The Doctrine of MEAN: Realizing Deduplication Storage at Unreliable Edge

Junxu Xia, Geyao Cheng, Lailong Luo, Deke Guo, *Senior Member, IEEE*, Pin Lv, *Member, IEEE*, and Bowen Sun

***Abstract*—Placing popular data at the network edge helps reduce the retrieval latency, but it also brings challenges to the limited edge storage space. Currently, using available yet not necessarily reliable edge resources is common sense for edge space expansion, while deploying deduplication storage strategies is a general method for better space utilization. However, a contradiction arises when jointly implementing data deduplication with unreliable edge resources. On the one hand, the deduplication policy stipulates that any data chunk can be stored exactly once; on the other hand, the use of unreliable resources imposes that data should be backed up for the seek of file availability. To resolve such contradiction, we propose MEAN, a deduplication-enabled storage system using unreliable resources at the network edge. The core idea of MEAN is to place similar files together for better deduplication and maintain replicas of popular files for higher reliability. We first formulate this problem and prove its NP-hardness, then provide efficient heuristics based on similarity-aware hierarchical clustering. Three different reliability scenarios are comprehensively considered to develop our algorithms. We also implement a prototype system and evaluate the performance of MEAN with a real-world dataset. The results show that MEAN can fortify the file hit ratio under unreliable environments by 77% while reducing the file retrieval delay up to 71%, compared with the state-of-the-art approach.**

***Index Terms*—Deduplication, fault tolerance, storage system, edge computing.**

I. Introduction

With the flourishment of time-sensitive applications (e.g., augmented reality, Internet of Things, and self-driving), placing popular files at the network edge has become one of the ubiquitous paradigms [1], [2], [3], [4], [5], [6]. This can reduce the number of data requests to remote data centers, thereby alleviating network congestion and shortening service delays. A key metric for such a system is the hit ratio, which quantifies the percentage of data requests that can be served at the network edge. Consequently, popular data is welcome to be stored there.

Currently, the edge cluster suffers from limited storage space and cannot cope with the explosive growth of data. It is estimated that the worldwide number of IoT-connected devices will reach 43 billion by 2023 [7], and 75% of data is projected to be created and processed outside the cloud by 2025 [8]. While this can be solved by renting more proprietary resources, it is not always a cost-friendly option for service providers. Therefore, many studies [3], [9], [10], [11] suggest expanding edge storage space by incorporating various available edge resources, even though some resources may be unreliable. They range from the idle resources provided by various enterprises and individuals to the resources reserved for other applications that are not fully utilized yet. This gives content providers both a cost-efficient and instant way to expand their storage space. By storing more files with extended space, this methodology can improve the hit ratio to some extent. The downside, nevertheless, is that many of these resources are often unreliable. Some edge servers may erase the stored content or leave the storage system at any time. Therefore, redundancy should be generated to guarantee file availability.

As a key technology to achieve space efficiency, data deduplication has been adopted by many modern storage systems [12], [13], [14], [15], [16]. A common practice for data deduplication is to split files into multiple fixed/variable-size chunks, and only one copy of each chunk is maintained [17], [18]. This methodology explores similarities between files, allowing the storage cluster to retain only unique data. In recent years, there have also been many efforts to implement deduplication for edge storage systems [2], [19]. It is reported that the redundancy of IoT and multimedia data can be reduced by over 70% with data deduplication [2], [14], [20]. In addition, according to the study conducted by Microsoft [21], the typical space saving is around 50–60% in general file share scenarios, while datasets with high duplication could see optimization rates of up to 95%, or a 20× reduction in storage utilization. Since more files can be stored with limited storage resources, the hit ratio is somehow improved.

In production storage systems, the above methodologies should be combined together so that the space can be extended freely and occupied with deduplicated chunks. However, it should be noted that these two methodologies may present some contradictions. On the one hand, the deduplication policy

Manuscript received 17 December 2022; revised 24 July 2023; accepted 5 August 2023. Date of publication 15 August 2023; date of current version 23 August 2023. This work was partially supported in part by the National Natural Science Foundation of China under Grants U19B2024 and 62002378, and in part by the Research Funding of NUDT under Grant ZK20-30. Recommended for acceptance by O. Ozkasap. *(Corresponding author: Lailong Luo.)*

Junxu Xia, Geyao Cheng, Deke Guo, and Bowen Sun are with the Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: 1922770344@qq.com; chenggeyao13@nudt.edu.cn; guodeke@gmail.com; sunbowen@nudt.edu.cn).

Lailong Luo is with the Science and Technology on Information Systems Engineering Laboratory, National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: luolailong09@nudt.edu.cn).

Pin Lv is with the School of Computer Electronics and Information, Guangxi University, Nanning, Guangxi 530004, China (e-mail: lvpin@gxu.edu.cn).

Digital Object Identifier 10.1109/TPDS.2023.3305460

stipulates that any data chunk can be stored exactly once; on the other hand, the use of unreliable resources imposes that data should be backed up for the seek of file availability. For the extended space, it is not uncommon that a stored chunk becomes unavailable due to hardware failure, software crash, or the space is recycled by its responsible application. As a consequence, all the files which share that chunk will be incomplete and unavailable. Therefore, a crucial question here is: *how should we use the unreliable space at the network edge, to store more deduplicated chunks of more files or to back up chunks in case of failures?*

To resolve the above dilemma, we present MEAN,[1] a deduplication-enabled edge storage system using unreliable resources. With the ambition of a high hit ratio, MEAN takes no extreme policies (neither backing up nor deduplicating all the chunks), while it goes to the middle (replicating a part of chunks and deduplicating the rest). Specifically, MEAN selects the stored files with the joint consideration of file popularity, file similarity, and server reliability. MEAN improves the availability of popular files and takes up less extra space through redundancy, while others tend to be deduplicated for space efficiency.

We first formulate this problem and prove its NP-hardness. Thereafter, to reduce the search space in similarity detection, we propose a similarity-aware hierarchical clustering algorithm. Based on this algorithm, we elaborately design a set of heuristic algorithms to determine which files to store and where to place their chunks or replicas. The algorithms are progressively generalized according to three different reliability scenarios. The core insight is to dynamically compare the hit ratio gains of adding replicas of the already-stored files with those of directly storing a new file. In this way, MEAN can provide a trade-off between file availability and space efficiency, thus improving the file hit ratio under the limited storage space.

The examples of MEAN and its comparisons are illustrated in Fig. 1. We assume that the cloud is a conventional deduplication storage system utilizing either inline or offline deduplication. The cloud partitions files into chunks and selects a subset of frequently accessed files to be placed at the edge. The popularity of each file is predetermined, indicating its expected access frequency in the future. For simplicity, we assume that all chunks are of equal size in this example. The deduplication-aware scheme (a) [2] stores the most popular files and allocates their deduplicated chunks across the edge servers evenly. Such a method can improve the file hit ratio. Nevertheless, it is hard to guarantee the stored file available (whose expected file hit ratio is only around 40.7%). We find that production distributed storage systems [22], [23], [24] typically employ replication for fault tolerance. Additionally, some deduplicated storage systems [25] also consider incorporating a fixed number of replicas for unique chunks to ensure their availability. Inspired by this, one possible improvement is to add replicas for each popular file, as shown in Fig. 1(b). Nevertheless, the expected file hit ratio (around

[1]Mean or moderation is a basic philosophy of Confucian School in Chinese culture. It suggests taking moderate rather than extreme actions so that contradictions or conflicts can be tactfully avoided.

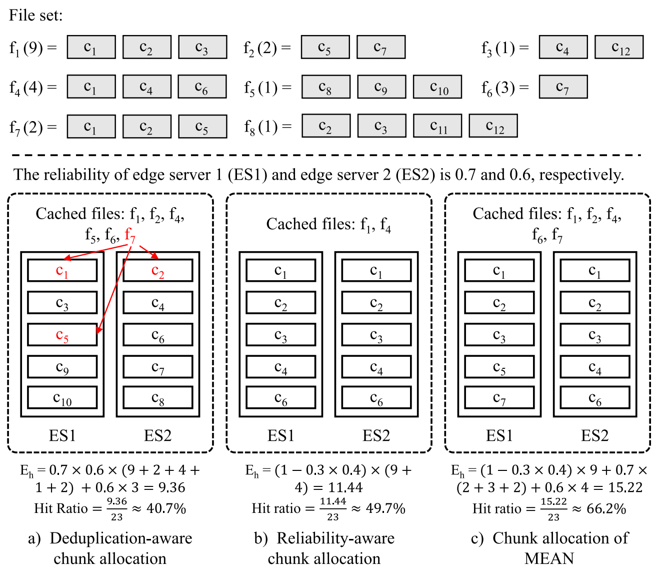

File set:
f1 (9) = c1 c2 c3
f2 (2) = c5 c7
f3 (1) = c4 c12
f4 (4) = c1 c4 c6
f5 (1) = c8 c9 c10
f6 (3) = c7
f7 (2) = c1 c2 c5
f8 (1) = c2 c3 c11 c12
The reliability of edge server 1 (ES1) and edge server 2 (ES2) is 0.7 and 0.6, respectively.
Cached files: f1, f2, f4, f5, f6, f7
ES1 ES2
Cached files: f1, f4
Cached files: f1, f2, f4, f6, f7
Eh = 0.7 × 0.6 × (9 + 2 + 4 + 1 + 2) + 0.6 × 3 = 9.36
Hit Ratio = 9.36/23 ≈ 40.7%
Eh = (1 − 0.3 × 0.4) × (9 + 4) = 11.44
Hit ratio = 11.44/23 ≈ 49.7%
Eh = (1 − 0.3 × 0.4) × 9 + 0.7 × (2 + 3 + 2) + 0.6 × 4 = 15.22
Hit ratio = 15.22/23 ≈ 66.2%
a) Deduplication-aware chunk allocation
b) Reliability-aware chunk allocation
c) Chunk allocation of MEAN

Fig. 1. Illustrative example of MEAN and some comparison methods. Eight files ($f_1 \sim f_8$), attached with their popularity (9, 2, 1, 4, 1, 3, 2, 1), are partitioned into 12 chunks ($c_1 \sim c_{12}$). The storage resources are composed of two edge servers ($ES1$ and $ES2$), each with a storage size of five chunks. The reliability of server $ES1$ and $ES2$ is 0.7 and 0.6, respectively. The aim is to maximize the hit ratio when storing a part of the files at the edge.

49.7%) is just slightly improved due to the decreased number of stored files. Scheme (c) (i.e., MEAN), by contrast, is a relatively superior solution with a maximum file hit ratio of around 66.2%. It eliminates a part of redundancies to free up the storage space, and the most popular file f_1 is replicated to enhance the file availability. Thus, it can achieve an elegant trade-off between space efficiency and file availability. The strengths of MEAN lie in its ability to select stored files dynamically and decide the location and number of replicas for each stored chunk, considering space efficiency and file availability jointly. The major contributions can be summarized as follows.

- We are the first to consider the problem of implementing deduplication-enabled storage with unreliable edge resources. We propose MEAN to realize a high file hit ratio by reaching a balance between replication and deduplication of data chunks.
- We formulate the data deduplication problem at the unreliable edge and prove its NP-hardness. Efficient heuristic algorithms are designed to generate a feasible solution, based on similarity-aware hierarchical clustering.
- We implement a prototype system of MEAN and evaluate the performance under realistic environments with a real-world dataset. The results show that MEAN can fortify the file hit ratio under unreliable environments by 77%, while reducing the retrieval delay by up to 71%.

The rest of this paper is organized as follows. Section II states the related work and motivation. Section III presents the problem formulation and the hardness analysis. Section IV exhibits the heuristic algorithms for the three heterogeneous edge storage scenarios. Section V reports our experimental results, and finally, Section VI concludes this paper. The algorithm of MEAN is available at https://github.com/JX-Xia/MEAN.

II. Related Work and Motivation

In this section, we first introduce the related work and then present the motivation of MEAN.

A. *Related Work*

With the explosive growth of digital data, deduplication [17], [18] has attracted increasing attention in large-scale storage systems to realize the space efficiency rationale. The typical chunk-level deduplication process is to split files or data streams into fixed [26], or variable-size [17], [27] chunks and then calculate their fingerprints (e.g., MD5 or SHA256). Only chunks with the unique fingerprint are stored, while duplicate chunks are eliminated. Such an approach can effectively reduce data redundancy and free up a large amount of storage space. Studies conducted by Microsoft [28], [29] and EMC [30], [31] indicate that about 50% and 85% of the data in the production primary and secondary storage systems can be removed using the deduplication technology. In the general file share scenarios, the typical space savings can be up to 50-60% after deduplication [21].

Due to the competitive advantage, the deduplication technology has also been explored for deployment at the network edge in recent years to address space efficiency. It is reported that more than 70% of redundancy in IoT and multimedia data can be eliminated by data deduplication [2], [14], [20]. Li et al. [19] present a collaborative edge-facilitated deduplication technique to balance the deduplication ratio and the deduplication throughput. Luo et al. [32] propose a graph-based approach to maximize the data deduplication ratio with delay constraints. Cheng et al. [33] design a lightweight three-layer hash mapping method to allocate the most similar files into one edge server for better redundancy elimination. However, these efforts do not take into account the file popularity and thus cannot achieve a higher hit ratio with limited edge storage space. In contrast, HotDedup [2] models the file similarities as a δ-similarity graph, and then allocates files with higher popularity at the edge network. In this way, HotDedup allows more popular files to be stored and thus improving the file hit ratio. Since HotDedup is relevant to the work in this paper, we mainly compare our MEAN method with it. Nevertheless, HotDedup is primarily concerned with space efficiency, while edge storage servers are always assumed to be reliable.

Expanding storage space is another way to make the edge hold more files. Due to the diversity of edge resources, many researches propose to expand edge storage space by using various available resources. For example, Pu et al. [10] advocate edge storage in cloud radio access networks to facilitate mobile multimedia services. Liu et al. [9] propose a cost-efficient edge storage system using embedded storage nodes. Various idle resources and reserved resources are further emphasized in literature [3] to achieve cost-effective space expansion. However, these methods are currently not incorporated with the data deduplication technologies. Thus, the scarce edge resources cannot be fully utilized with the duplicated chunks, especially for the stored files with high similarity. In addition, the diversity of resources makes server reliability a challenge. The storage system should deal with unreliable and dynamic resources to ensure file availability.

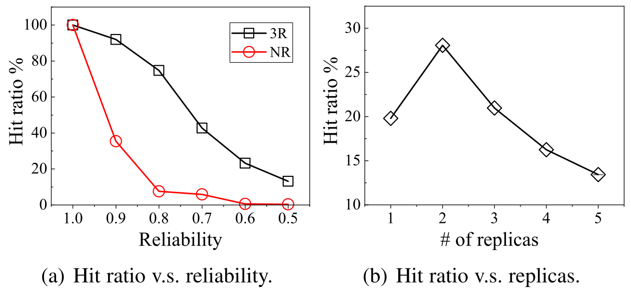


Fig. 2. Impact of server reliability and the number of replicas on the file hit ratio under limited storage space.

Unlike the existing strategies, our proposed MEAN is path-breaking in highlighting both the space efficiency and file availability in edge storage. With the elegant trade-off between these two rationales, the file hit ratio can be maximized. More data retrieval requirements can be served at the nearby edge with less service latency.

B. *Motivation*

While space efficiency and file availability can both effectively improve the file hit ratio, the combination of the two rationals invokes intractable challenges. As illustrated by the example in Fig. 1, data deduplication assists space efficiency, but magnifies the negative impact of data failure in unreliable edge environments. To explore the relationship between the two, we run tests to compare the hit ratio with different server reliabilities and redundancies. For simplicity, we consider the scenario of microservice deployment at the edge, where users or edge servers pull code images from repositories to deploy various container-based microservices. Due to real-time service requirements, these microservices can be short-term and dynamically activated and deactivated, making the data retrieval frequent [34]. We download some popular repositories from the GitHub website [35], as well as some of their most downloaded versions to conduct the test. They are split into chunks using the variable-sized policy [17], which has been widely demonstrated to be more efficient than the fixed-size chunking method [28], [29]. Then, we set up 10 Virtual Machines (VMs) to act as edge storage servers to maintain these repositories. Replication is adopted for fault tolerance, since it is widely proven to have better read/write performance [36], [37].

In each round of experiments, these VMs are randomly shut down according to their reliability. We generate 1,000 retrieval requests on a new VM, which counts as a hit if the required file can be retrieved from these VMs; otherwise, as a miss. The results are based on an average of 100 rounds of experiments. Fig. 2(a) exhibits the hit ratios under different reliability of VMs. The first comparison method is the deduplicated storage without replicas, denoted as NR, where the unique chunks are randomly distributed across 10 VMs. The second is the 3-replica method, denoted as 3R, where each deduplicated chunk maintains 3 replicas across different VMs for fault tolerance. This setting is also implemented by many production storage systems [22], [23], [24] as the default fault tolerance mode. As shown in

Fig. 2(a), the hit ratio of NR experiences a rapid decline from 100% to only 7.56% as the server reliability decreases from 1.0 to 0.8. By contrast, the hit ratio of the 3R method still remains at a high level (74.80%, to be specific) when the reliability is dropped to 0.8. Thus, redundancy can have a positive impact on the hit ratio to some extent.

However, if we supplement the chunk replicas arbitrarily, the extra occupied space of the redundancies would crowd out the original stored content, which is not conducive to the increase of the hit ratio. Therefore, we further conduct tests to observe the impact of the number of replicas. The results are shown in Fig. 2(b). In this set of tests, the total storage space of the 10 VMs is set to 40% of the dataset size, and the reliability of each VM is fixed as 0.8. The file hit ratio grows initially when one more replica is assembled for each unique chunk. However, the excessive redundancy unnecessarily consumes a significant amount of storage space, thereby reducing the capacity of the cluster to hold more popular files. As a result, the file hit ratio declines from 28.07% to only 13.42%, when the number of chunk replicas increases from 2 to 5.

From the above test results, we conclude that *server unreliability can have a significant negative impact on the hit ratio of deduplicated-enabled storage, while replication is a double-edged sword.* Therefore, the trade-off between deduplication and replication should be delicately balanced when implementing deduplication-enabled storage with unreliable storage resources. This problem is intractable, and its complexity would be multiplied when the heterogeneity of file popularity and server reliability is further considered. In addition, although some studies [25], [38] have emphasized the importance of fault tolerance in deduplication storage systems and utilized hash-based mechanisms such as CRUSH or DHT algorithms to determine file or chunk placement, they fail to take into account factors such as file popularity and heterogeneous server reliability, which are crucial for making an elegant balance between deduplication and replication. For example, to enhance the hit ratio, we prioritize adding more replicas to highly popular files to enhance their fault tolerance. We also conduct the deduplication technique for the space-saving purpose, which helps to accommodate more files at the resource-limited edge. The combination of these two achieves an elegant balance between redundancy and deduplication, which promotes the file hit ratio. However, these optimizations cannot find the corresponding operations for the hash-based methods. Therefore, they cannot be directly employed in our scenarios to enhance the hit ratio for edge storage.

To this end, this paper presents MEAN, a deduplication-enabled storage system at the unreliable edge. MEAN leverages replication to enhance file availability, while the deduplication technology is also assembled to eliminate unnecessary redundancies for space efficiency. As far as we know, it is the first work to enhance the hit ratio with joint consideration of both space efficiency and file availability for edge storage. To address this, our MEAN supplements the chunk replicas according to the popularity of different files and data-sharing dependencies with the existing stored content. The chunk locations are also considered to further promote the file availability with heterogeneous server reliabilities.

III. Problem Formulation

In this section, we first formulate the deduplication storage problem in Section III-A. With the problem being defined, we analyze and prove the problem hardness in Section III-B.

A. Problem Formulation

We assume that there is a collection of files $F = \{f_1, f_2, \ldots\}$ that are frequently fetched by users. Their popularities are estimated through statistical analysis within a specific time frame, denoted as $H = \{h_1, h_2, \ldots\}$. A higher level of popularity indicates that the file will be accessed more frequently for a period in the future. There exist many studies that have extensively explored methods to predict the popularity values of files [39]. All these methods can be applied to our work. Since this is not the main concern of this paper, we assume that they are known in advance. Let $C = \{c_1, c_2, \ldots\}$ be the set of unique chunks (after data deduplication) that are partitioned from files in F. The Boolean variable $x_{i,j}$ indicates an inclusion relation, where $x_{i,j} = 1$ means that chunk c_j is included in file f_i. A part of files in F would be stored at the edge to facilitate data requests and reduce retrieval delays. This constitutes a two-tier storage architecture: the cloud data center keeps the whole set of files and is considered infallible, while the edge cluster stores only some popular files and may suffer from data loss. Due to the high bandwidth and low latency of edge storage, file requests are preferentially responded to by the edge servers. If the file is not stored at the edge, or if the server fails and data is lost, the request is further forwarded to the cloud data center. The deployment of edge storage can be performed during off-peak hours to reduce the traffic pressure on the backbone network [3], [4]. The service provider can specify the update intervals between two deployments to achieve a balance between traffic overhead and service performance. The edge metadata overhead due to deduplication mainly comes from two aspects. The first is called file recipes, which are effectively a list of per-chunk metadata as they appear in each file stored at the edge. If one chunk exists in a file multiple times, its metadata also occurs multiple times in its file recipe. This helps file reconstruction in the data retrieval process. The second records the mappings between chunks and edge servers, which helps to retrieve the specific chunks based on the recorded address.

The edge resources are composed of a set of edge servers $S = \{s_1, s_2, \ldots\}$. The storage capacity of server s_k is denoted by $cap(s_k)$. Let Boolean variable $y_{j,k}$ indicate whether chunk c_j is stored at the edge server s_k. We do not consider the case that a chunk or file is replicated several times at one server, because it has no effect on the access shunt but only aggravates data redundancies. Let $size(c_j)$ denote the size of chunk c_j, then the storage cost of server s_k is the total size of the chunks stored on it, i.e., $size(s_k) = \sum_{c_j \in C} y_{j,k} \cdot size(c_j)$. Note that, this size function generates a constant value for the fixed-size chunking [26],

TABLE I
NOTATIONS OF MODEL FORMULATION

Symbol:	*Description:*
Index:	
i	The index of files
j	The index of chunks
k	The index of edge servers
Sets:	
F	The set of files
H	The set of file popularities
C	The set of unique chunks
S	The set of edge servers
R	The set of server reliability
Parameters:	
h_i	The heat degree of file f_i
r_k	The reliability of server s_k
Boolean variables:	
α_i	Whether file f_i is stored at the network edge
$x_{i,j}$	Whether file f_i includes chunk c_j
$y_{j,k}$	Whether chunk c_j is stored at edge server s_k

and varies for the variable-size chunking algorithms [17]. The specific definitions are summarized in Table I.

A file request can only be hit when all of its referenced chunks are available at the edge. This depends on two critical preconditions. The first is that all referenced file chunks are stored at the edge. We let α_i indicate whether this precondition is satisfied. When $\alpha_i = 1$, any chunk c_j of file f_i ($x_{i,j} = 1$) should be stored on at least one edge server, i.e, $x_{i,j} \cdot \sum_{k=1}^{|S|} y_{j,k} \geq 1$. The file f_i contains in total $\sum_{j=1}^{|C|} x_{i,j}$ chunks. Therefore, the Boolean variable $\alpha_i \in \{0,1\}$, $\forall f_i \in F$ can be determined by

$$\alpha_i = \begin{cases} 1, & if\ \sum_{j=1}^{|C|} x_{i,j} \cdot Bool\left(\sum_{k=1}^{|S|} y_{j,k}\right) = \sum_{j=1}^{|C|} x_{i,j}. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

where the $Bool$ function returns “1” when its variable is not zero.

The second precondition is that, for $\sum_{k=1}^{|S|} y_{j,k}$ servers holding chunk c_j, there must be at least one available server when retrieving file f_i with $x_{i,j} = 1$. We use $R = \{r_1, r_2, \ldots\}$ to denote the reliability of each edge server. In this paper, we consider that the unreliability of edge servers mainly stems from two aspects. One is the inherent properties of edge servers. For instance, service providers may deploy some inexpensive servers at the edge to reduce costs, and their reliability can typically be inferred from historical data or equipment manufacturers. Another type of unreliability stems from the fact that these storage resources are either idle resources provided by various enterprises and individuals or proprietary resources reserved for other applications that have not been fully utilized yet. A distinctive feature of such resources is that these storage resources may be reclaimed at any time by the owners or used for other applications. The reliability of such resources can be ascertained through prior negotiation, since obtaining permission to use these resources requires consent from their owners. Therefore, we assume that the reliability of edge servers is predetermined in this paper. Let $P(f_i, x, y, R)$ indicate the file availability of f_i under the server reliability R. It depends on the data sharing dependencies between the stored files (Boolean x), and is deeply associated with the locations of its contained chunks and their replicas (Boolean y). There is currently a lack of closed-form quantification, but the state interrelations of $P(f_i, x, y, R)$ can be estimated coarsely as the reliability product of the stored servers without loss of generality, i.e.,

$$P(f_i, x, y, R) = \Pi_{s_k \in S(i)} r_k, \tag{2}$$

where $S(i)$ denotes the minimum set of servers that can cover all chunks of file f_i.

With the aforementioned Boolean variables about files and chunks, we can formulate the deduplication storage problem as follows.

- When a file is stored ($\alpha_i = 1$), all of its partitioned chunk c_j should have at least one replica at the edge:

$$\sum_{k=1}^{|S|} y_{j,k} \geq \alpha_i \cdot x_{i,j}, \quad \forall f_i \in F \ \&\ \forall c_j \in C. \tag{3}$$

- When the chunk c_j is not referenced by any stored file, it is unnecessary to store c_j at the edge. In addition, for any necessary chunks, there are at most $|S|$ replicas across the $|S|$ edge servers

$$\sum_{k=1}^{|S|} y_{j,k} \leq |S| \cdot \sum_{i=1}^{|F|} \alpha_i \cdot x_{i,j}, \quad \forall c_j \in C. \tag{4}$$

- The total size of chunks stored on each edge server cannot exceed its storage capacity

$$\sum_{c_j \in C} y_{j,k} \cdot size(c_j) \leq cap(s_k), \quad \forall s_k \in S. \tag{5}$$

- The state variables are all Boolean

$$\alpha_i, x_{i,j}, y_{j,k} \in \{0,1\}, \ \forall f_i \in F, \ c_i \in C, \ s_k \in S. \tag{6}$$

We develop the optimization objective of our MEAN scheme, i.e., maximize the file hit ratio, as follows:

$$\max \ \frac{\sum_{f_i \in F} \alpha_i \cdot h_i \cdot P(f_i, x, y, R)}{\sum_{f_i \in F} h_i}. \tag{7}$$

This requires an elegant trade-off between space efficiency and file availability rationales. The space efficiency is described as maximizing the number of stored files, i.e., α_i. The file availability can be represented by maximizing the reliability of each store file, i.e., $P(f_i, x, y, R)$. In addition, placing popular files at the edge can serve more data requests per unit of time, i.e., h_i for file f_i, which further augments the file hit ratio. Then the deduplication storage problem can be formulated with (7) as the objective and (3)∼(6) as the constraints.

B. Problem Analysis

In this subsection, we analyze that the defined problem is NP-hard by proving the hardness in a particular case, i.e., there is no redundancy between the files, and all servers are reliable.

Theorem 1: The problem of deduplication storage with unreliable resources is NP-hard.

Proof: We prove that the problem is NP-hard by showing that a special version of the proposed storage problem can be simplified as the knapsack problem, which is known to be NP-hard [40]. The special case is when zero redundancy exists between any pair of files, and all servers are reliable. Specifically, we consider a set of $|F|$ items, each of size $size(f_i)$ and associated with a reward h_i. The knapsack size is set as M. The knapsack problem aims to find a subset $SF \in F$ of the items that have a total size no greater than M and achieves maximum reward, i.e., to maximize $\sum_{i\in SF} h_i$ under $\sum_{i\in SF} size(f_i) \leq M$. This knapsack problem can directly correspond to the simplified problem when no duplicated chunks exist between any pair of files and the storage resources are all reliable. Thus, the server capacities can be directly quantified as $\sum_{s_k\in S} cap(s_k) = M$. As a result, the knapsack problem can be exactly viewed as a special case of the proposed deduplication storage problem, which implies NP-hardness. □

Although the knapsack problem has been extensively studied, the deduplication storage problem in this paper is much more complex than the knapsack problem. The similarity of files can significantly increase the complexity, and the placement of chunks can lead to different file reliability, which further complicates the problem. Therefore, we provide efficient heuristic algorithms based on the similarity-aware hierarchical clustering in Section IV to enhance the file hit ratio for the deduplication-enabled storage in unreliable environments.

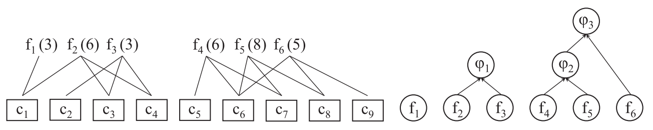
IV. The MEAN Methodology

In this section, we first present similarity-aware hierarchical clustering (SHC). Based on this, we propose the MEAN methodology according to three different reliability scenarios. The three scenarios are progressive, where the latter is a generalization of the former.

A. *Similarity-Aware Hierarchical Clustering*

The algorithms of MEAN are derived from the greedy idea that the storage scheme with the highest gain is selected at each step until the storage space is full. We define a *ranking index* $h \cdot \Delta p/\Delta c$, which indicates the gain of the hit ratio per unit of storage space. The files with higher ranking indexes are more beneficial to be stored at the edge, where h indicates the file popularity, Δp represents the increment of file availability, and Δc denotes the extra space cost. The popularity of a file can be estimated based on its access frequency within a specified period [39], which is assumed to be predetermined in this paper. Δp is calculated by comparing the file's reliability before and after adding new chunks to the storage system, which is discussed in detail in Section IV-B2. Δc denotes the data size required for storage by the system.

However, two main challenges exist when employing the ranking index directly for searching candidate files. The first is that the Δc value is determined based on the difference with the existing stored data. This makes it hard to select a set of closely related (with a great portion of shared chunks) but big-volume files, because there are few co-existing data chunks with the stored files. The second challenge is that the searching process is time-consuming for the numerous file candidates, especially when calculating the Δc value by comparing the contained



(a) Similarity between files. (b) Hierarchical Clustering.

Fig. 3. Illustrative example of hierarchical clustering.

chunks between the file candidates and the stored content. Besides, the calculation is repeated because the Δc value would be updated after each decision with the stored chunks increase. To handle these two challenges, we first propose a Similarity-aware Hierarchical Clustering (SHC) method, which clusters multiple closely related files in advance. This can improve the chance for big-volume files to be selected through deduplication among them. Then, we design an acceleration scheme based on Bloom Filter to reduce the time complexity of file comparison.

1) Clustering Based on Ranking Index: We show the example in Fig. 3(a) to illustrate the first challenge, assuming that all chunks have the same size of one unit. For simplicity, we ignore the server reliability here. The edge storage capacity is assumed to be 5 chunks. Based on the ranking index, file f_1 will be selected first, with the maximum ranking index value of $h/\Delta c = 3/1 = 3$. Since chunk c_1 has been chosen, files containing this chunk will have a Δc value smaller than their actual size in the following computation. Therefore, files similar to file f_1 can have a higher probability of being selected to be stored at the edge. Therefore, file f_2 is chosen, which only needs to store two new chunks c_3 and c_4, with ranking index value $h/\Delta c = 6/2 = 3$. In the same manner, file f_3 will be selected in the next round, which requires a new chunk c_2 to be stored. As a result, file f_1, f_2, and f_3 would be successively selected to be stored at the edge with the total popularity of $3 + 6 + 3 = 12$. However, a superior solution can be to store files f_4, f_5, and f_6, with a total popularity of $6 + 8 + 5 = 19$. The reason for this difference is that files f_4, f_5, and f_6, which are popular and closely related, cannot be detected. Each of these files consists of multiple chunks and therefore has a large Δc value. Thus, each of them has a small ranking index value, making them ignored by the ranking-based heuristic.

To improve the performance of the heuristic, we propose a similarity-aware hierarchical clustering. Hierarchical clustering [41] is an iterative clustering process. In each iteration, it merges the most similar pair of clusters or files (share a large portion of their chunks) into a new cluster. Considering the popularity of different files, we improve it in a popularity-driven manner, i.e., the most similar cluster pairs are merged only if the combined ranking index is greater than each previous one. The updated popularity h is the sum of that of the two original clusters, while c is the size of their union set. The reliability of the updated cluster should be recalculated based on the locations of the chunks in this union set (see Section IV-B). In this way, the number of generated clusters is much fewer than the number of original files, thus significantly decreasing the computation complexities in comparing the ranking indexes. In addition, the updated index value of the cluster is generally larger than before, because the value of c can be greatly reduced after data deduplication. This facilitates the detection of large-volume but closely-related files, like $f_4 \sim f_6$ in Fig. 3(a).

The similarity-aware hierarchical clustering relies on a similarity function that indicates which pair of clusters to merge in each iteration. For this purpose, we use the commonly used Jaccard similarity coefficient [42]. For two clusters A and B, the Jaccard similarity coefficient of them is defined as $J(A, B) = |A \cap B|/|A \cup B|$. To derive the intersection set and union set of the two clusters, an intuitive method is to compare the chunk fingerprints (e.g., using MD5 [43] or SHA-1 [44] coding). The intersection of two files can be calculated as the sum of the sizes of chunks in both files that have the same fingerprint. The union of two files is the sum of the sizes of their unique chunks. In each round, we can choose the two files with the largest Jaccard similarity coefficient to merge. If the merged ranking index is larger than either of the previous ones, the two files are merged to generate a new cluster. Otherwise, the Jacquard index between them is set to 0, and the two files are not considered to be merged in the following rounds.

Take Fig. 3(a) as an example, the Jaccard similarity coefficient of files f_2 and f_3 is $J(f_2, f_3) = 2/4$. After merging them, the ranking index value is $(6 + 3)/4$, which is greater than each of their values, i.e., 2 and 1, respectively. Therefore, files f_2 and f_3 can be merged to generate a cluster φ_1. Thereafter, φ_2 and φ_3 are sequentially constructed, resulting in a final set of three clusters: $\{f_1, \varphi_1, \varphi_3\}$, as shown in Fig. 3(b). Therefore, we can choose the cluster φ_3 with the maximum ranking index to store at the edge. The total popularity is $6 + 8 + 5 = 19$, which is greater than that of the basic heuristic, i.e., 12 as aforementioned. It is worth noting that the maximum number of clustering hierarchies can be preset so that the cluster sizes are within a reasonable range. For the example in Fig. 3, we can stipulate that the files are merged at most once. Thus, SHC outputs only two clusters, i.e., φ_1 and φ_2. When the total storage space is 4 chunks instead of 5 chunks, this approach can still achieve elegant performance.

2) BF-Based Sketch and Acceleration: Although hierarchical clustering can reduce the space of comparison, the calculation of the Jacquard index may still bring significant time overhead. For example, for two clusters φ_1 and φ_2 with $|\varphi_1|$ and $|\varphi_2|$ chunks, it takes $O(|\varphi_1| \times |\varphi_2|)$ time-complexity to determine the number of shared chunks. To further decrease the computation complexity, we adopt Bloom Filter (BF) [45], [46] to sketch the fingerprints of chunks in each cluster. The basic BF is a hashing mapping method that has been widely utilized in various networking and distributed systems. We transfer a similar idea to the process of file comparison and propose the BF-based sketch. This facilitates the computation of the Jaccard similarity coefficient from pair-wise fingerprint checking to the membership queries on the cluster sketches.

When calculating the intersection set between cluster φ_i and φ'_i, it would first require the BF vector of φ'_i. For any chunk c_j in φ_i, the BF judges that this chunk does not belong to φ'_i, if any bit at the k_{BF} hashed positions in that BF vector is 0, where k_{BF} denotes the number of hash functions used by BF. Otherwise, the BF believes that the queried chunk c_j belongs to

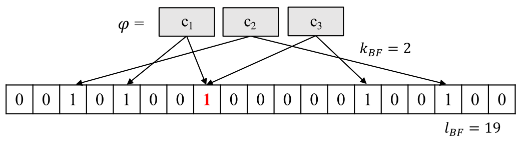


Fig. 4. Illustrative example of the BF-based cluster sketch. Note that the 8th bit of the sketch suffers from the hash collisions.

φ'_i with a rate of false positives. Then, the belonged chunks of φ'_i are exactly the intersection set between φ_i and φ'_i.

Fig. 4 provides an illustrative example of the BF-based sketch. Given a cluster with the chunk set $\varphi = \{c_1, c_2, c_3\}$, the BF represents φ with a bit vector of length $l_{BF} = 19$. All l_{BF} bits in the vector are initially set as 0. The $k_{BF} = 2$ independent hash functions are employed to map each chunk into k_{BF} positions in the bit vector. Those hit positions would be all set to 1. The binary string derived from the hash functions is exactly the BF-based sketch. These sketches would be updated with the XOR operations when the clusters are merged. In this way, each cluster maintains a bit vector to record the membership information at the chunk level. According to the mapped bit vector and the utilized hash functions, we can realize the lightweight membership queries against any clusters, which accelerates the computation for the Jaccard similarity coefficient. To achieve this, we maintain two variables, $d_\cap$ and $d_\cup$. The former is initially 0, while the latter is the sum size of the two clusters. When calculating the Jaccard similarity coefficient of a cluster φ_i with the current cluster φ'_i, we only need to hash each chunk of cluster φ_i to the BF-based sketch of φ'_i in turn. If the query finds that the chunk has been recorded, the value of $d_\cap$ is updated to the original value plus the size of the chunk, and the value of $d_\cup$ is updated to the original value minus the size of the chunk. In this way, the Jaccard similarity coefficient of the two clusters can be represented as $d_\cap/d_\cup$. The time complexity can be decreased as $O(|\varphi| \cdot k_{BF})$, where k_{BF} indicates the number of utilized hash functions.

The penalty of such an approach is the false positive, i.e., for any chunk $c \notin C$, all of its k_{BF} hash positions in the bit vector may be set as 1 when representing other chunks in set C. This is caused by the unavoidable hash conflicts, as the 8th bit in Fig. 4. The false-positive probability, denoted as p, can be derived by $p = (1-(1-1/l_{BF})^{n \cdot k_{BF}})^{k_{BF}}$ [45], where n represents the number of represented chunks in set C.

B. SHC-Based Heuristics for Different Scenarios

SHC elaborates on a feasible and effective method to accelerate cluster generation and index calculation. Based on this, we propose effective heuristic algorithms to improve the file hit ratio in deduplication-enabled storage at the unreliable edge. We consider three heterogeneous scenarios to develop the algorithms of MEAN, where the former scenario is a special case of the latter: 1) all servers are reliable (Section IV-B1); 2) all servers have the same reliability (Section IV-B2); 3) servers are with heterogeneous reliabilities (Section IV-C1).

1) Scenario One: All Servers are Reliable: When all edge servers are reliable, i.e., $r_1 = r_2 = \cdots = 1$, it is unnecessary to

Algorithm 1: Heuristic for Scenario One.

```
   Input: The set of clusters $\Phi$; the set of edge servers $S$.
   Output: The set of selected clusters $\Omega$.
 1 Select an initial cluster ($\varphi_{init}$) whose ranking index $h/c$ is maximum;
 2 Update the cluster set $\Phi = \Phi - \varphi_s$;
 3 The set of already selected clusters is recorded as $\Omega = \varphi_{init}$;
 4 while $size(\Omega) < \sum_{s_k \in S} cap(s_k)$ do
 5     for $\varphi_i \in \Phi$ do
 6         $\Delta c_i = size(\varphi_i - \Omega \cap \varphi_i)$;
 7         Calculate the ranking index of $\varphi_i$ as $h_i/\Delta c_i$;
 8     Select cluster $\hat{\varphi}$ with the maximum ranking index;
 9     Updated the set $\Omega = \Omega \cup \hat{\varphi}$;
10     Update cluster $\Phi = \Phi - \hat{\varphi}$;
11 Distribute $\Omega$ into servers with their capacity constraints.
```

maintain chunk replicas because each chunk is available without the risk of server crashes. Therefore, the ranking index can be directly simplified as $h \cdot \Delta p/\Delta c = h \cdot \Delta c$. Besides, the location of chunks would no longer affect the file availability, because users can retrieve these chunks no matter which edge server they are placed on. To this insight, we only need to consider how to store more popular files with limited storage space, regardless of the mapping between each chunk and the edge server. The specific algorithm is detailed in Algorithm 1.

The algorithm's inputs include the generated set of clusters Φ from SHC and the set of edge servers S. For simplicity, the file that is not clustered is also treated as a cluster. The objective is to select a part of clusters to be stored at the edge, with the objects shown in (7). We first select an initial cluster φ_{init} with the maximum ranking index (Line 1). Thereafter, we calculate and update the ranking index $h/\Delta c$ for all candidate clusters $\varphi_i \in \Phi$, where $\Delta c_i = size(\varphi_i - \Omega \cap \varphi_i)$ is derived from the intersection operation between the set Ω and the current cluster φ_i. Based on this, we select the cluster with the maximum ranking index consecutively, until the size of set $size(\Omega)$ reaches the total storage capacity $\sum_{s_k \in S} cap(s_k)$ (Lines 4-10). The set Ω and candidate clusters Φ are also updated in each round of cluster selection (Lines 9-10). At last, the set of selected clusters Ω would be distributed to the edge servers randomly with the constraint of their storage capacities (Line 11).

2) Scenario Two: Homogeneous Reliability: In contrast to Scenario One, the impact of server reliability is further taken into account. As edge servers are no longer assumed to be reliable, files stored on them may be susceptible to loss. Therefore, enhancing file availability by incorporating chunk replicas is necessary.

Specifically, when all servers have the same reliability, i.e., $r_1 = r_2 = \cdots = r$, a critical measure to enhance the file availability is to hold chunks of a file on as few servers as possible. The reason is that, as the number of servers in $S(i)$ (the minimum set of servers that can cover all chunks of file f_i) reduces, the file availability can be strengthened, as shown in (2). By contrast, when chunks of a file are distributed across many servers, the file can be acquired only if all of these servers are available. Based on this, an effective approach to improving the availability of a file is to place all its chunks on the same server. However, a significant drawback of this approach is that it can result in a significant number of redundant chunks being repeatedly stored across multiple servers, which is not conducive to the edge cluster storing more files.

We design a dynamic trade-off solution to tackle such a conflict. The basic idea is that, for the file with high popularity, we tend to store all of its chunks in the same server to improve reliability. If some files are extremely popular, we can even keep their replicas on multiple servers to further address the file availability. By contrast, less popular files can be stored using data deduplication, where only the unique chunks that make up these files are added to the edge cluster (which may not reside on the same server), thereby enabling the edge cluster to accommodate more files. We further propose three judging metrics to determine the specific storage solution for each file according to three different cases. In addition, the mapping relationship between chunks and servers should be taken into account in this scenario, i.e., which chunks are stored by each server. To this end, we number these servers beforehand and then determine which chunks each server should store. Since the reliability of servers is assumed to be consistent, we ignore the effect of the numbering sequence on the results. For the first server, it dose not know what chunks the other servers will store. Therefore, we still use the method presented in Algorithm 1 to determine which chunks it should store. Starting with the second server, we show three storage cases of how to select and store the file/cluster dynamically as follows.

- Store the remainder of a new cluster φ_i. This storage scheme suggests that we store a new cluster at the edge with deduplication. The space overhead can be denoted as $\Delta c_i = size(\varphi_i - \varphi_i \cap (\Omega \cup \Omega_{\bar{k}}))$, where Ω and $\Omega_{\bar{k}}$ represents the involved chunks that have been stored at previous servers, and the remainder that will be stored at the current server $s_{\bar{k}}$, respectively. In addition, we can calculate the minimum number of servers that have stored the cluster's involved chunks, which is denoted as θ ($\theta = 0$ when there is no involved chunk on other servers). Thus, the cluster can be retrieved with the reliability of $r^{\theta+1}$, since retrieving this cluster requires all involved servers (including the current server) to be available.
- Store the chunks of cluster φ_i that are already stored on previous servers to the current server. This storage scheme suggests that we should hold the entire data of an already-stored cluster φ_i at the current server, rather than spreading its chunks across multiple servers with deduplication. This enables cluster φ_i can be retrieved from a single server. Notably, this scheme is the reverse process of deduplication in the first case, rather than storing a new cluster. It improves the availability of an already-stored cluster φ_i from $r^{\theta+1}$ to r, and the space overhead, i.e., data that is stored repeatedly, can be represented by $\Delta c_i = size(\varphi_i \cap \Omega)$.
- Add a full replica of an already-stored cluster φ_i. This storage scheme suggests that we should add a full replica of cluster φ_i at the current server, instead of storing a

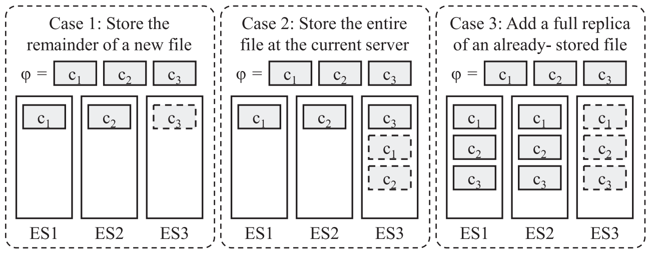

Case 1: Store the remainder of a new file
Case 2: Store the entire file at the current server
Case 3: Add a full replica of an already- stored file
φ =
c1
c2
c3
ES1
ES2
ES3

Fig. 5. Illustrative example of three storage cases for the file/cluster φ. Currently, we need to determine which chunks are stored on the server ES3. The dashed chunks represent the chunks that will be stored next, and the solid chunks represent the ones that have been stored.

Algorithm 2: Heuristic for Scenario Two.

```
Input: The set of clusters $\Phi$; the set of edge servers $S$.
Output: The set of selected clusters $\Omega$.
1  $\omega_k = \emptyset,\ \forall k = 1,\dots,|S|;\ \omega_k \subseteq \Omega$;
2  Select clusters for server $s_1$ according to Algorithm 1. The
     result is denoted as $\omega_1$, and $\omega_1 \rightarrow \Omega$;
3  $\Phi_F = \omega_1,\ \Phi_P = \emptyset$;
4  for $k = 2 \rightarrow |S|$ do
5      while $size(\omega_k) < cap(s_k)$ do
6          for $\varphi_i \in \Phi$ do
7              Case 1: $\Delta p_i = r^{\theta+1}$;
                 $\Delta c_i = size(\varphi_i - \varphi_i \cap (\Omega \cup \omega_k))$;
8          for $\varphi_i \in \Phi_P$ do
9              Case 2: $\Delta p_i = r - r^{\theta+1}$; $\Delta c_i = size(\varphi_i \cap \Omega)$;
10         for $\varphi_i \in \Phi_F$ do
11             Case 3: $\Delta p_i = (1-r)^{\lambda} - (1-r)^{\lambda+1}$;
                 $\Delta c_i = size(\varphi_i - \varphi_i \cap \omega_k)$;
12         Select cluster $\hat{\varphi}_i$ with the highest $h_i \cdot \Delta p_i / \Delta c_i$;
13         If the cluster is stored under case 1, then
             $\Phi_P = \Phi_P \cup \hat{\varphi}_i$ and $\Phi = \Phi - \hat{\varphi}_i$;
14         If the cluster is stored under case 2, then
             $\Phi_P = \Phi_P - \hat{\varphi}_i$ and $\Phi_F = \Phi_F \cup \hat{\varphi}_i$;
15         Update the cluster set $\omega_k = \omega_k \cup \hat{\varphi}_i$;
16     Update the result $\omega_k \rightarrow \Omega$;
```

new cluster at the edge. For particularly popular files, this method can effectively improve their availability. The space overhead of such a scheme can be calculated as $\Delta c_i = size(\varphi_i - \varphi_i \cap \Omega_{\bar{k}})$. The availability of cluster φ_i can be increased from $1-(1-r)^{\lambda}$ to $1-(1-r)^{\lambda+1}$, where λ represents the number of replicas of cluster φ_i at the preceding servers.

C. Time and Space Complexity

The three storage cases are illustrated in Fig. 5, where a new file/cluster φ is required to be stored at the edge. Currently, we should determine how to store the file/cluster φ on the edge server ES3. In the first case, we consider storing a new cluster φ at the edge by only adding the un-stored unique chunks into the edge cluster. Since the involved chunks c_1 and c_2 are already stored, we only need to store chunk c_3 on the current server to achieve space efficiency. The limitation of this scheme lies in the fact that retrieval of the file/cluster is contingent upon the availability of all participating servers. We assume that each server has a reliability of r. Thus, the increased availability Δp of cluster/file φ can be calculated as $r^{\theta+1} - 0 = r^3$, and the storage cost Δc is the size of chunk c_3. In the second case, we consider storing a file/cluster φ completely on the current server, i.e., replicating chunks c_1 and c_2 on the current server ES3. In this way, although the redundancy is increased, the reliability of file/cluster φ is improved. The increased availability Δp of file/cluster φ can be approximately computed as $r - r^3$. It is worth noting that if the file/cluster is stored this way, we will remove it from the set Φ_P and record it in the set Φ_F, which indicates this file/cluster is fully stored on a single server. In the third case, we consider adding a full replica of an already-stored file/cluster in the set Φ_F to the current server ES3. If a file/cluster is stored using this scheme, it indicates that the file/cluster may have a high popularity and therefore is not suitable for distributed storage. In order to ensure high availability, all chunks of this file/cluster replica should be held on a single server. In this way, as long as any involved server is available, the file/cluster φ can be obtained from the edge, which further enhances the availability of popular files. The increased availability Δp can be calculated as $(1-r)^{\lambda} - (1-r)^{\lambda+1} = (1-r)^2 - (1-r)^3$, where two replicas of φ have already been stored on the servers ES1 and ES2.

In particular, considering more cases can potentially promote better solutions, but can bring significant iteration time. Therefore, we mainly consider the above three storage schemes for each file, where the gains of hit ratio are most significant. After evaluating these three storage cases for each file/cluster, we can select the optimal file/cluster and the corresponding storage scheme to maximize benefits and implement it on the edge server. This iterative process sequentially selects one storage scheme at a time until all servers have reached their maximum storage capacity.

The specific procedure is elaborated in Algorithm 2. The input bears resemblance to that of Algorithm 1, while the output determines the stored content at each edge server. We select files to store at the first server according to Algorithm 1 directly (Line 2). Subsequently, we compute the ranking index for each remaining server across all potential clusters in three cases. The cluster with the highest ranking index signifies that its corresponding storage method yields greater benefits by improving the hit ratio while minimizing storage costs. Therefore, we select the cluster with the highest ranking index consecutively, until the size of the stored data reaches the storage capacity of each server (Lines 4-16).

1) Scenario Three: Heterogeneous Reliability: When all servers have heterogeneous reliabilities, i.e., $r_1 \neq r_2 \neq \dots r_{|S|}$, placing files on different servers can result in distinct file availability. The files with higher ranking indexes should be stored on more reliable servers, since such files tend to yield higher storage gains with less space overhead. Otherwise, the availability of these popular files can only be ensured with replicas across multiple unreliable servers, which extra occupies a large volume

TABLE II
TIME AND SPACE COMPLEXITY ANALYSIS

Algorithm	Time complexity	Space complexity
Similarity-aware clustering	$O(\|F\|^2 \cdot \|C_F\|_{max} \cdot k_{BF})$	$O(\|F\| \cdot l_{BF})$
Heuristics: Scenario One	$O(\|\Phi\|^2 \cdot \|C_\Phi\|_{max} \cdot k_{BF})$	$O(\|\Phi\| \cdot l_{BF})$
Heuristics: Scenario Two	$O(\|\Phi\|^2 \|S\| \cdot \|C_\Phi\|_{max} \cdot k_{BF})$	$O((\|\Phi\|+\|S\|) \cdot l_{BF})$
Heuristics: Scenario Three	$O(\|\Phi\|^2 \|S\| \cdot \|C_\Phi\|_{max} \cdot k_{BF})$	$O((\|\Phi\|+\|S\|) \cdot l_{BF}+\|\Phi\|\|S\|)$

of precious storage resources. To this insight, we sort these servers in descending order according to their reliability. Starting with the most reliable server, we progressively select files based on improvements to the scheme in Scenario Two. The three different cases in this scenario are presented as follows.

- Store the remainder of a new cluster φ_i. The space overhead can be denoted as $\Delta c_i = size(\varphi_i - \varphi_i \cap (\Omega \cup \Omega_{\bar{k}}))$. The file availability can be derived by $r_{\bar{k}} \cdot \Pi_{s_k \in \theta_i} r_k$, where the minimum number of servers that have stored its involved chunks is denoted as θ_i ($\theta_i = 0$ when there is no involved chunk on other servers).
- Store the chunks of cluster φ_i that are already stored on previous servers to the current server. It improves the availability of an already-stored cluster φ_i from $r_{\bar{k}} \cdot \Pi_{s_k \in \Theta_i} r_k$ to $r_{\bar{k}}$, and the space overhead, i.e., data that is stored repeatedly, can be represented by $\Delta c_i = size(\varphi_i \cap \Omega)$.
- Add a full replica of an already-stored cluster φ_i. The space overhead can be calculated as $\Delta c_i = size(\varphi_i - \varphi_i \cap \Omega_{\bar{k}})$. The availability can be increased from $1 - \Pi_{s_k \in \Lambda_i}(1 - r_k)$ to $1 - (1 - r_{\bar{k}}) \cdot \Pi_{s_k \in \Lambda_i}(1 - r_k)$, where Λ_i represents the set of previous servers that store the replicas of φ_i.

The algorithm in this scenario is quite similar to Algorithm 2. The difference is that servers in the set S should be pre-ordered based on their respective reliabilities, and Δp_i in lines 7, 9, and 11 should be replaced with the above three formulas. Therefore, the algorithm is omitted here. Furthermore, it is worth noting that MEAN can actively create replicas for highly popular files. This increases their availability, and the replicas also help avoid server hot spots. In this way, retrieval requests for popular files can be effectively balanced across multiple servers, thus avoiding the overload on a single server.

We analyze the time and space complexities of the above algorithms in this subsection, with the results being shown in Table II. The similarity comparison between two files/clusters adopts the BF-based sketch. The time complexity of the similarity-aware hierarchical clustering is $O(|F|^2 \cdot |C_F|_{\max} \cdot k_{BF})$, where $|F|$ denotes the number of files, k_{BF} indicates the number of utilized hash functions for the BF-based sketch, and $|C_F|_{\max}$ represents the maximum number of chunks of any file. The space complexity is $O(|F| \cdot l_{BF})$, where l_{BF} expresses the BF length.

The time complexity of the heuristic algorithm in Scenario One is $O(|\Phi|^2 \cdot |C_\Phi|_{\max} \cdot k_{BF})$, where $|\Phi|$ denotes the number of generated clusters and $|C_\Phi|_{\max}$ indicates the maximum number of chunks in any cluster. The space complexity is $O(|\Phi| \cdot m)$. For Scenario Two, the time complexities are multiplied by S, while $|S| \cdot l_{BF}$ additional space is occupied with recording the integrated sketches for data storage at the $|S|$ servers. As for Scenario Three, the space complexity $|\Phi||S|$ is caused by maintaining a list of possible storage servers for each cluster.

V. PERFORMANCE EVALUATION

In this section, we implement a prototype of MEAN and evaluate the performance using a real-world dataset. We describe our experimental settings and then present the numerical results of different methods.

A. Experimental Settings

We implement a prototype system of MEAN to evaluate the performance in real-world environments. The prototype includes a cloud and an edge cluster to simulate the file retrieval behavior of edge storage. This constitutes a two-tier storage architecture, where requests for files are first responded to by the edge storage cluster, and the missed requests are further forwarded to the cloud. The edge cluster stores a subset of popular files, while the cloud keeps a complete backup of all files. In our prototype system, the cloud is deployed on the Elastic Compute Service (ECS) of Alibaba Cloud [47], which is equipped with 2.5 GHz 8 vCPU, 16 GB RAM, and 40 GB SSD. The ECS runs Ubuntu Linux 16.04 x64. The edge storage consists of 11 VMs deployed on a Desktop PC, equipped with a 3.50 GHz Intel(R) Core(TM) i9-11900 K CPU with 8 cores and 64 GB RAM using 500 GB SSD. Each VM is allocated 4 GB of RAM and 30 GB virtual disk drive, running Ubuntu Linux 20.04 x64. The CPU cores are shared by all VMs. We use the `iPerf` and `ping` tools to measure the network performance. The bandwidth between the ECS and the local VMs is 91.6 Mbps, and the latency is 29.05 ms, while the bandwidth between any two local VMs is 1.24 Gbps, and the latency is 1.43 ms, based on the average of 10 measurements.

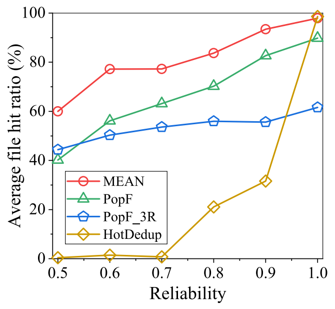
In our experiments, 10 VMs act as the edge storage servers, and the remaining one acts as a data requester to retrieve files from these edge servers or the cloud. This VM also acts as a management node, which is responsible for monitoring the status of other edge servers through heartbeat packets among them. The management node maintains the mapping information between each file and its chunks, as well as the mapping between each chunk and servers in the edge cluster. When the requested file cannot be served by the edge cluster due to failures, the request is further forwarded to the cloud server by the management node.

Datasets: We evaluate the performance of MEAN using two real-world datasets from the GitHub website [35], which are extracted from 357 popular repositories. The first dataset comprises code images in the `.zip` format, referred to as SRC. The second dataset encompasses released installers in formats such as `.rpm`, `.deb`, `.apk`, etc., denoted as RLS. These repositories are selected randomly under some hot topics, e.g., *Azure*, *Amazon Web Services*, *Docker*, etc. We download multiple popular versions from each repository. The SRC dataset comprises a total of 3,099 files with file sizes ranging from 2.74 KB to 12.6 MB. Meanwhile, the RLS dataset contains 1,617 files with file sizes varying between 3.26 KB and 24.1 MB. We chunk these files using the variable-size chunking method [17], which declares chunk boundaries based on the byte contents and is widely proven to be more efficient than the fixed-size chunking method [28], [29]. The SRC and RLS datasets exhibit an average chunk size of 4.07 KB and 3.84 KB, respectively, with deduplication ratios (the size of duplicated data divided by the total size, where a larger value indicates that deduplication can eliminate more redundancy) of 53.01% and 26.03%. The popularity of each file is generated with the widely utilized Zipf distribution [18], [48].

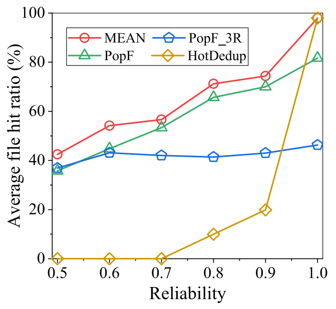
Comparison Methods: We implement the following comparison methods to determine which files the edge cluster stores.

- *HotDedup*, which is an implementation of the HotDedup algorithm [2]. The popularity of the stored files is maximized with the capacity constraints of edge servers. These files are deduplicated in a global sense, and the unique chunks are distributed across the edge servers evenly.
- *PopF*, which selects the most popular files to store at the edge. Such a Popularity-First strategy is widely adopted by many edge storage systems [4], [5], [6]. We improve this strategy by emphasizing file availability and space efficiency. All chunks of the selected file are stored on one server and deduplication is used to eliminate duplicate chunks on each server.
- *PopF_3R*, which is assembled with the replication theory to enhance file availability based on the PopF method. The number of replicas is set to 3, which is the default value in many production distributed storage systems [22], [23], [24].
- *Cloud_only*, which retrieves all requested files from the cloud, regardless of the edge storage.

Settings and Metrics: The results are based on the average from 10 rounds of experiments. The default reliability of the 10 edge servers is set as [0.8; 0.5; 0.7; 0.7; 0.8; 0.6; 0.5; 0.9; 0.4; 0.6], and their total storage capacity defaults to 20% of the dataset size. In each round, we evaluate the performance of different comparison methods by randomly generating 500 file retrieval requests based on their popularity. Then, we randomly shut down part of the edge servers according to the reliability and retrieve files in the retrieval list according to the Poisson distribution. The arrival rate λ is set as 90 by default, which indicates the expected number of retrieval requests in one minute. Our metrics include the *file hit ratio*, the *average retrieval delay*, and the *average retrieval throughput*. Besides, we also consider the balance of service load for the involved edge servers. In each round, we record the total amount of data sent by each server as its service load. Then, we calculate the balance metric ϵ, which is defined as the deviation between the maximum and the average service load.

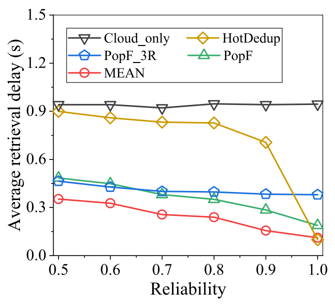


(a) Hit ratio of the SRC dataset.

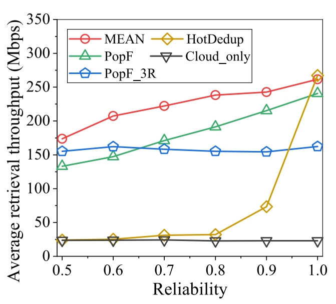


(b) Hit ratio of the RLS dataset.

Fig. 6. Impact of different levels of reliability on file hit ratios across two datasets.



(a) Average retrieval delay.



(b) Average retrieval throughput.

Fig. 7. Impact of different levels of reliability on the file retrieval performance under the SRC dataset.

B. Experimental Results

1) Performance at Varying Levels of Reliability: We first evaluate the sensitivity of different methods to server reliability. To facilitate comparison, we maintain uniform reliability levels across all edge servers and vary this parameter from 0.5 to 1.0. This corresponds to the scenario of homogeneous reliability. When the reliability is set as 1.0, MEAN adopts the algorithm of Scenario One in Section IV-B1. Other reliability settings correspond to the algorithm in Scenario Two, as illustrated in Section IV-B2. The total storage space is set to the default value, i.e., 20% of the dataset size. The results are presented in Figs. 6 and 7, with the former presenting the average file hit ratio across both datasets, while the latter displays the average retrieval delays and throughput for the SRC dataset.

The enhancement of server reliability positively impacts file hit ratio and retrieval performance, while low reliability results in varying degrees of performance degradation for these methods. HotDedup is the most sensitive to server reliability, whose hit ratio is at a low value when the reliability of servers is below 0.9, as shown in Fig. 6. Because of this, the average retrieval delay is just slightly lower than that of Cloud_only, with more than 0.7 seconds per file (in Fig. 7(a)), and the throughput is below 73 Mbps (in Fig. 7(b)). The optimal hit ratio can only be attained under the condition that all servers are reliable. The main reason is that HotDedup assumes that these edge servers are reliable,

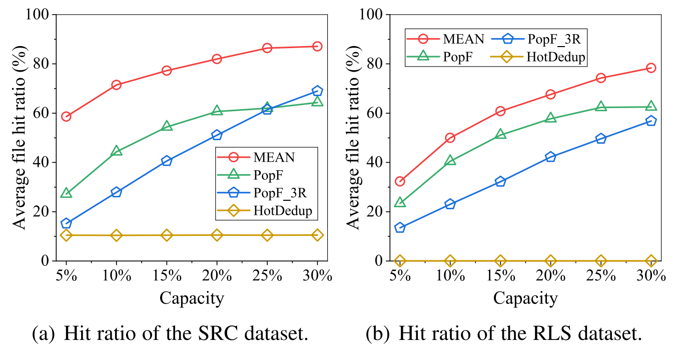


Fig. 8. Impact of different storage capacities on file hit ratios across two datasets.

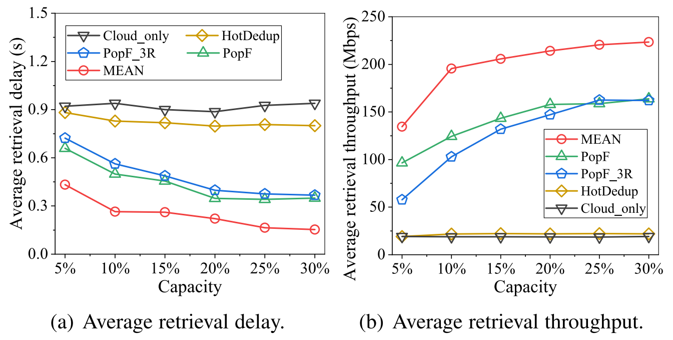


Fig. 9. Impact of different storage capacities on the file retrieval performance under the SRC dataset.

and the chunks of each file are evenly distributed across these servers. Such an approach is not friendly to file availability. The failure of any one server can make a large number of associated files unavailable, which significantly impacts the file hit ratio.

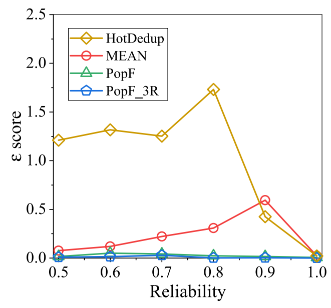
The PopF and PopF_3R methods exhibit a superior hit ratio compared to HotDedup in unreliable environments, resulting in better retrieval delay and throughput performance when server reliability falls below 0.9, as shown in Fig. 7. In particular, PopF_3R exhibits only a slightly superior hit ratio to PopF when the server reliability is at 0.5. This can be attributed to the fact that the three-way replicas policy further bolsters file availability in an unreliable environment, with each popular file being able to withstand up to two server failures. However, such a fault-tolerant approach also brings disadvantages due to extra space occupation. With the growth of server reliability, the PopF method surpasses PopF_3R. When the server reliability exceeds 0.9, there is a significant difference in hit ratio of over 20% and a throughput difference of more than 50 Mbps between the two methods, as illustrated in Figs. 6(a) and 7(b). The reason for this is that with the improvement of server reliability, the advantages of replication decrease. Maintaining three replicas requires a significant amount of space, which limits the number of files stored at the edge and results in a high volume of requests that must be served by the cloud. In contrast, MEAN demonstrates superior file retrieval performance across a wide range of reliability settings due to its ability to efficiently execute data deduplication while adjusting the number of chunk replicas for varying reliability scenarios. However, since the RLS dataset has a lower deduplication ratio than the SRC dataset (i.e., fewer duplicated chunks), the gap in hit ratio between the MEAN and PopF methods is smaller in the RSL dataset compared to that in the SRC dataset, as Fig. 6 shows.

2) *Performance Under Different Storage Capacities:* We set varied storage capacities to evaluate their impact on file retrieval performance. The total storage capacity of edge servers is increased from 5% of the dataset size to 30%, with the server reliability remaining in default. MEAN adopts the algorithm of Scenario Three, as described in Section IV-C1. The results are presented in Figs. 8 and 9.

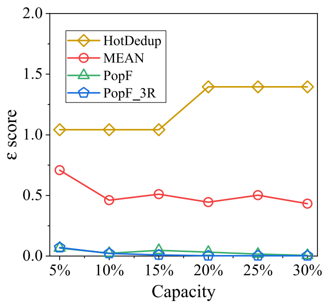
Fig. 8 depicts the impact of storage capacity on the average file hit ratio. MEAN consistently has the highest hit ratio, since it considers both space efficiency and file availability, followed by PopF and PopF_3R. The PopF method has a higher hit ratio than PopF_3R when the edge storage capacity is low. Nevertheless, when the capacity exceeds 25%, the PopF_3R method achieves the reverse in the SRC dataset. This is because when the storage space is large enough, replication can increase the file availability and cope with more server failures. However, neither of the two methods can effectively exploit file similarity to increase space efficiency. Thus, their hit ratios are lower than MEAN under the same server capabilities. In contrast, HotDedup exhibits a relatively low hit ratio, with only approximately 10% in the SRC dataset and close to zero in the RLS dataset. This is due to the fact that the failure of any involved server of a file can easily result in a missed hit. When storage capacity reaches 30%, MEAN and HotDedup differ significantly in their average file hit ratios, with up to a 77% discrepancy observed across both datasets.

Fig. 9 reports the average retrieval delay and throughput in the SRC dataset. When all files are retrieved from the cloud, the average delay is around 0.9 seconds with a throughput of about 19 Mbps. Storing files at the edge can significantly reduce the average retrieval delay. With the storage capacity only accounting for 5% of the dataset size, MEAN can decrease retrieval delays by over 50%, to around 0.43 seconds, while the average retrieval throughput can reach up to 135 Mbps. As the storage capacity increases, this gap gradually widens. Specifically, when the storage capacity accounts for 30% of the dataset size, MEAN exhibits an average retrieval delay of approximately 0.15 seconds. This represents a reduction in retrieval time by 83% compared to cloud-based file retrieval and a reduction of 71% compared to HotDedup. Furthermore, this delay is roughly half of that observed with PopF and PopF_3R methods.

3) *Load-Balancing Performance:* To facilitate comparison, we utilize the SRC dataset as a benchmark for evaluation in subsequent experiments. The load-balancing performance of the compared methods under the SRC dataset is depicted in Fig. 10. Specifically, Fig. 10(a) illustrates the load-balancing performance of the compared methods under different reliability settings. As the server reliability increases, MEAN prioritizes storing new files over adding replicas of stored files. Consequently, the ϵ score of MEAN experiences a gradual and slight

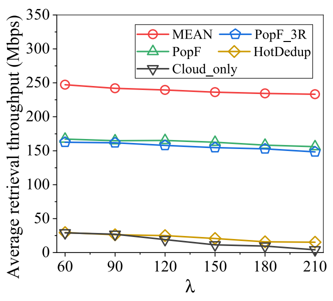


(a) Different reliability.

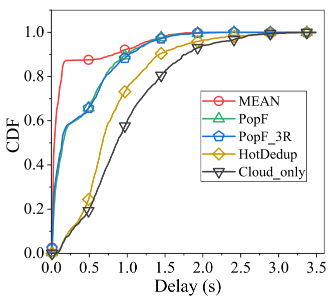


(b) Different storage capacity.

Fig. 10. Impact of reliability and storage capacity on load-balancing performance.



(a) Average retrieval throughput.



(b) CDF of retrieval delays.

Fig. 11. Average retrieval throughput and CDF of retrieval delays.

TABLE III
EVALUATION OF THE AVERAGE RETRIEVAL DELAY AT DIFFERENT ARRIVAL RATES

	Value of λ					
	60	90	120	150	180	210
Cloud_only	0.699s	0.764s	1.126s	2.067s	2.417s	9.351s
HotDedup	0.694s	0.758s	0.851s	1.091s	1.904s	7.622s
PopF	0.334s	0.346s	0.358s	0.364s	0.388s	0.424s
PopF_3R	0.383s	0.371s	0.379s	0.403s	0.434s	0.454s
MEAN	0.196s	0.220s	0.215s	0.223s	0.236s	0.253s

increase. However, when server reliability is set at 1, MEAM reverts back to Scenario One's algorithm where chunks are evenly and randomly distributed across all edge servers. This results in uniform traffic distribution among different servers and promotes better load-balancing performance. In unreliable settings, HotDedup fails to achieve optimal load balancing due to the fact that only a small number of servers handle users' requests while the majority are idle. Therefore, the measurement results will exhibit a relatively large ϵ score.

Fig. 10(b) presents the load-balancing performance under varying storage capacities. PopF and PopF_3R methods exhibit superior load-balancing performance due to their even distribution of stored files across different servers. In contrast, MEAN's ϵ score is slightly higher as it tends to store popular files on more reliable servers while less popular files are placed on lower reliability ones. Nonetheless, MEAN's ϵ score remains relatively stable and reasonable due to its ability to distribute retrieval traffic based on chunk replicas of stored files.

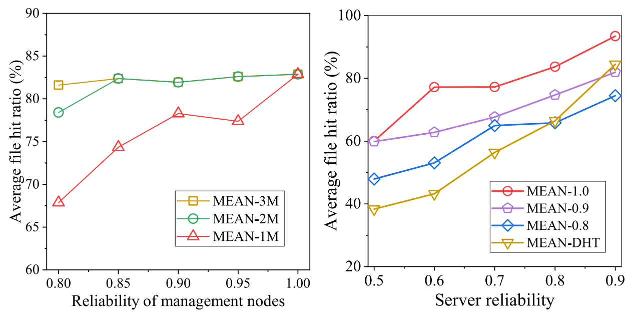
4) Performance Under Different Workloads: We further evaluate the impact of different workloads on file retrieval performance by setting different arrival rates λ. A large arrival rate can effectively reflect a large number of retrieval requests during peak hours. We increase the arrival rate λ from 60 to 210 and then measure the average retrieval delay and throughput of different methods for retrieving 100 files.

Table III illustrates the average retrieval delay under different arrival rates, and the corresponding throughput is presented in Fig. 11(a). With the increase in arrival rates, the retrieval delay of all methods exhibits an upward trend. However, the number of requests that edge storage can serve varies due to different methods of file allocation. As a benchmark, the Cloud_only method forces all retrieval requests to be served by the cloud server, leading to severe congestion on the backbone network. Therefore, it experiences a more rapid increase than others. Besides, since server failures are not considered, HotDedup can only slightly reduce the latency to a limited extent. When the arrival rate is set to 210, the average retrieval delay reaches 9.351 seconds, followed by HotDedup at 7.622 seconds. This is almost 30 times higher than MEAN. The high volume of requests competing for the limited bandwidth of the backbone network also resulted in the inefficiency of average throughput, both of which are lower than 50 Mbps. In contrast, the other three methods can improve file availability because of their fault-tolerant mechanisms. They experience only a slight increase in retrieval delays, while MEAN can further achieve a delay reduction of more than 40%, compared to PopF and PopF_3R.

Fig. 11(b) exhibits the CDF of retrieval delays in one round of experiments, where the arrival rate is fixed as 120. MEAN completes up to 87.5% of requests within 0.5 seconds, compared to only around 65% for the PopF and PopF_3R methods, and less than 20% for the Cloud_only method. In addition, Cloud_only and HotDedup both suffer from long-tail distributions. Their maximum delay is up to 3.5 seconds, compared to around 2.23 seconds for other methods.

5) Failure of the Management Node: In the above experiments, we assume that the management node is deployed on a highly reliable server (such as the proprietary metadata server provided by the service provider) and disregard any potential impact resulting from its failure. Nevertheless, the management node can be a single point of failure in the practical deployment, as it is responsible for retaining all metadata related to files stored at the edge. If the management node fails, it may cause a complete system failure and all retrieval requests will have to be handled by the cloud.

To mitigate the impact of single-point failure, there are two viable approaches. The first is to deploy multiple management nodes at the edge. We vary the reliability of management nodes from 0.8 to 1.0 and conduct 100 rounds of experiments to evaluate the impact of management node reliability. In each round, we randomly designate some nodes (including the management nodes and the storage nodes) as failures based on their reliabilities and generate 1,000 file requests according to the file popularity to simulate data retrievals. The results are presented



(a) Different configurations of management nodes. (b) Comparison between two deployment approaches.

Fig. 12. Effect of two distinct metadata deployment approaches on the average file hit ratio.

in Fig. 12(a), where MEAN-1 M, MEAN-2 M, and MEAN-3 M correspond to the MEAN method with 1, 2, and 3 management nodes, respectively. The storage capacity of the edge cluster is set at 20% of the dataset size while the reliability of each storage server is set at 0.8.

When there is only one management node (MEAN-1 M), variations in reliability can have a significant impact on the average file hit ratio. A decrease in reliability from 1.0 to 0.8 can lead to a reduction of over 10% of the average hit ratio. Deploying more management nodes can significantly enhance the system's robustness and effectively mitigate single-point failure. When the reliability of management nodes is 0.8, deploying two management nodes can achieve a 10% increase in the average hit ratio compared to using a single management node. This indicates that increasing the number of management nodes is an effective strategy for managing the failure of management nodes. However, adopting more management nodes may yield diminishing marginal returns. When the reliability of management nodes exceeds 0.85, there is a negligible difference in the average hit ratio between utilizing two or three management nodes.

The second approach involves utilizing the distributed hash table (DHT) or its variants [49] to store metadata in a decentralized manner. In such a deployment, the metadata of edge files is distributed among edge storage servers based on the hash values of files, following a consistent hashing algorithm. The file can only be retrieved from the edge if both the server retaining its metadata and the servers storing its referenced chunks are available. The experimental results are depicted in Fig. 12(b), where MEAN-DHT employs a DHT-based approach, while the other three comparisons utilize a management node with varying degrees of reliability (e.g., MEAN-0.9 indicates a management node reliability of 0.9). The effectiveness of the DHT-based approach is highly sensitive to the reliability of storage nodes, with an average hit ratio increase of over 40% as the reliability of storage nodes increases from 0.5 to 0.9. When the reliability of storage nodes is 0.9, the DHT-based approach can yield a slightly higher hit ratio compared to employing a management node with the same reliability. However, MEAN consistently achieves the highest hit ratio when the management node is reliable. The average hit ratio can reach up to 93.44% when storage nodes have a reliability of 0.9. This suggests that implementing a highly reliable management node is an effective strategy for enhancing the performance of MEAN.

VI. Conclusion

In this paper, we present MEAN, a deduplication-enabled storage system using unreliable resources at the network edge. MEAN improves the file hit ratio by jointly considering space efficiency and file reliability. Thus, it can effectively reduce file retrieval delays under unreliable environments and alleviate the congestion of the backbone network. We provide efficient heuristics based on similarity-aware hierarchical clustering and elaborate our MEAN strategy with three different reliability scenarios. The comprehensive experimental results based on the prototype and the real-world dataset demonstrate the superiority of MEAN in the file hit ratio, average retrieval throughput, and average retrieval delay.

References

[1] A. Nicolaescu, O. Ascigil, and I. Psaras, "Edge data repositories - The design of a store-process-send system at the edge," in *Proc. ACM CoNEXT Workshop Emerg. Netw. Comput. Paradigms*, 2019, pp. 41–47.

[2] 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.

[3] Y. Liu, Y. Mao, X. Shang, Z. Liuy, and Y. Yang, "Distributed cooperative caching in unreliable edge environments," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1049–1058.

[4] X. Cao, J. Zhang, and H. V. Poor, "An optimal auction mechanism for mobile edge caching," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 388–399.

[5] A. Nicolaescu, S. Mastorakis, and I. Psaras, "Store edge networked data (SEND): A data and performance driven edge storage framework," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.

[6] X. Wei and Y. Wang, "Popularity-based data placement with load balancing in edge computing," *IEEE Trans. Cloud Comput.*, vol. 11, no. 1, pp. 397–411, First Quarter 2023.

[7] F. Dahlqvist, M. Patel, A. Rajko, and J. Shulman, "Growing opportunities in the Internet of Things," Jul. 22, 2019. [Online]. Available: https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things

[8] R. van der Meulen, "What edge computing means for infrastructure and operations leaders," Oct. 3, 2018. [Online]. Available: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders

[9] J. Liu, M. L. Curry, C. Maltzahn, and P. Kufeldt, "Scale-out edge storage systems with embedded storage nodes to get better availability and cost-efficiency at the same time," in *Proc. USENIX Workshop Hot Topics Edge Comput.*, 2020.

[10] L. Pu, L. Jiao, X. Chen, L. Wang, Q. Xie, and J. Xu, "Online resource allocation, content placement and request routing for cost-efficient edge caching in cloud radio access networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 8, pp. 1751–1767, Aug. 2018.

[11] C. Li, J. Bai, Y. Chen, and Y. Luo, "Resource and replica management strategy for optimizing financial cost and user experience in edge cloud computing system," *Inf. Sci.*, vol. 516, pp. 33–55, 2020.

[12] P. Hamandawana, A. Khan, J. Kim, and T. Chung, "Accelerating ML/DL applications with hierarchical caching on deduplication storage clusters," *IEEE Trans. Big Data*, vol. 8, no. 6, pp. 1622–1636, Dec. 2022.

[13] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," in *Proc. USENIX Conf. File Storage Technol.*, 2011, pp. 15–29.

[14] Y. Zhang, Y. Wu, and G. Yang, "Droplet: A distributed solution of data deduplication," in *Proc. IEEE/ACM Int. Conf. Grid Comput.*, 2012, pp. 114–121.

[15] B. Mao, H. Jiang, S. Wu, and L. Tian, “POD: Performance oriented I/O deduplication for primary storage systems in the cloud,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2014, pp. 767–776.
[16] B. Balasubramanian, T. Lan, and M. Chiang, “SAP: Similarity-aware partitioning for efficient cloud storage,” in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 592–600.
[17] W. Xia et al., “FastCDC: A fast and efficient content-defined chunking approach for data deduplication,” in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 101–114.
[18] G. Cheng, D. Guo, L. Luo, J. Xia, and Y. Sun, “Jingwei: An efficient and adaptable data migration strategy for deduplicated storage systems,” in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1659–1668.
[19] S. Li, T. Lan, B. Balasubramanian, M. Ra, H. W. Lee, and R. K. Panta, “EF-dedup: Enabling collaborative data deduplication at the network edge,” in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 986–996.
[20] H. Yan, X. Li, Y. Wang, and C. Jia, “Centralized duplicate removal video storage system with privacy preservation in IoT,” *Sensors*, vol. 18, no. 6, p. 1814, 2018.
[21] Data deduplication overview, Feb. 19, 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/storage/data-deduplication/overview
[22] HDFS Architecture, May 18, 2022. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[23] 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. Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[24] S. Ghemawat, H. Gobioff, and S. Leung, “The Google file system,” in *Proc. ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
[25] M. Oh et al., “Design of global data deduplication for a scale-out distributed storage system,” in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1063–1073.
[26] S. Quinlan and S. Dorward, “Venti: A new approach to archival storage,” in *Proc. USENIX Conf. File Storage Technol.*, 2002, pp. 89–101.
[27] 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.
[28] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, “Primary data deduplication - Large scale study and system design,” in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 285–296.
[29] D. T. Meyer and W. J. Bolosky, “A study of practical deduplication,” in *Proc. USENIX Conf. File Storage Technol.*, 2011, pp. 1–13.
[30] P. Shilane, M. Huang, G. Wallace, and W. Hsu, “WAN optimized replication of backup datasets using stream-informed delta compression,” in *Proc. 10th USENIX Conf. File Storage Technol. (FAST 12)*, San Jose, CA, Feb. 2012. [Online]. Available: https://www.usenix.org/conference/fast12/wan-optimized-replication-backup-datasets-using-stream-informed-delta-compression
[31] G. Wallace et al., “Characteristics of backup workloads in production systems,” in *Proc. USENIX Conf. File Storage Technol.*, 2012, Art. no. 4.
[32] R. Luo, H. Jin, Q. He, S. Wu, Z. Zeng, and X. Xia, “Graph-based data deduplication in mobile edge computing environment,” in *Proc. 19th Int. Conf. Service-Oriented Comput.*, Springer, 2021, pp. 499–515.
[33] G. Cheng, D. Guo, L. Luo, J. Xia, and S. Gu, “LOFS: A lightweight online file storage strategy for effective data deduplication at network edge,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2263–2276, Oct. 2022.
[34] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, “Exploring layered container structure for cost efficient microservice deployment,” in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–9.
[35] GitHub Topics, 2023. [Online]. Available: https://github.com/topics
[36] Z. Cheng et al., “ERMS: An elastic replication management system for HDFS,” in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2012, pp. 32–40.
[37] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, “Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads,” in *Proc. USENIX Conf. File Storage Technol.*, 2012, Art. no. 20.
[38] 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. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2018, pp. 87–93.
[39] C. Hamdeni, T. Hamrouni, and F. B. Charrada, “Data popularity measurements in distributed systems: Survey and design directions,” *J. Netw. Comput. Appl.*, vol. 72, pp. 150–161, 2016.
[40] P. C. Chu and J. E. Beasley, “A genetic algorithm for the multidimensional knapsack problem,” *J. Heuristics*, vol. 4, no. 1, pp. 63–86, 1998.
[41] R. Kisous, A. Kolikant, A. Duggal, S. Sheinvald, and G. Yadgar, “The what, the from, and the to: The migration games in deduplicated systems,” in *Proc. USENIX Conf. File Storage Technol.*, 2022, pp. 265–280.
[42] O. A. Abdul-Rahman and K. Aida, “Google users as sequences: A robust hierarchical cluster analysis study,” *IEEE Trans. Cloud Comput.*, vol. 8, no. 1, pp. 167–179, First Quarter 2020.
[43] R. L. Rivest, “The MD5 message-digest algorithm,” *RFC*, vol. 1321, pp. 1–21, 1992.
[44] D. E. E. III and P. E. Jones, “US secure hash Algorithm 1 (SHA1),” *RFC*, vol. 3174, pp. 1–22, 2001.
[45] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, “Optimizing bloom filter: Challenges, solutions, and comparisons,” *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1912–1949, Second Quarter 2019.
[46] L. Luo et al., “Efficient multiset synchronization,” *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1190–1205, Apr. 2017.
[47] Elastic Compute Service (ECS), 2023. [Online]. Available: https://www.aliyun.com/
[48] J. Li et al., “Popularity-driven coordinated caching in named data networking,” in *Proc. Symp. Architecture Netw. Commun. Syst.*, 2012, pp. 15–26.
[49] D. Guo, J. Xie, X. Shi, H. Cai, C. Qian, and H. Chen, “HDS: A fast hybrid data location service for hierarchical mobile edge computing,” *IEEE/ACM Trans. Netw.*, vol. 29, no. 3, pp. 1308–1320, Jun. 2021.

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

Geyao Cheng received the BS and MS degrees in management science and engineering from the National University of Defense Technology, Changsha, China, in 2017 and 2019, respectively, where she is currently working toward the PhD degree with the College of Systems Engineering. Her research interests include edge computing and distributed system.

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. He was also a visiting research scholar with the National University of Singapore, Singapore, in 2018. His research interests include probabilistic data structures and data analysis.

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 member of the ACM.

Pin Lv (Member, IEEE) received the BS degree in software engineering from Northeastern University, Shenyang, in 2006, and the PhD degree in computer science and technology from the National University of Defense Technology, Changsha, in 2012. He is currently with the School of Computer Electronics and Information, Guangxi University, Nanning, and also with Guangxi Key Laboratory of Multimedia Communications and Network Technology, Nanning. His research interests include wireless networks, mobile computing, Internet of Things, etc. He is a member of CCF and ACM.

Bowen Sun received the double bachelor's degree in petroleum engineering and computer science from the China University of Petroleum, Beijing, in 2020. He is currently working towards the MS degree with the College of Computer, NUDT, Changsha. His main research interests include network measurement and in-network computing.