiency and reducing capacity requirements," *White Paper*, the Enterprise Strategy Group, Feb. 2007[J]. 2012.
[8] A. Jas, J. Ghosh-Dastidar, M. E. Ng, et al., "An efficient test vector compression scheme using selective Huffman coding[J]," *IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst.*, vol. 22, no. 6, pp. 797–806, Jun. 2003.
[9] J. H. End III, "Hardware-based," LZW data compression Co-processor: U.S. Patent 6624762[P]. Sep. 9, 2003.
[10] H. Che, Z. Wang, K. Zheng, et al., "DRES: Dynamic range encoding scheme for tcam coprocessors," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 902–915, Jul. 2008.
[11] L. P. Deutsch, "DEFLATE compressed data format specification version 1.3," RFC Editor, 1996.
[12] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proc. SYSTOR: Israeli Experimental Syst. Conf.*, 2009, p. 7.
[13] A. Adya, W. J. Bolosky, M. Castro, et al., "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment[J]," in *Proc. 5th Symp. Operating Syst. Des. Implementation*, 2002, pp. 1–14.
[14] (2002). EMC Centera, content addressed storage, product description [Online]. Available: http://www.emc.com/pdf/products/centera/centera_guide.pdf
[15] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario[C]," in *Proc. Israeli Exp. Syst. Conf.*, 2009, p. 8.
[16] J. Kubiatowicz, D. Bindel, Y. Chen, et al., "Oceanstore: An architecture for global-scale persistent storage[J]," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.
[17] S. Quinlan and S. Dorward, "Venti: A new approach to archival Storage,"in *Proc. Conf. File Storage Technol.*, 2002, vol. 2, pp. 89–101.
[18] M. O. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technol., Harvard Univ., Cambridge, MA, USA, Tech. Rep. TR-15-81, 1981.
[19] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.
[20] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: Making backup cheap and easy," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, no. SI, pp. 285–298, 2002.
[21] D. Bhagwat, K. Eshghi, D. D. E. Long, et al., "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proc. IEEE Int. Symp. Modell. Anal. Simulation Comput. Telecommun. Syst.*, 2009, pp. 1–9.
[22] W. Dong, F. Douglis, K. Li, et al., "Tradeoffs in scalable data routing for deduplication clusters," *Proc. Conf. File Storage Technol.*, 2011, pp. 15–29.
[23] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep Store: An archival storage system architecture," in *Proc. 21st Int. Conf. Data Eng.*, 2005, pp. 804–815.
[24] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms[J]," Hewlett-Packard Labs, Palo Alto, CA, USA, Tech. Rep. HPL-2005-30R1, 2005.
[25] C. Liu, Y. Lu, C. Shi, et al., "ADMAD: Application-driven metadata aware de-duplication archival storage System," in *Proc. 5th IEEE Int. Workshop Storage Netw. Archit. Parallel I/Os*, 2008, pp. 29–35.
[26] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Trans. Storage*, vol. 2, no. 4, pp. 424–448, 2006.
[27] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *Proc. USENIX Conf. File Storage Technol.*, 2010, pp. 239–252
[28] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. Conf. File Storage Technol.*, 2008, vol. 8, pp. 1–14.
[29] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[30] M. Lillibridge, K. Eshghi, D. Bhagwat, et al., "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. Conf. File Storage Technol.*, 2009, vol. 9, pp. 111–123.
[31] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. Compression Complexity Sequences*, 1997, pp. 21–29.
[32] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proc. Conf. USENIX Annu. Techn. Conf.*, 2010, pp. 16–16.
[33] EMC Data Domain Global Deduplication Array [Online]. Available: http://www.datadomain.com/products/global-deduplication-array.html, visited in 2015.
[34] C. Dubnicki, L. Gryz, L. Heldt, et al., "HYDRAstor: A Scalable Secondary Storage[C]," *FAST*, 2009, vol. 9, pp. 197–210.
[35] D. Frey, A.-M. Kermarrec, and K. Kloudas, "Probabilistic Deduplication for Cluster-based Storage Systems[C]," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012, pp. 17:1–17:14.
[36] 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[C]," *ATC*, 2011.
[37] Y. Fu, H. Jiang, and N. Xiao, "A Scalable Inline Cluster Deduplication Framework for Big Data Protection[C]," *Proceedings of the 13th International Middleware Conference*, 2012, pp. 354–373.
[38] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan, "DEBAR: A scalable high-performance de-duplication storage system for backup and archiving[C]," in *Proc. IEEE Symp. Parallel Distrib. Process*, 2010, pp. 1–12.
[39] T. Yang, D. Feng, Z. Niu, and Y. Wan, "Scalable high performance deduplication backup via hash join," *J. Zhejiang Univ.–Sci.*, 2010, vol. 11, pp. 1–13.
[40] R. Real and J. M. Vargas, "The probabilistic basis of Jaccard's index of similarity," *Syst. Biol.*, 1996, pp. 380–385.
[41] A. Z. Broder, M. Charikar, A. M. Frieze, et al., "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, no. 3, pp. 630–659, 2000.
[42] FIU IODedup.Traces [Online]. Available: http://iotta.snia.org/traces/391, visited in 2015.

Shengmei Luo received the Bachelor's and Master's degrees in communication and electronic from the Harbin Institute of Technology, in 1994 and 1996, respectively. He is currently working toward the PhD degree at the Tsinghua University. He joined the ZTE company in 1996, and his current research interests include big data, cloud computing, and network storage.

Guangyan Zhang received the Bachelor's and Master's degrees in computer science from the Jilin University, in 2000 and 2003; the doctor's degree in computer science and technology from the Tsinghua University, in 2008. He is currently an associate professor in the Department of Computer Science and Technology Tsinghua University. His current research interests include big data computing, network storage, and distributed systems. He is a professional member of the ACM.



Chengwen Wu received the Bachelor's degree in computer science from the Beijing University of Posts and Telecommunications, in 2014. He is currently working toward the Master's degree at the Department of Computer Science and Technology, Tsinghua University. His current research interest is in big data processing and network storage.

Samee U. Khan received the PhD degree from the University of Texas, in 2007. Currently, he is the Department head and the James W. Bagley chair of Electrical & Computer Engineering at the Mississippi State University (MSU). Before arriving at MSU, he was Cluster lead (2016-2020) for Computer Systems Research at National Science Foundation and the Walter B. Booth professor at North Dakota State University. His research interests include optimization, robustness, and security of computer systems. His work has appeared in more than 400 publications. He is associate editor-in-chief of the IEEE IT Pro, and an associate editor of *Journal of Parallel and Distributed Computing* and *ACM Computing Surveys*.

Keqin Li is a SUNY Distinguished Professor of computer science. His current research interests include parallel computing and highperformance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published over 380 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *Journal of Parallel and Distributed Computing*. He is a fellow of the IEEE.