LOFS: A Lightweight Online File Storage Strategy for Effective Data Deduplication at Network Edge

Geyao Cheng, Deke Guo, Senior Member, IEEE, Lailong Luo, Junxu Xia, and Siyuan Gu

Abstract—Edge computing responds to users’ requests with low latency by storing the relevant files at the network edge. Various data deduplication technologies are currently employed at edge to eliminate redundant data chunks for space saving. However, the lookup for the global huge-volume fingerprint indexes imposed by detecting redundancies can significantly degrade the data processing performance. Besides, we envision a novel file storage strategy that realizes the following rationales simultaneously: 1) space efficiency, 2) access efficiency, and 3) load balance, while the existing methods fail to achieve them at one shot. To this end, we report LOFS, a Lightweight Online File Storage strategy, which aims at eliminating redundancies through maximizing the probability of successful data deduplication, while realizing the three design rationales simultaneously. LOFS leverages a lightweight three-layer hash mapping scheme to solve this problem with constant-time complexity. To be specific, LOFS employs the Bloom filter to generate a sketch for each file, and thereafter feeds the sketches to the Locality Sensitivity hash (LSH) such that similar files are likely to be projected nearby in LSH tablespace. At last, LOFS assigns the files to real-world edge servers with the joint consideration of the LSH load distribution and the edge server capacity. Trace-driven experiments show that LOFS closely tracks the global deduplication ratio and generates a relatively low load std compared with the comparison methods.

Index Terms—Data deduplication, locality sensitivity hash, edge computing, storage systems

1 INTRODUCTION

With the threat of big data, storing data at network edge can save precious bandwidth and respond to users with relatively low latency [1], [2]. However, most end devices, such as sensors and surveillance cameras, often collect massive duplicated data due to their similar application scenarios or requirements for file backup. This leads to unnecessary storage space overhead and network transmission [3], which burdens the edge storage systems significantly. To decrease the redundancy, data deduplication is a feasible solution for space saving at the network edge. The deduplication ratio (i.e., the ratio of saved space after deduplication) can reach 75% for the accelerometer data [4], while data redundancy can be reduced up to 70% in multimedia and traffic IoT data [5], [6].

A common practice for data deduplication is to split files into multiple chunks with fixed size [4], [7], [8] or variable size [9], [10], [11], [12]. The variable-sized chunking has been demonstrated to be more effective for similarity detection, because it declares chunk boundaries based on the byte contents of the data stream instead of on the byte offset. Only one copy of each chunk is maintained in the storage system by comparing their fingerprints. Inline deduplication [3], [13], [14], [15], [16] maintains a global fingerprint index and realizes data deduction through index lookup before allocating data into a specific server. The redundancies would be discarded directly in the write path when their fingerprints are detected. However, the big-volume fingerprint index may retard the writing throughput, because the lookup time inflates linearly with the number of stored fingerprints [17]. This situation is even worse at edge, where the fingerprint indexes can be scattered across edge servers. The reason is that frequent access to the scattered indexes requires high network-protocol costs between the clients and remote edge servers.

The post-process deduplication [4], [8], [18], [19], by contrast, writes the raw data to the disk directly and then deduplicates them offline. Therefore, no operations are introduced within the critical write path, which enlarges the writing throughput. This feature is especially important at edge, where the terminals may generate data massively and quickly. The disadvantage is that more space resources would be occupied to store the raw data temporarily. A feasible scheme for this intractable disadvantage is allocating similar

---

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
data to the same edge server transcendentally. This might avoid the subsequent data redirection and become a prerequisite for the efficient data deduplication at the server level.

Therefore, in this paper, we focus on leveraging the post-process deduplication to provide a constant-time online file allocation strategy with the transcendent detection of file similarity. Hence, in the situations that files arrive in rapid succession, the undelayed and similarity-aware file allocation would not bring extra space overhead and data migration to store and reallocate such files. We also envision the following three design rationales for the file storage strategy with data deduplication at the network edge. It is path-breaking to achieve these rationales simultaneously.

The first rationale is space efficiency, which is frequently emphasized to reduce space overhead. To achieve this, the file storage strategy should be able to capture the similarities between files and eliminate their duplicates. The second rationale, i.e., access efficiency [8], [20], is appreciated in recent years. It pursues to store all partitioned chunks of one file in whole with a sole server, such that accessing a file will not require excessive rounds of communications to retrieve all involved chunks from multiple edge servers. This is crucial in the edge environment with uncontrollable transmission delay, because the inefficient file access will further retard the reading throughput and impact users’ QoS. Third, file storage must consider load balance to reduce potential space overflows. To this end, proper file storage should match the server capacities, whether uniform or heterogeneous, to avoid hotspots.

Some file deduplication strategies indicate the file similarities relying on the construction of $s$-similarity graph [4], [8], where the load balance is not discussed in detail. Such an offline strategy is also not desirable in real practices, which inevitably leads to a high computation overhead and long allocation latency. PDCSS [9] allocates data to a suitable storage server based on similarity detection according to the cardinality estimation. MinHash [21] and its related works [22], [23] utilize the representative fingerprint to define the data’s signature. The drawback of these two is that the access efficiency is not satisfied, and the load balance can only be achieved for cluster storage systems with uniform server capacities. Furthermore, excessive rounds of communications are required to exchange storage messages or retrieve the fingerprint indexes. The information interaction is time-consuming, especially at edge, which impacts both the reading and writing throughput of the storage systems.

To this end, we propose a Lightweight Online File Storage strategy, LOFS, which aims at allocating files online to the proper edge servers. The LOFS can detect file similarity with constant time complexity and low generation latency. This similarity-aware storage strategy would facilitate the subsequent post-process data deduplication locally. Furthermore, the LOFS scheme would satisfy the aforementioned design rationales simultaneously. A demonstration of LOFS, as well as two state-of-the-art strategies, is shown in Fig. 1. Specifically, Scheme 1 is the common storage method for deduplication systems. It promises space efficiency and load balance by storing data chunks evenly without redundancies. However, due to the lack of data integrity, scheme 1 has to direct a file request to multiple servers to fetch the dispersed data chunks across the network. The SAP strategy [8] (i.e., scheme 2) ensures both access efficiency and space efficiency, but remains inapplicable since load balance cannot be strictly guaranteed, where the server capacities are heterogeneous, especially at the network edge or P2P networks [24]. By contrast, LOFS (i.e., scheme 3) is a relatively ideal solution, which simultaneously considers content similarity, file integrity, and space heterogeneities.

Specifically, LOFS adopts an elegant three-layer hash mapping scheme to realize the Lightweight Online File Storage strategy with satisfying the three design rationales satisfactorily. In the first layer, an online-arriving file is partitioned into several variable-sized chunks ($B_1 \sim B_2$). Two servers are considered, whose storage capacities are $S_1 = 5$ and $S_2 = 4$.

![Fig. 1. An illustrative example of the three criteria verified for the LOFS strategy. Five files ($A_1 \sim A_5$) are partitioned into variable-sized chunks ($B_1 \sim B_6$). Two servers are considered, whose storage capacities are $S_1 = 5$ and $S_2 = 4$.](image-url)

Files:

| Server 1: $S_1 = 5$ | Server 2: $S_2 = 4$ |
|---------------------|---------------------|
| $A_1 - B_1, B_4$   | $B_2, B_3, B_5$    |
| $A_2 - B_1, B_2$   | $B_3, B_4, B_5$    |
| $A_3 - B_1, B_2, B_3$ | $B_4, B_5$       |
| $A_4 - B_2, B_3, B_4$ | $B_1, B_5$       |
| $A_5 - B_3, B_4, B_5$ | $B_1, B_2$       |

1) Space efficiency $\checkmark$ 2) Space efficiency $\checkmark$ 3) Space efficiency $\checkmark$

Access efficiency $\times$ 4) Access efficiency $\checkmark$ 5) Access efficiency $\checkmark$

Load balance $\checkmark$ 6) Load balance $\checkmark$ 7) Load balance $\checkmark$

- We report LOFS, a lightweight online file storage strategy for data storage at the network edge, which simultaneously achieves the three design rationales. It can be solved by the three-layer hash mapping scheme elegantly with constant-time complexity and low generation latency.
We attach the mathematical analyses to the three-layer hash mapping scheme, which theoretically highlight the effectiveness of LOFS.

We validate LOFS using two real-world datasets under realistic scenarios. The results show that LOFS realizes and outperforms its same-kinds in terms of the three design rationales.

The rest of this paper is organized as follows. Section 2 presents the problem statement. In Section 3, we discuss the challenges and solutions in each layer of hash mappings. Section 4 demonstrates performance analyses. Section 5 reports our experimental results. Section 6 introduces related work. Finally, Section 7 provides some discussions and Section 8 concludes this paper.

2 Problem Statement

We consider an online distributed storage system that consists of heterogeneous servers at the network edge. The ambition is to provide a lightweight online file storage strategy to distribute the incoming files into the storage system with satisfying the three design rationales simultaneously. We describe a scenario as follows. The end devices, such as surveillance cameras, collect massive data rapidly and constantly. To save precious bandwidth resources and respond to users with low latency, we prefer to store data at the network edge. The data (such as surveillance videos) may share some common content inevitably due to the similar frames at some specific monitoring point of view. Storing these common contents multiple times is obviously uneconomic. Thus, the utilization of data deduplication techniques is indispensable.

To avoid the linear time-complexity of index lookups and enlarge the deduplication throughput, we prefer to utilize the post-process deduplication. This deduplication mode conducts data reduction when the data has been allocated to a specified edge server. Then, the redundancy can be eliminated locally with relatively sufficient computing and storage resources, rather than in the critical write path. The crucial technique to realize the efficient data deduplication is determining the optimal edge server to allocate the online arriving data.

To state the problem clearly, we assume a scenario as follows. A set of $n$ files $A = \{A_1, A_2, \ldots, A_n\}$ have been stored in the system with $m$ edge servers $S = \{S_1, S_2, \ldots, S_m\}$. The storage capacities of these servers are $C_1, C_2, \ldots, C_m$, respectively. To facilitate data deduplication, files are split into multiple variable-sized chunks, and each chunk is marked by a calculated fingerprint. Storage server $S_j$ caches an index $I_j$ to record the chunk fingerprints that it stores. Only one copy of each chunk is maintained at each server by looking up the local index and comparing the fingerprints.

When an online file $A_{new}$ is generated, it would be first chunked at the edge gateway, and then the allocation scheme is generated there. The target is to find a lightweight and online file allocation for $A_{new}$ with low time complexity. Besides, the strategy should realize the three design rationales simultaneously. Note that, the access efficiency can be easily realized when we assign all chunks of file $A_{new}$ into exactly one edge server. A direct and naive approach is HashID, where files are indexed by their IDs provided by the underlying storage systems (performance shown in Section 5).

A tradeoff exists between the access efficiency and the space efficiency. The access efficiency realizes the convenient file access, such that a file can be fetched with one single round of communication. However, it affects the space efficiency to some extent. The reason is that some chunks may exist at multiple servers simultaneously to maintain the file integrity with all involved data chunks. We regard the access efficiency as a prerequisite of others, and think it is especially important for the edge environment. The reason is that the inefficient file access will greatly impact the reading throughput of the storage system, which is not conducive to the large-scale information interaction at edge.

A tradeoff is also maintained between the load balance and the space efficiency. To minimize data duplicates, files with the most same chunks should be allocated to the same edge server. However, the heterogeneous similarity between files can easily lead to the storage burden for some specific servers, because some files may be accumulated to one server due to the high similarity across the files. For the load balance purpose, the file should be allocated to the most underloaded server that owns the least unique data chunks, i.e., capacity-aware storage strategy (CASS, performance shown in Section 5). These two rationales may contradict each other. The reason is that the most underloaded server stores the least unique chunks, thus may have a low probability to contain numerous common chunks of the new-arrived files.

To provide the elegant trade-offs between space efficiency, load balance, and access efficiency in file storage, we present the LOFS strategy in this paper. LOFS implements a three-layer hash mapping scheme for file storage at the network edge. This scheme allocates files in the absence of fingerprint lookups or excessive rounds of communications. Only lightweight hash calculations are utilized to determine the proper allocation server with the constant time complexity and low generation latency. This promotes online file storage and enlarges the deduplication throughput. In addition, the lightweight hash mappings satisfactorily sketch the files and preserve the similarity between them through aggregating similar sketches together in the hash table space. The table space partition further supports the load balance rationale by catering the mapped data volume in the buckets to the server capacities.

3 Design of LOFS

In this section, we first propose the system architecture of our LOFS strategy in Section 3.1. Then, we mainly focus on introducing the three-layer hash mappings. After providing the framework in Section 3.2, we discuss the encountered problems and put forward feasible solutions with time-complexity analyses in each layer of hash mappings. We also provide possible strategies to reduce overhead through granularity coarsening in Section 3.6.

3.1 System Architecture of LOFS

The overall system architecture of our LOFS strategy is shown in Fig. 2, which includes three key stages: i) Chunking & fingerprinting, ii) Three-layer hash mappings, and iii) Offline deduplication. The dataflow generated from the end devices would be split into chunks, and then, each chunk would be attached with a fingerprint (i.e., SHA1 digest) in

- Provide the elegant trade-offs between space efficiency, load balance, and access efficiency in file storage, we present the LOFS strategy in this paper. LOFS implements a three-layer hash mapping scheme for file storage at the network edge. This scheme allocates files in the absence of fingerprint lookups or excessive rounds of communications. Only lightweight hash calculations are utilized to determine the proper allocation server with the constant time complexity and low generation latency. This promotes online file storage and enlarges the deduplication throughput. In addition, the lightweight hash mappings satisfactorily sketch the files and preserve the similarity between them through aggregating similar sketches together in the hash table space. The table space partition further supports the load balance rationale by catering the mapped data volume in the buckets to the server capacities.

3.1 System Architecture of LOFS

The overall system architecture of our LOFS strategy is shown in Fig. 2, which includes three key stages: i) Chunking & fingerprinting, ii) Three-layer hash mappings, and iii) Offline deduplication. The dataflow generated from the end devices would be split into chunks, and then, each chunk would be attached with a fingerprint (i.e., SHA1 digest) in

- Provide the elegant trade-offs between space efficiency, load balance, and access efficiency in file storage, we present the LOFS strategy in this paper. LOFS implements a three-layer hash mapping scheme for file storage at the network edge. This scheme allocates files in the absence of fingerprint lookups or excessive rounds of communications. Only lightweight hash calculations are utilized to determine the proper allocation server with the constant time complexity and low generation latency. This promotes online file storage and enlarges the deduplication throughput. In addition, the lightweight hash mappings satisfactorily sketch the files and preserve the similarity between them through aggregating similar sketches together in the hash table space. The table space partition further supports the load balance rationale by catering the mapped data volume in the buckets to the server capacities.

3.1 System Architecture of LOFS

The overall system architecture of our LOFS strategy is shown in Fig. 2, which includes three key stages: i) Chunking & fingerprinting, ii) Three-layer hash mappings, and iii) Offline deduplication. The dataflow generated from the end devices would be split into chunks, and then, each chunk would be attached with a fingerprint (i.e., SHA1 digest) in
the Chunking & fingerprinting stage. Thereafter, the gateway conducts the three-layer hash mappings to quantify file similarities and decide the allocated edge server for each online-arriving file. Our LOFS strategy mainly focuses on the first two stages, i.e., how to allocate those online-arriving files to edge servers. We assume that the gateway serves a large number of terminals in a region and is composed of processors and peripheral functional modules. This permits the edge gateway to act as the data access point and perform chunking and hashing calculations.

Based on the local fingerprint index, the data deduplication can be realized offline at each edge server. The duplicates would be discarded when their fingerprints are detected from the index, while the unique chunks are stored with the index updated. A master node is also needed for our LOFS strategy to collect the server load information from the edge servers. This assists the adjustment on the bucket interval partitions in the third layer hash mappings for the load balance purpose. Note that, this interval update can be realized periodically, but not necessary in the strategy generation process in real time. Therefore, there is no impact on the write throughput.

### 3.2 Framework of the Three-Layer Hash Mapping

The existing deduplicated storage systems are often based on cluster storage with centralized designs. They generally require frequent information interactions between the involved storage nodes. The premise of such designs is that the delay between any nodes is usually low and stable. However, in edge scenarios, communications between geo-distributed edge servers may bring significant delays, leading to the impact on the writing throughput and users’ QoS. To this end, one insight of LOFS is to allocate files through hash mappings in a manner similar to that of hash functions in the common distributed storage systems, such as Ceph [27]. That is, files with similar contents can be “hashed” directly to the same server. It seems unrealistic in the existing designs of deduplicated storage systems, because information queries and fingerprint comparisons are required among storage nodes to capture the similarity of files. However, if it is realized, it has the advantage of providing high writing throughput without fingerprint indexing and excessive rounds of communication between edge servers.

To this end, LOFS adopts a novel three-layer hash mapping scheme, which is inspired by the inherent properties of Bloom filter (BF) and Locality Sensitivity Hash (LSH). The hash mapping method for our LOFS strategy is composed of three layers, as shown in Fig. 3. In the first layer, the online-arriving file $A_i$ is partitioned into several variable-sized chunks. The chunks are hashed into a BF through hash functions in $\xi_1$, where each hit position in the BF is set as “1”. The derived sketch captures the characteristics of the arriving file and then acts as the input in the second layer. In this layer, LSH is employed to hash similar file sketches (with short Hamming distance) to nearby positions in the LSH table using hash functions in $\xi_2$. In the third-layer hash mappings with $\xi_3$, we partition the LSH tablespace into several unique buckets, where the space of each bucket caters to the storage capacity of its hashed edge server heterogeneously. Thereafter, all files whose hashed values are fallen in the bucket region would be allocated to the corresponding edge server. The three layers together promote the probability that similar files are allocated to the same server. Therefore, data deduplication is accomplished at the server level. Through deploying the three-layer hash mapping, files stored in the same server have a more significant similarity and repetition rate, which plays a vital role in improving the deduplication effect afterwards.

From a high level of view, the access efficiency is originally guaranteed when the storage policy assigns each file as a whole to exactly one edge server. The similarity characteristics of the online-arriving files are quickly captured through the first-layer hash mappings. Then the similar files would be allocated into the same edge server through the collective efforts of the lightweight hash functions in $\xi_1$, $\xi_2$, and $\xi_3$. This would significantly save the storage overhead and promote space efficiency because more redundancies can be detected and then eliminated when similar files are located on the same server. Furthermore, the file storage scheme caters to the heterogeneous storage capacity of each participated edge server, thus achieving load balance. In brief, the three-layer hash mapping method can achieve the three design rationales simultaneously.

#### 3.3 First Layer: Bloom Filter-Based File Sketch

The main idea of the first layer is to hash files into sketches, where the elements in each sketch can indicate whether the file contains the corresponding chunk or not. The sketch can decrease the computing overhead and enlarge the processing throughput of the afterward calculations. The hash mappings can be denoted as $\xi_1: A \rightarrow C^d$, where $A$ donates the files, and $C^d$ represents the transformed $d$-dimensional sketches.

A draft scheme of file sketching is to construct a sample space $\tilde{F}$, which collects all possible unique file chunks. In
this way, any file can find its partitioned chunks from $\hat{F}$ exactly and then construct the sketch by setting that position as 1. The file similarity can be measured by calculating the Hamming distance between a pair of sketches. However, this taken-for-granted design incurs the following challenges.

**Challenge 1:** For an online-arriving file, we cannot predict its arrival time, file size, and the most important, its content in advance. Therefore, we are unable to construct the sample space before all the files in $A$ have arrived.

**Challenge 2:** For the massive and high-dimensional big data at the network edge, the total number of chunks is tremendous. Even if the sample space can be constructed, the sequential comparison of the partitioned chunks with the sample space will still cause non-trivial computation overhead.

**Methodology in LOFS:** Due to the difficulties in the construction of sample space and its comparisons, we try to decouple the partitioned chunks with the specific chunk samples. Some hash functions can be utilized in the first layer to reduce the dependency of data encoding on the sample space. Bloom filter (BF) [28] is a proper choice, which has been widely used for set representation and membership queries. Our intention is to sketch the file content without breaking the original file similarities. We found that similar chunks of different files can be hashed to the same locations of BF. Based on this feature, we modify the utilization of classical BF to sketch the file chunks into a fixed-length bit array. In this way, files with more similar chunks would be hashed to the two BF with shorter Hamming distances. Such improvements translate the file into a small and fixed-length bit array, while maintaining the file similarity with the high-dimensional vector representation.

Inspired by this idea, LOFS employs Bloom filter to sketch each file by mapping the chunks into a fixed-length bit vector. Given the file set $A = \{A_1, \ldots, A_n\}$, a Bloom filter represents $A$ with a bit vector of length $d$. All $d$ bits in the vector are initially set as 0. A group of $k_{BF}$ independent hash functions, $h_1, h_2, \ldots, h_{k_{BF}}$, are employed to map each chunk of a given file into $k$ positions in the bit vector, such as MD5 [29], SHA-1 [30] or cityhash [31]. Those hit positions of the bit vector are all set to 1. The 0-1 string derived from the hash functions is exactly the file sketch we needed. For example, as Fig. 4 shows, file $A_1$ is partitioned into 3 chunks, i.e., $B_1, B_2,$ and $B_3$. The three chunks are hashed into the 19-bit vector with two hash functions. Typically, the first chunk $B_1$ is hashed into the 5th and 8th bits of the 19-bit vector, and set the two bits to 1. The other two chunks also hit two positions in the vector, respectively. In this way, the file $A_1$ can be sketched as "00101001000010001000".

Through the first-layer hash mapping, similar files can acquire their sketches with a short Hamming distance. The reason is that, with the same hash functions, the duplicated data chunks would hit the same bits in the Bloom filter. Those same bits will not inflate the Hamming distance between the file sketches. The time complexity in this first layer is $O(\beta \cdot k_{BF})$, where $\beta$ is the number of partitioned chunks for the incoming files.

To reduce the sketch density of big-volume files and alleviate hash collisions in representing the massive data, the sketch length should be large enough. A lengthy Bloom filter can guarantee the accuracy of similarity detection in the backward hash mappings. In addition, the sampling technologies can also be employed to further decrease the density of file sketches. The specific performances of LOFS with different sketch lengths and different sample ratios are evaluated in Section 5. In the next layer, we aim to realize similarity mining between the fixed-length file sketches.

### 3.4 Second Layer: LSH-Based Similarity Mining

The $p$-stable Locality Sensitivity Hash (LSH) [32] is an efficient approximate algorithm for high-dimensional similarity search. Therefore, in this layer, with the $d$-dimensional file sketches acting as the input, we can derive the projected points in the LSH tablespace through LSH functions. The projection distance reflects the file similarities, and the hash mappings are expressed as $\xi_2 : C^d \rightarrow TS$, where $TS$ denotes the LSH tablespace.

LSH has the property that maps similar data into the nearby locations in hash tables with a higher probability than dissimilar ones, which is efficient for the $k$-Nearest Neighbor (kNN) query. The basic idea behind the LSH-based approaches is the application of locality-sensitive hashing functions. In this work, we consider the family of LSH functions based on 1-stable distributions (Cauchy distribution) [33], which suits the distance measure between points in $l_1$ norm. Therefore, given two file sketches $c_{A_1}$ and $c_{A_2}$ in the $d$-dimensional vector space, the sketch distance can be viewed as the Hamming distance, which is widely utilized for representing data similarity. To be specific, $\|c_{A_1} - c_{A_2}\|_1 = \sum_{i=1}^{d} (c_{A_1}[i] \oplus c_{A_2}[i])$.

**Challenge 3:** For $p$-stable LSH, the usual practice is to use $l$ hash tables, and each table contains $k_{LSH}$ hash functions. This would reduce the probability of projecting dissimilar files into the same bucket to $1 - (1 - p^{l_{LSH}})^l$, where $p$ is the probability of file similarity in our case. However, the similarity between sketches cannot be unified in different hash tables. The basic reason is that the projection vectors of different hash tables are randomly selected.

**Methodology in LOFS:** One possible solution for Challenge 3 is to use exactly one hash table. This ensures the consistency of the file similarity, although disobeying the design habits of LSH. However, the intention of using $l$ concatenations of hash functions in the traditional LSH is to increase the overall recall of the kNN search through AND then OR operations on these $l$ hash tables. In the second-layer hash mappings, we only focus on reducing data dimensionality while maintaining the file similarity. Therefore, one hash table with a sufficient number of hash functions ($k_{LSH}$) is competent to fulfill this requirement.

Therefore, we utilize exactly one hash table in the second layer, where $k_{LSH}$ independent hash functions are considered. In the case of 1-stable LSH, which is suitable for the sketches in the $l_1$ norm, each hash function follows the following form for the $d$-dimensional file sketch $c$.
where $\alpha$ is a $d$-dimensional random vector following the Cauchy distribution. $b$ is a real number chosen uniformly from the range $[0, \omega)$, wherein $\omega$ is a large constant. As shown in Fig. 5, with $k_{LSH}$ such hash functions, the hash result is a $k_{LSH}$-dimensional integer vector in the following form

$$g(c) = \{h_{a_1,b_1}(c), \ldots, h_{a_{k_{LSH}},b_{k_{LSH}}}(c)\}.$$  

To hash the data vector $g(c)$ into the LSH tablespace, we utilize another hash function, i.e., $h_{TS} = \alpha' \cdot g(c)$. $\alpha'$ is a $k_{LSH}$-dimensional vector, where each element is also chosen independently from the standard Cauchy distribution. Thereafter, the hash mappings of the second layer would be projected to one specific point in the LSH tablespace. Note that, the tablespace is formed from the LSH-based projection points for all arrived files. The distribution of the tablespace is changed as any file arrives, and the projection point is impacted by the file sketch density and the chosen LSH hash functions. For two similar files with a quantity of same data blocks, the hash result in this second layer would also be adjacent. This benefits the lightweight similarity detection and mining. The time complexity in the second layer is $O(d \cdot k_{LSH})$.

In this second layer, the similarity between two files can be reflected by ranging the distance between the projected points in the LSH tablespace. To be specific, similar files with more duplicated chunks would generate similar file sketches. Then the sketches’ projected points in the LSH tablespace would be closer, based on the LSH-based similarity mining. However, the distribution of the projected points in the LSH tablespace may be uneven, which is unfavorable for load balance among edge servers. In the next layer, we expect to find a method to divide the tablespace into heterogeneous-sized buckets for the load balance purpose.

### 3.5 Third Layer: Capacity-Aware LSH Tablespace Division

In the third-layer hash mappings, with the projected point positions in the LSH tablespace, we aim at partitioning the tablespace into $m$ buckets, each of which corresponds to exactly one edge server with heterogeneous capacities. The files whose projected points fall into the same bucket region would be allocated to one edge server, without the track of fingerprint index deployed at each edge server. The mappings can be denoted as $\xi_3 : TS \rightarrow M$.

**Challenge 4:** In the traditional designs, the bucket size in the LSH tablespace is all fixed. However, the even bucket partition would create a load imbalance between servers. It is because that the distribution of the hash values is not uniform and tends to change as files arrive. If we utilize a fixed partitioning scheme on the LSH tablespace, the storage server that corresponds to the dense-projected bucket would be overloaded.

**Challenge 5:** To measure the storage overhead of each edge server based on the partitioned buckets of the LSH table, three factors should be considered: 1) the bucket region; 2) the distribution of the projected points in that bucket region; 3) the average file volume in each bucket. Any of the three factors is necessary. For example, if we only take the bucket region and the distribution of the projected points as the indexes to partition the bucket regions, the number of stored files of each edge server is balanced, but not the storage overhead.

**Methodology in LOFS:** First, the partition size for each LSH bucket should adapt to the heterogeneous capabilities of the involved edge servers. Note that, the capacities can be reflected as the available storage resources, the available transmission bandwidth, or the comprehensive metric of these two, depending on the practical scenarios. In addition, the average file volume in each bucket can be reflected from the corresponding bucket region. We explain the design ideas in detail as follows.

The output distribution in the tablespace is predictable based on the second layer hash mappings. Given the characteristics of the 1-stable distributions, the projected points in the LSH table, i.e., $\alpha \cdot g(c)$, follows the distribution of $\|c\|_1 X$. $X$ follows the Cauchy distribution [34]. Therefore, with $n$ files, whose file sketches are represented as $c_1, \ldots, c_n$, the projected points in the LSH tablespace is distributed with the Probability Distribution Functions (PDF) as follows:

$$cr(x, 0, \sum_{i=1}^{n} \|c_i\|_1 / n) = \frac{n}{\pi} \sum_{i=1}^{n} \|c_i\|_1 \left(1 + \frac{n}{\sum_{i=1}^{n} \|c_i\|_1}ight)^2,$$

where the location parameter $x_0 = 0$, scale parameter $\gamma = \sum_{i=1}^{n} \|c_i\|_1 / n$. Thus, the cumulative distribution function (CDF) can be easily derived as follows:

$$CR(x, 0, \sum_{i=1}^{n} \|c_i\|_1 / n) = \frac{1}{\pi} \arctan \left(\frac{n x}{\sum_{i=1}^{n} \|c_i\|_1}\right) + \frac{1}{2}.$$  

With the increase of $n$, the scale parameter $\sum_{i=1}^{n} \|c_i\|_1 / n$ converges to a stable value, which is recorded as $r$.

Given the above PDF and CDF of the projected points in the third-layer hash mapping, we can partition the LSH tablespace into $m$ buckets. For example, Fig. 6 presents the PDF of $cr(x, 0, r)$, where the tablespace is partitioned into $m$ buckets with regions: $(-\infty, x_1], [x_1, x_2), \ldots, [x_{m-1}, +\infty)$. The average file volume in bucket $j$ is approximately positively correlated with $x_j$, which is the mean value of $|x|$ where $x \in [x_{j-1}, x_j)$. The reason behind is that big-volume files would be partitioned into a great number of chunks. Hence, more ‘1’s would appear in its file sketch, leading to a large absolute value of the projected points in $TS$. 
can be calculated by adjusting \( m(\beta) \) with
\[
\frac{\pi}{C_1} \approx 1. 
\]

is negligible. For any other file \( x \), we can derive
\[
\left[ \frac{[x]}{C_8} \right] \text{ for stable bucket assignment.}
\]

With the given \( k_{BF}, k_{LSH}, \) and \( d \), the three-layer hash mappings have linear time complexity \( O(\beta) \), where \( \beta \) is the number of partitioned chunks.

The three-layer hash mappings have to make an allocation decision for each arriving file. This decision might frequently change because the scale parameter \( \tau \) would be modified with files arrive. Our insights for reducing the decision-making overhead is to coarsen the decision granularity along two dimensions: (1) spatially grouping requests with similar characteristics; (2) temporally modifying the scale parameter \( \tau \) when a significant change occurs.

3.6 Overhead Reduction Through Granularity Coarsening

The three-layer hash mappings have to make an allocation decision for each arriving file. This decision might frequently change because the scale parameter \( \tau \) would be modified with files arrive. Our insights for reducing the decision-making overhead is to coarsen the decision granularity along two dimensions: (1) spatially grouping requests with similar characteristics; (2) temporally modifying the scale parameter \( \tau \) when a significant change occurs.

Coarsening Spatial Granularity. We coarsen decisions spatially by grouping similar files into the same buckets based on the bucket region as well as the projected points in the LSH table. Specifically, all files whose projected points fall in the same region are grouped in the same bucket. We run a decision-making policy over the buckets rather than individual allocation requests, which ensures that the running time of the decision-making process is always constant and low, rather than growing with the number of files.

Coarsening Temporal Granularity. The scale parameter \( \tau \) can be modified as files arrive, which has a profound impact on the tablespace partition. To decrease the allocation overhead, we choose to update the value of \( \tau \), in other words, the tablespace partitioned strategy, when some factors change by a “significant amount”. The policy for deciding the update frequency is orthogonal to our current strategy, it could be the number of newly arrived files, or the J-S divergence [35] between the new and old distributions of the projected points exceeds a certain threshold. We have empirically observed that the same decision assignment can yield close-to-optimal data deduplication. This policy also lessens the pressure of frequent tablespace partitions.

4 Performance Analysis

In this section, we perform mathematical analyses to guarantee the theoretical effectiveness of LOFS. We first quantify the probability that two random files are sketched the same in Section 4.1. After that, we prove that two file sketches with a shorter Hamming distance will be hashed to the same edge server with a higher probability through Theorems 2 and 3 in Section 4.2.

4.1 The Probability of File-Level Collisions Between Sketches

Theorem 1. File \( A_1 \) is sketched into a \( d \)-bit string \( (c_{A_1}) \) with \( \epsilon \) “1” by \( k \) hash functions in Bloom filter. For any other file \( A_2 \), the probability that \( h_{BF}(c_{A_1}) = h_{BF}(c_{A_2}) \) is negligible.

Proof. For any file \( A_2 \), the probability that it contains \( r \) chunks, which is represented as \( p(r) \), can be derived through counting the occurrence frequency of files with various size in the dataset. When \( r < \epsilon/k \), the maximum number of “1” in the file sketch \( c_{A_1} \) is \( k \times r < \epsilon \), which is not possible to match sketch \( c_{A_1} \). Therefore, we only consider the situations when \( r \in [1, \epsilon/k] \). In these cases, \( h_{BF}(c_{A_1}) = h_{BF}(c_{A_2}) \) only appears when the \( kr \) hashed positions of file \( A_2 \) all fall into the \( \epsilon \) positions that file \( A_1 \) does, while any of the \( \epsilon \) positions cannot be empty. Thus, the probability that \( h_{BF}(c_{A_1}) = h_{BF}(c_{A_2}) \) for the file \( A_2 \) with \( r \) chunks, which is denoted as \( P_{c_{A_1} = c_{A_2}}(k, r, d, \epsilon) \), can be calculated as follows:

\[
P_{c_{A_1} = c_{A_2}}(k, r, d, \epsilon) = \frac{S(kr, \epsilon) \cdot \epsilon!}{d^{kr}}. \tag{7}
\]

The \( S(kr, \epsilon) \) is the second kind of Stirling Numbers [36], whose value can be calculated as:
\[ S(kr, \epsilon) = \sum_{i=0}^{n} (-1)^i \cdot i! \cdot \frac{(\epsilon - i)^{kr}}{(\epsilon - i)!}. \] (8)

Therefore, for any file \( A_2 \), the probability that \( h_{BF}(c_{A_2}) = h_{BF}(c_{A_1}) \) can be calculated from the following equation

\[
\sum_{|k|}^{|r|} p(r) \cdot P_{c_{A_1}}(k, r, d, \epsilon) = \sum_{|k|}^{|r|} p(r) \cdot \frac{e^1_{kr}}{d^r} \sum_{i=0}^{r} (-1)^i \cdot i! \cdot \frac{(\epsilon - i)^{kr}}{(\epsilon - i)!}. \] (9)

This value can go up to an order of magnitude of \( d^{-kr} \) when \( d \gg kr \). Therefore, the probability that \( h_{BF}(c_{A_2}) = h_{BF}(c_{A_1}) \) is negligible, which proves Theorem 1.

This theorem validates the rationality of transforming files into sketches using the Bloom filter. Besides, this theorem also proves that the Bloom filter preserves the similarity between any pair of files satisfactorily.

### 4.2 The Probability of Similarity-Aware Sketch Projection

**Theorem 2.** For any three file sketches \( q, c_1, \) and \( c_2 \), where \( ||q - c_1||_1 = \eta_1, ||q - c_2||_1 = \eta_2, \) and \( \eta_1 < \eta_2 \), we have the following inequation

\[
p(|h(q) - h(c_1)| \leq \delta) \geq p(|h(q) - h(c_2)| \leq \delta) \] (10)

where \( \delta \) is a small constant.

**Proof.** Let \( s(\eta, \delta) := p(|h(q) - h(c_1)| \leq \delta) \) and \( t(\eta, \delta) := p(|h(q) - h(c_1)| = \delta) \), where \( ||q - c_1||_1 = \eta. \) Then the \( s(\eta, \delta) \) can be rewritten as \( \sum_{\eta} t(\eta, x) \). Theorem 2 is guaranteed when the \( s(\eta, \delta) \) monotonically decreases in terms of \( \eta \) for a fixed \( \delta \). We first focus on \( t(\eta, x) \). Note that the LSH hash functions in \( \xi \) are \( |a \cdot q - \alpha \cdot v| \) can be viewed as \( \frac{1}{2} f(\frac{\xi}{\delta}) \), where \( f(x) \) denotes the PDF of the absolute value of the standard Cauchy distribution as follows:

\[
f(x) = \begin{cases} 
0 & \text{if } x < 0, \\
\frac{1}{\pi(1+x^2)} & \text{if } x \geq 0.
\end{cases} \] (11)

For any given \( \delta > 0 \), \( |a \cdot q - \alpha \cdot v| \) lies in the value region of \( [(\delta - 1)a, (\delta + 1)a] \) in order to satisfy \( |h(q) - h(c_1)| \leq \delta \). Therefore, \( t(\eta, \delta) \) can be rewritten as follows:

\[
t(\eta, \delta) = \int_{(\delta - 1)a}^{\delta a} \frac{1}{\eta} f\left(\frac{x}{\eta}\right) \left(\frac{x}{\omega} - (\delta - 1)\right) dx \\
+ \int_{\delta a}^{(\delta + 1)a} \frac{1}{\eta} f\left(\frac{x}{\eta}\right) \left(\frac{x}{\omega} \right) dx. \] (12)

For \( \delta = 0 \), the integral of \( t(\eta, 0) \) from \( [-\omega, 0] \) would be set as 0 directly because \( f(x) = 0 \) when \( x \leq 0 \). Therefore, after summing up all derived \( t(\eta, \delta) \), then \( s(\eta, \delta) \) can be calculated by:

\[
s(\eta, \delta) = \int_0^{(\delta + 1)a} \frac{1}{\eta} f\left(\frac{x}{\eta}\right) dx + \int_{a(\delta - 1)}^{(\delta + 1)a} \frac{1}{\eta} f\left(\frac{x}{\eta}\right) \left(\frac{\delta}{\omega} - \frac{x}{\omega}\right) dx \\
= \int_0^{(\delta + 1)a} f(u) du + \int_{\delta a}^{(\delta + 1)a} \left(\frac{\eta u}{\omega} - \frac{\delta u}{\omega}\right) f(u) du, \] (13)

where \( u = \frac{x}{\eta} \). Therefore, We can get the derivative of \( s(\eta, \delta) \) in terms of \( \eta \) as follows:

\[
\frac{d\eta}{\eta} = \frac{1}{\omega} \left(f\left(\frac{(\delta + 1)a}{\eta}\right) - f\left(\frac{\delta a}{\eta}\right)\right). \] (14)

This value is smaller than 0 for \( \eta > 0 \). The reason is that \( f(x) \) decreases monotonically with \( x \). Therefore, we can take the derivative from \( t(\eta, \delta) \) as described in Proof 4.2. To prove Theorem 3, we can take the derivative from \( t(\eta, \delta) \) in terms of \( \delta \), which is shown as follows:

\[
\frac{dt(\eta, \delta)}{d\delta} = - \int_{(\delta - 1)a}^{\delta a} f(u) du + \int_{\delta a}^{(\delta + 1)a} f(u) du. \] (15)

This value is smaller than 0. Therefore, the value of the result of extensive simulations, which show the efficiency of our proposed hash mapping method over other comparison methods.

### 5 Evaluation

In this section, we empirically evaluate the performance of our LOFS strategy using two real-world datasets. We describe our experimental settings and then present the results of extensive simulations, which show the efficiency of our proposed hash mapping method over other comparison methods.

#### 5.1 Experimental Settings

In our experiments, we use an HP OMEN Desktop PC, equipped with an Intel(R) Core(TM) i7 processor with eight 3.80GHz cores and 64GB RAM. The machine runs Ubuntu Linux 16.04 x64 with 4.15.0 kernel.

**Datasets.** We use two kinds of real-world datasets to evaluate the universality of our LOFS strategy. The dataset information is shown in Table 1. GitHub1 and GitHub2 are both downloaded on GitHub websites, which consist of the zip-compressed source code of projects on some hot topics, such as Localstack on the topic of Amazon Web Service [37]. GitHub1 is composed of all historical versions of 56 projects, ranging from 2 to 191 source code files per project. This kind of dataset represents the entire backup workload that contains all backup versions. For the GitHub2 dataset, only 6 historical versions are maintained for 117 randomly
TABLE 1
The Characteristics of Datasets

| Dataset          | GitHub1 | GitHub2 |
|------------------|---------|---------|
| Total Data Size  | 4.00 GB | 685.36 MB |
| Average File Size| 9.69 KB | 7.37 KB |
| Max. File Size   | 8.59 MB | 2.49 MB |
| Min. File Size   | 1 B     | 1 B     |
| No. File         | 432,484 | 95,179  |
| Average Chunk Size| 3.53 KB | 3.14 KB |
| Global Dedup Ratio| 88.98% | 72.0%   |

selected projects. This dataset represents the backup workload with the retention policy, i.e., retaining the latest backups and deleting backups older than the retention policy. This assists the resource reclamation on the precious storage space at edge, which is more in line with the existing storage systems.

Comparison Methods. We consider five comparison methods. The first method is Global, which discards all duplicates and generates the theoretical maximum duped ratio. The second method is MinHash [21], which selects the minimum hash value of the chunks as the data’s signature. Then, the data would be allocated to the server with the same signature. The third method is PDCSS [9]. In this method, a probabilistic method for computing the cardinality of a multiset is utilized to evaluate the chunk intersection between the arrived data and the storage data at each server. In these two methods, we convert the basic storage unit to a file. We also modify the load balance schemes to support the heterogeneous servers at edge. The fourth method is HashID, where the file storage locations are addressed by file IDs provided by the underlying storage system. This is also the comparison method provided in literature [8]. The fifth comparison method is a Capacity-Aware Storage Strategy (CASS). In this strategy, the online-arriving files would be allocated to the edge server that contains the maximum idle storage capacity so as to balance the storage burden of the involved servers to the maximum extent.

We do not compare the works in literature [15], [16], [17], due to the low writing throughput in the inline deduplication. These works do not fit our requirements for edge storage, where the writing throughput should be a priority. In addition, these works cannot meet the access efficiency. The works in literature [4], [8], [18], [19] are also not included in comparisons. The reason is that such offline storage strategies impose a heavy storage load and long strategy generation latency inevitably. This is inconsistent with our requirement of Online deduplication at edge, where data are expected to be allocated to a proper server timely. In addition, these methods neglect the realization of load balance with heterogeneous server capacities in detail.

Metrics. First, we evaluate the probability of file-level sketch collisions with different sketch lengths using Collision Ratio, which is defined as the probability that different files are sketched as the same. This can verify the effectiveness of our LOFS strategy in terms of file sketching. Thereafter, we compare LOFS with the comparison methods in terms of the deduplication ratio and the standard deviation of load (load std), so as to quantify the rationale realization of space efficiency and load balance, respectively. The Dedup Ratio is defined as the ratio of saved storage space after data deduplication to the original space, which is always less than 1. The Load std is defined as the normalized standard deviation of the occupied storage resources among edge servers. Thereafter, we compute the Hash Time of LOFS to verify that the generation time of the allocation strategy is always constant and low. At last, we evaluate the CPU utilization, which reflects the lightweight computation complexity. Note that, LOFS has easily achieved the access efficiency rationale in the file storage process when the file acts as the minimum unit.

Methodology. Our experiments utilize two different datasets and are built on two server environments with uniform or heterogeneous capacities. In experiments, we first unzip and partition the files into variable-sized chunks using the content-defined chunking approaches [10], [11], [12]. Each chunk is represented by its fingerprint using MD5 coding [29]. We set $k_{BF} = 2$ in the first layer and $k_{LSU} = 8$ in the second layer. The original tablespace partition strategy is constructed based on the first 20,000 files as the training data in each dataset. The partition is thereafter updated considering both the distribution of projected points and the remaining idle storage capacity of each involved edge server. The default update is executed every file arrives.

5.2 Performance Quantification

We conduct large-scale experiments to test the respective performance of LOFS with two different kinds of datasets in terms of the Collision Ratio, Dedup Ratio, Load std, Hash Time, and CPU utilization respectively. The dedup ratio under different sample ratios and coarsen granularities is further exhibited to prove the robustness of LOFS with the reduction of computational overhead. There are ten edge servers in the experiments with uniform or heterogeneous capacities.

5.2.1 Collision Ratio

Table 2 evaluates the hash collisions between file sketches with different sketch lengths using two real-world datasets.

| Datasets | GitHub1 | GitHub2 |
|----------|---------|---------|
| Length of sketch (d) | 500 | 1000 | 2000 | 4000 | 8000 | 500 | 1000 | 2000 | 4000 | 8000 |
| Max # of collisions | 5 | 3 | 2 | 2 | 2 | 3 | 3 | 2 | 2 | 2 |
| Total # of collisions | 2,491 | 594 | 136 | 28 | 4 | 2,224 | 549 | 133 | 22 | 2 |
| Collision ratio | 8.31% | 1.98% | 0.45% | 0.093% | 0.013% | 9.47% | 2.37% | 0.57% | 0.094% | 0.017% |
The Max # of collisions refers to the maximum number of collisions mapped to one single sketch, which reflects the concentration of collisions. When the sketch length for dataset GitHub1 varies from 500 to 8000, the number of hash collisions drops sharply from 2,491 (8.31%) to only 4 (0.013%). For a specific sketch, at most 5 files are hashed to the same positions in the BF table when \( d = 500 \) in the first layer of hash mappings. This number decreases to 2 when the sketch length increases to 8000. The invoked hash collisions become less for the GitHub2 dataset, which has a smaller file average size than GitHub1. Overall, this table verifies that our LOFS strategy incurs a low probability of sketch collisions when the sketch length is appropriately set. For the afterward evaluations, we set the sketch length as \( d = 8000 \) to alleviate hash collisions.

### 5.2.2 Dedup Ratio

Fig. 7 depicts the deduplication ratio for servers with uniform capacities. We take the curves for the GitHub1 dataset as an example. LOFS consistently achieves about 30% higher deduplication ratio than the HashID and CASS; and about 20% higher deduplication ratio than the PDCSS method. Although the dedup ratio of Minhash is slightly higher than that of LOFS, Minhash cannot achieve access efficiency and load balance simultaneously. These verify the performance advantages of LOFS in file allocations. In addition, the LOFS closely tracks the Global method, due to its great similarity-awareness. A small gap is then slightly widened between LOFS and the Global method, because LOFS needs to zone files to maintain load balance. LOFS can realize a nearly 85% deduplication ratio ultimately, which verifies the effectiveness of the three-layer hash mappings.

Fig. 8 further exhibits the deduplication ratio performance when edge servers are associated with heterogeneous storage capacities. The capacities of the ten servers are normalized as 6, 2.5, 0.8, 1.6, 3.2, 2, 1.9, 2.5, 4.5, 5, respectively. It can be seen that the deduplication ratio of LOFS also prevails over the PDCSS, HashID, and CASS satisfactory, which demonstrates the application generality of LOFS. In addition, the CASS method performs better than HashID when facing heterogeneous storage capacities. The root cause is that the CASS method enables servers with abundant storage capacities to store more files, which facilitates local deduplication. In order to save space, we will only exhibit the performance using GitHub2 in the following experiments.

### 5.2.3 Load Std

With the arrival of files, the standard deviation of the server load is exhibited in Fig. 9. The load std of MinHash is always larger than other comparison methods. More seriously, the load std of MinHash is around four times that of others with heterogeneous server capacities, as shown in Fig 9(b). The reason is that the MinHash takes the data deduplication as the center of gravity, but its load balance scheme cannot suit different storage systems. The load std of PDCSS and CASS experiences violent fluctuations with the arrival of files. An example is shown in Fig. 9a, with the consecutive arrival of large files (9563\(^{th}\)), the load std of PDCSS and CASS surges. Nevertheless, the load std of LOFS comes out on top regarding stability because it combines the similarity detection and the load balance (both in uniform or heterogeneous environments) into the file allocations.

Fig. 10 depicts the server load of LOFS for the ten servers with uniform or heterogeneous capacities. The load balance is accomplished well for the LOFS strategy. The surges on server load can be pacified with the subsequent file allocations. Take Fig. 10a with uniform capacities as an example, when the 10000th file (with a big volume) being stored...
at 9th server (S9), the server capacity of S9 shoots up. Thereafter, the subsequent files would avoid being allocated to S9 (the bucket region would shrink), but to its nearby servers, such as S8 and S10. Furthermore, the occupied server load increases steadily and is roughly in line with its storage capacity, which naturally leads to the realization of the load balance rationale.

5.2.4 Hash Time

Fig. 11 evaluates the hash time of LOFS for each online-arriving file. LOFS always generates a fluctuate but overall stable hash time, with an average of 18.320ms and 17.496ms separately for the two datasets. Files with a vast volume trigger the peak hash time. These files will be split into more chunks; thus, the hash time would become longer with more hash calculations. The average hash time for GitHub1 is slightly higher than that of GitHub2. The reason is that the average file size of GitHub1 is larger than that of GitHub2. For both datasets, the overall hash time remains stable, which reflects the constant-time complexity of our LOFS strategy. The performance of LOFS outperforms other existing methods, because it does not detect data redundancies through fingerprint indexing (MinHash) or with excessive rounds of communications (PDCSS). The indexing latency would increase gradually with the inflation of the index volume, while information interaction between the storage nodes would further extend the strategy generation latency. Our LOFS strategy determines the proper allocation strategies with only hash calculations, which provides a lightweight and online file storage scheme with constant and low time complexity.

5.2.5 Performance Under the Reduction of Computational Overhead

To decrease the computation overhead for file-allocation making, Fig. 12 demonstrates the deduplication ratio utilizing the sampling technologies [21] on the partitioned chunks to support the first-layer hash mappings. With the sampling ratio increase, fewer chunks in a file would be included in the construction of the file sketch, resulting in the absence of information used for similarity detection. Specifically, the ultimate deduplication ratio decreases from 65.5% without sampling (LOFS) to about 48.3% at a sampling rate of 1/2 (LOFS-1/2), and 34.8% at a sampling rate of 1/16 (LOFS-1/16). In addition, due to the information absence caused by sampling, the allocation scheme for each file would be impacted. The load std for methods with sampling technologies is higher than that of LOFS, and even doubled when 1/16 chunks are sampled. The specific results are shown in Fig. 12b.

Fig. 13 thereafter illustrates the dedup ratio with different update frequencies for the tablespace partition in the third layer hash mappings. This temporal granularity coarsen can further reduce the computation overhead for file-allocation decision making. When the bucket region is updated when every file arrives, i.e., LOFS, the accuracy of region division is the most accurate, and the control of load balancing is the timeliest. With the update rate slows down, the ultimate dedup ratio declines from the original 65.5% (LOFS) to 61%–62% (LOFS-10, LOFS-100, LOFS-1000, and LOFS-10000). The LOFS-10000 achieves a slightly higher dedup ratio than LOFS initially, benefiting from the rational tablespace partition realized from the adequate training data. However, this situation appears at the expense of the ever-increasing load std, which is shown in Fig. 13b in detail. To conclude, the above two figures (Figs. 12 and 13) illustrate the robustness of our LOFS. This strategy still provides relatively satisfactory performance (dedup ratio and load std) with lower computational consumption caused by sampling technologies or granularity coarsening.

5.2.6 CPU Utilization

Fig. 14 finally evaluates the CPU utilization under our LOFS strategy with different file arrival rates. We assume
that the file arrives following Poisson distributions with different parameters, so as to test the CPU utilization of the running machine under different processing throughput conditions. The LOFS (1), LOFS (5), and LOFS (10) methods represent the LOFS strategy with the file arrival frequency being set as 1, 5, and 10 per second, respectively. To avoid the interference with the CPU usage from other running applications, we also test the CPU utilization of the machine when the hash mappings are not operated, i.e., the Initial method. The initial CPU utilization is relatively stable, while our LOFS strategy generates a fluctuating utilization of the CPU resources. The reason is that the hash operations for large-volume files would cause a relatively high CPU utilization. Furthermore, the augment of the processing throughput would speed up the exhaustion of CPU resources. Specifically, when the arrival frequency increases from 1 per second to 10, the CPU utilization inflates from 4.82% to 14.36%. The figure also shows that the hash operations of the LOFS strategy is lightweight, which only occupy at most 41% CPU resources. This overhead is acceptable for the real-world hash mapping calculations.

In summary, LOFS realizes a lightweight online file storage strategy with constant time, while achieving a high deduplication ratio and satisfactory load balance simultaneously.

6 RELATED WORK

A common practice for data deduplication is to partition arriving files into chunks. The simplest and fastest approach is to break apart the input stream into fixed-size chunks [4], [7], [8], which have been taken in the rsync file synchronization tool [38], [39]. We utilize variable-sized chunking approaches [10], [11], [12] in this paper, which declare chunk boundaries based on the byte contents of the data stream and have been demonstrated to be more effective for similarity detection. This characteristic is especially important for archival storage, where a single backup file is composed of multiple data files stored at different offsets and possibly with partial modifications [40].

The inline deduplication [3], [13], [14], [15], [16] realizes data reduction through index lookups in the critical write path, which leads to an increase of time-consuming I/O operations. To alleviate the impact, Xia et al. [3] try to group strongly correlated small files into a segment and segment large files so as to save ram overhead and improve deduplication throughput. Literature [13] detects both the spatial and temporal locality of data to reduce fragmentation and amortize the lookups. However, these methods can only alleviate the burden in the write path. The index will still grow linearly in theory with files accumulating, which triggers low writing throughput.

Post-process deduplication strategies offer another line of thought, which supports the deduplication can be achieved in the background process. For example, AA-Dedupe [41] reduces the computational overhead and increases the deduplication throughput via clustering the application types. EF-dedup [42] estimates the probability distribution of the sources, thus balancing the network-storage cost of edge deduplication. Literature [17] controls the rate of background deduplication threads and conducts selective deduplication to minimize performance degradation. The above methods are limited in file access efficiency, since the split chunks are distributed evenly in the storage cluster.

There are still other important related works in the deduplicated storage systems. Khan et al. [43], [44] discard duplicated chunks according to their content fingerprints. The utilization of Distributed Hash Table for data placement can realize the load balance in different storage systems. Kaiser et al. [45] present an inline deduplication cluster with a jointly distributed chunk index, which is able to detect as many data redundancies as a single node solution. Literature [46] proposes a lazy data deduplication method. The method buffers incoming fingerprints that are used to perform on-disk lookups in batches, with the aim of improving subsequent prefetching. However, the access efficiency is not realized in these works, because they cannot retrieve a file from exactly one server.

Access efficiency is addressed in Similarity-Aware Partitioning (SAP) [8], where one file’s chunks are allocated to the same server. SAP models the file similarities as a δ-similarity graph, and then similar files would be clustered into the same server to achieve better deduplication performance. Hot-Dedup [4] continues this work and picks up the relatively hot files (absolutely, as a whole) to edge, considering the file access frequencies. However, these two works do not discuss the load balance issue in detail. In addition, such offline storage strategies are not desirable in real practices, which inevitably impose a heavy storage load and extended allocation latency.

PDCSS [9] is most relevant to our LOFS in recent years. It also focuses on the selection of storage servers in the post-process deduplication mode. A probabilistic method for computing the cardinality of a multiset is utilized to identify the content similarity. The MinHash method [21] selects the minimum hash value of the chunks as the data’s signature. Then, the data would be allocated to the server with the same signature. The common drawback of these two works is that the access efficiency is not satisfied, because the chunks of a file are not stored at exactly one server. Second, the load balance schemes are only applicable for cluster storage systems with uniform server capacities. Third and most importantly, excessive rounds of communications are required to exchange storage messages (PDCSS) or retrieve the fingerprint indexes (MinHash). The information interaction is time-consuming, especially at edge, which impacts both the reading and writing throughput of the storage systems.

Therefore, none of the existing works satisfy the three aforementioned rationales simultaneously. Otherwise, our LOFS adopts the post-process deduplication scheme to avoid the frequent index lookups in the critical write path. In addition, LOFS conducts three-layer hash mappings to provide a lightweight and online file allocation strategy while realizing the three designed rationales. The file similarities can be effectively captured through lightweight hash calculations. This computation-friendly nature of LOFS makes it possible to be deployed online, thus avoiding the incremental space overhead and the subsequent data migration.

7 DISCUSSION

Several uninvolved aspects of our LOFS strategy warrant further discussion. We introduce them from four design standpoints, which also suggest avenues for future work.
Application Scenarios. Our LOFS strategy in this paper is designed for edge storage systems with heterogeneous edge servers. Nevertheless, it is also applicable in traditional cloud data centers. Servers in data centers are generally composed of the same hardware devices, so they tend to have homogeneous storage and service capabilities. In this case, the storage problem in data centers can be viewed as a subset of issues we are addressing. The complex edge environment puts forward a higher accuracy requirement for load balance. Therefore, our LOFS is a more adaptive and flexible file allocation scheme that fits both heterogeneous and homogeneous storage systems.

Data Reliability. The absolute data deduplication may corrupt data reliability because it removes all extra copies appended by redundancy scheme [17]. The first challenge is the high data restore overhead with frequent I/Os, especially for hot data with great access frequencies [4]. The other challenge is the corruption or failure of the underlying storage without data replicas raises a high risk of data unreliability and unavailability. We do not consider these challenges elaborately in this paper. The reason is that we mainly focus on the file storage strategy but not the following data deduplication or storage at the server level. In addition, these challenges have been greatly solved by previous works, such as data replication/eraser coding [17], as well as the two-tier storage hierarchy with prefetch/preconstruct cache [47] and sliding look-back window cache [15], [16]. Nevertheless, the data reliability technologies can be further parallelized across our LOFS as future work.

Transmission Overhead. The allocated edge servers with highly correlated data may not always be close to the point of data generation. The remote data transmission between the two points may delay the delivery latency and augment the network cost. An intuitionistic solution for this puzzle may be partitioning the decentralized edge nodes into several groups, considering the inter-node network cost. The LOFS strategy spawns the implementation just in the range of any single group. Thus, the LOFS can be achieved with an acceptable file transmission overhead. We do not emphasize this problem in this paper, but the consideration of transmission overhead provides an interesting avenue for future work.

Data Migration. Data migration is a common measure to solve the space overflow problem. Within a large-scale deduplicated system, data migration might reallocate tens of terabytes of data over a wide area network with a busy interconnection [20]. Furthermore, to migrate data that share some common contents with others, the shared data must be duplicated and stored separately at the previous and migrated edge servers. This increases the total physical capacity occupation of the storage system. These potentially high costs of data migration justify the importance of the proposed load balance rationale. We do not discuss data migration elaborately in this paper of the current version. The reason is that the achieved load balance performance can alleviate the data skewness and preserve the capacity availability to the utmost.

8 Conclusion

In this paper, we report LOFS, a lightweight online file storage strategy to allocate files to the proper edge servers that reside the similarities. The three-layer hash mapping scheme is presented to achieve the LOFS strategy. With such a scheme, LOFS realizes the design rationales of space efficiency, access efficiency, and load balance simultaneously. We also give performance analyses to guarantee the theoretical effectiveness of LOFS. The comprehensive experiments conducted on two different datasets indicate that LOFS closely tracks the global deduplication ratio and generates a relatively low load std compared with the comparison methods.

References

[1] A. Aral and T. Ovatman, “A decentralized replica placement algorithm for edge computing,” IEEE Trans. Netw. Service Manag., vol. 15, no. 2, pp. 516–529, Jun. 2018.
[2] Z. Tan, J. Zhao, Y. Li, Y. Xu, and S. Lu, “Device-based LTE latency reduction at the application layer,” in Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation, 2021, pp. 471–486.
[3] W. Xia, H. Jiang, D. Feng, and Y. Hua, “SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput,” in Proc. USENIX Annu. Tech. Conf., 2011, pp. 26–28.
[4] S. Li and T. Lan, “HotDedup: Managing hot data storage at network edge through optimal distributed deduplication,” in Proc. IEEE Conf. Comput. Commun., 2020, pp. 247–256.
[5] Y. Zhang, Y. Wu, and G. Yang, “Droplet: A distributed solution of data deduplication,” in Proc. 13th ACM/IEEE Int. Conf. Grid Comput., 2012, pp. 114–121.
[6] H. Yan, X. Li, Y. Wang, and C. Jia, “Centralized duplicate removal via two storage systems with privacy preservation in IoT,” Sensors, vol. 18, no. 6, 2018, Art. no. 1814.
[7] G. Liu, X. Zhu, J. Wang, D. Guo, W. Bao, and H. Guo, “SP-partitioner: A novel partition method to handle intermediate data skew in spark streaming,” Future Gener. Comput. Syst., vol. 86, pp. 1054–1063, 2018.
[8] B. Balasubramanian, T. Lan, and M. Chiang, “SAP: similarity-aware partitioning for efficient cloud storage,” in Proc. IEEE Conf. Comput. Commun., 2018, pp. 592–600.
[9] D. Frey, A. Kermarrec, and K. Kloudas, “Probabilistic deduplication for cluster-based storage systems,” in Proc. ACM Symp. Cloud Comput., 2012, Art. no. 17.
[10] W. Xia et al., “FastCDC: A fast and efficient content-defined chunking approach for data deduplication,” in Proc. USENIX Annu. Tech. Conf., 2014, pp. 1–14.
[11] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, “Ddleta: A deduplication-inspired fast delta compression approach,” Perform. Eval., vol. 79, pp. 258–272, 2014.
[12] Y. Zhang et al., “AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication,” in Proc. IEEE Conf. Comput. Commun., 2015, pp. 1337–1345.
[13] K. Kosiriwong, S. Janggaran, and K. Voruganti, “Dedup: Latency-aware inline data deduplication for primary storage,” in Proc. 10th USENIX Conf. File Storage Technol., 2012, Art. no. 24.
[14] S. Mandal et al., “Using hints to improve inline block-layer deduplication,” in Proc. 14th USENIX Conf. File Storage Technol., 2016, pp. 315–322.
[15] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. C. Du, “Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance,” in Proc. 17th USENIX Conf. File Storage Technol., 2019, pp. 129–142.
[16] Z. Cao, H. Wen, F. Wu, and D. H. C. Du, “ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching,” in Proc. 16th USENIX Conf. File Storage Technol., 2018, pp. 309–324.
[17] M. Oh et al., “Design of global data deduplication for a scale-out distributed storage system,” in Proc. 38th IEEE Int. Conf. Distrib. Comput. Syst., 2018, pp. 1063–1073.
[18] S. Park and C. Park, “Offline selective data deduplication for primary storage systems,” IEICE Trans. Informat. Syst., vol. 99-D, no. 2, pp. 370–382, 2016.
[19] EMC, “Achieving storage efficiency through EMC celerra data deduplication,” White Paper, Mar. 2010. [Online]. Available: https://www.docin.com/p-688598633.html.
A. Nachman, G. Yadgar, and S. Sheinvald, “GoSeed: Generating an optimal seeding plan for deduplicated storage,” in Proc. 18th USENIX Conf. File Storage Technol., 2020, pp. 193–207.

W. Dong, F. Dougis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane, “Tradeoffs in scalable data routing for deduplication clusters,” in Proc. 19th USENIX Conf. File Storage Technol., 2011, pp. 15–29.

D. Bhagwat, K. Eshghi, D. D. E. Longhi, and M. Lillibridge, “Extreme binning: Scalable, parallel deduplication for chunk-based file backup,” in Proc. 17th IEEE/ACM Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst., 2009, pp. 1–9.

M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, “Sparse indexing: Large scale, inline deduplication using sampling and locality,” in Proc. 7th USENIX Conf. File Storage Technol., 2009, pp. 111–123.

D. Guo, J. Wu, Y. Liu, H. Jin, H. Chen, and T. Chen, “Quasi-Kautz digraphs for peer-to-peer networks,” IEEE Trans. Parallel Distrib. Syst., vol. 22, no. 6, pp. 1042–1055, Jun. 2011.

A. Gionis, P. Indyk, and R. Motwani, “Similarity search in high dimensions via hashing,” in Proc. 25th Int. Conf. Very Large Data Bases, 1999, pp. 518–529.

Y. Hua, B. Xiao, D. Feng, and B. Yu, “Bounded LSH for similarity search in peer-to-peer file systems,” in Proc. 37th Int. Conf. Parallel Process., 2008, pp. 644–651.

S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in Proc. 7th Symp. Oper. Syst. Des. Implementation, 2006, pp. 307–320.

L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, “Optimizing bloom filter: Challenges, solutions, and comparisons,” IEEE Commun. Surv. Tutor., vol. 21, no. 2, pp. 1912–1949, Apr.–Jun. 2019.

R. L. Rivest, “The MD5 message-digest algorithm,” Request Comments, vol. 1321, pp. 1–21, 1992.

D. E. E. III and P. E. Jones, “US secure hash algorithm 1 (SHA1),” Request Comments, vol. 3174, pp. 1–22, 2001.

City hash, 2013. [Online]. Available: https://github.com/google/cityhash

P. Indyk and R. Motwani, “Approximate nearest neighbors: Towards removing the curse of dimensionality,” in Proc. 30th Ann. ACM Symp. Theory Comput., 1998, pp. 604–613.

M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions,” in Proc. 20th ACM Symp. Comput. Geometry, 2004, pp. 253–262.

P. Haghani, S. Moshiri, and K. Aberer, “Distributed similarity search in high dimensions using locality sensitive hashing,” in Proc. 12th ACM Int. Conf. Extending Database Technol., 2009, pp. 744–755.

C. D. Manning and H. Schütze, Foundations of Statistical Natural Language Processing, Cambridge, MA, USA: MIT Press, 2001.

A. Tridgell and P. Mackerras, “The rsync algorithm,” Dep. Comput. Sci. FEIT, ANU, Australian Nat. Univ., Canberra ACT, Australia, Tech. Rep. TR-CS-96-05, 1996.

A. Tridgell, “An efficient algorithm for sorting and synchronization,” Ph.D. dissertation, Dept. Comput. Sci., Australian Nat. Univ., Canberra ACT, Australia, 2000.

R. Koller and R. Rangaswami, “I/O deduplication: Utilizing content similarity to improve I/O performance,” in Proc. 8th USENIX Conf. File Storage Technol., Dep. Comput. Sci., 2010, pp. 221–224.

Y. Fu, H. Jiang, N. Xiao, L. Tian, and F. Liu, “AA-Dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment,” in Proc. IEEE Int. Conf. Cluster Comput., 2011, pp. 112–120.

S. Li, T. Lan, B. Balasubramanian, M. Ra, H. W. Lee, and R. K. Panta, “EF-Dedupe: Enabling collaborative data deduplication at the network edge,” in Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst., 2019, pp. 986–996.

A. Khan, P. Hamandawana, and Y. Kim, “A content fingerprint-based cluster-wide inline deduplication for shared-nothing storage systems,” IEEE Access, vol. 8, pp. 209 163–209 180, 2020.

A. Khan, C. Lee, P. Hamandawana, S. Park, and Y. Kim, “A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems,” in Proc. 26th IEEE Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst., 2018, pp. 87–93.

J. Kaiser, D. Meister, A. Brinkmann, and S. Effert, “Design of an exact data deduplication cluster,” in Proc. IEEE 28th Symp. Mass Storage Syst. Technol., 2012, pp. 1–12.

J. Ma et al., “Lazy exact deduplication,” in Proc. 32nd Symp. Mass Storage Syst. Technol., 2016, pp. 1–10.

N. Zhao et al., “DupHunter: Flexible high-performance deduplication for docker registries,” in Proc. USENIX Annu. Tech. Conf., 2020, pp. 769–783.

Deke Guo (Senior Member, IEEE) received the BS degree in industrial engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001, and the PhD degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2008. He is currently a professor with the College of System Engineering, National University of Defense Technology. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of the ACM.

Lailong Luo received the bachelor’s, master’s, and PhD degrees from the National University of Defense Technology, China, in 2013, 2015, and 2019, respectively. In 2018, he was also a visiting research scholar with the National University of Singapore, Singapore. His research interests include probabilistic data structures and data analysis.

Junxu Xia received the BS and MS degrees in management science and engineering in 2018 and 2020, respectively, from the National University of Defense Technology, Changsha, where he is currently working toward the PhD degree with the College of Systems Engineering. His main research interests include data centers, cloud computing, and distributed storage systems.

Siyuan Gu received the BS degree in mathematics from the Officers College of PAP, Chengdu, China, in 2015, and the MS degree from the College of Systems Engineering, National University of Defense Technology, Changsha, China, in 2020. His research interests include edge computing and distributed computing.