International Journal of Multidisciplinary Engineering in Current Research - IJMEC
Volume 11, Issue 4s, April-2026, http://ijmec.com/, ISSN: 2456-4265

Full Length Article

Ripple: A Decentralized Edge Based Data Deduplication Frame Work

Mohd Abraar[1], Mohd Rafay[2], Syed Mahmood Hussaini [3], Dr. Md Zainlabuddin[4]

[1,2,3]BE.Students; Dept of CSE, ISL engineering College (affiliated to Osmania University)Hyderabad India

[4]Associate professor; Dept of CSE, ISL engineering College (affiliated to Osmania University) Hyderabad India

Mail Id;Mabraar799@gmail.com,160522733103@islec.edu.in,buqari666@gmail.com

Accepted 24-04-2026

Abstract

With its advantages in ensuring low data retrieval latency and reducing backhaul network traffic, edge computing is becoming a backbone solution for many latency-sensitive appli cations. An increasingly large number of data is being generated at the edge, stretching the limited capacity of edge storage sys tems. Improving resource utilization for edge storage systems has become a significant challenge in recent years. Existing solutions attempt to achieve this goal through data placement optimization, data partitioning, data sharing, etc. These approaches overlook the data redundancy in edge storage systems, which produces substantial storage resource wastage. This motivates the need for an approach for data deduplication at the edge. However, existing data deduplication methods rely on centralized control, which is not always feasible in practical edge computing environments. This article presents Ripple, the first approach that enables edge servers to deduplicate their data in a decentralized manner. At its core, it builds a data index for each edge server, enabling them to deduplicate data without central control. With Ripple, edge servers can 1) identify data duplicates; 2) remove redundant data without violating data retrieval latency constraints; and 3) ensure data availability after deduplication. The results of trace driven experiments conducted in a testbed system demonstrate the use fulness of Ripple in practice. Compared with the state-of-the-art approach,Ripple improves the deduplication ratio by upto 16.79% and reduces data retrieval latency by an average of 60.42%. ,

Index Terms : Edge computing, data redundancy, data retrieval latency, data deduplication, data index, cloud deduplication

I. INTRODUCTION

The rapid growth of the Web of Things (WoT) and Artificial Intelligence (AI) has led to a huge increase in the number of smart and mobile devices connected to the internet. These devices continuously generate large amounts of data at the network edge. According to IDC, the total global data volume is expected to reach around 180 zettabytes (ZB) by the year 2025. As data generation continues to rise, traditional cloud computing alone is no longer sufficient to handle real-time processing requirements efficiently.

To solve this issue, Edge Storage Systems (ESS) have been introduced. An ESS consists of multiple interconnected edge servers located within a specific geographical area. Unlike centralized cloud servers that may be far from users, edge servers are placed closer to users, often near 5G or 6G base stations. This allows faster data delivery with lower latency, improving user experience for applications such as video streaming, gaming, IoT services, and mobile computing.

However, edge servers have limited physical space and fewer resources compared to large cloud data centers. Their storage capacity is especially limited, making efficient storage management an important challenge. Researchers have proposed several solutions to improve storage utilization, including optimized data placement, erasure coding techniques to reduce storage overhead, and dynamic data transfer between edge servers when required.

Despite these efforts, one important issue has not been fully explored: edge data redundancy. Because 5G networks use millimeter-wave (mmWave) communication technology, base stations provide very high bandwidth and support many simultaneous connections. However, their coverage area is relatively small. To avoid coverage gaps, 5G base stations and edge servers are deployed very densely, sometimes reaching up to 50 base stations per square kilometer.

In crowded locations such as universities, shopping centers, airports, or business districts, users often request similar types of content. To meet these demands quickly, application providers cache frequently accessed data—such as viral videos, trending social media content, software updates, and popular applications—on multiple nearby edge servers. While this improves response time, it also creates significant duplication of data across servers.

For example, researchers analyzed 16,342 popular TikTok videos worldwide in September 2023. Their study showed that user attention is highly concentrated

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

on a small number of very popular videos. Specifically, only 8.53% of the most popular videos accounted for 98.6% of the total likes. Since these highly popular videos are cached repeatedly on many edge servers, a large amount of storage space is wasted due to redundant copies of the same data.

Therefore, reducing edge data redundancy is becoming a critical challenge for future edge computing systems. Efficient redundancy management can help save storage resources, improve system performance, reduce operational costs, and support scalable data delivery in next-generation mobile and smart network environments.

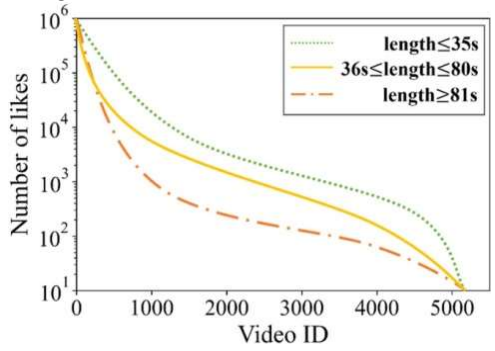


Fig.1.VideolikesofTikTok.Wecollectedinformationabout popularTikTokvideos over 15 days in September 2023 in four different geographic locationsworldwide

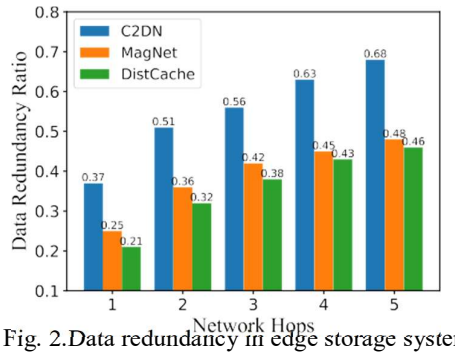


Fig. 2.Data redundancy in edge storage system maintained by differentpopularity-based caching algorithms: C2DN [14], MagNet [7], and Dist-Cache [15]. In this figure, the number of hops confines the scope of ourinspection

Challenges : There is also significant data redundancy in cloud storages ystems and many deduplication approaches have been proposed in the past decade[16],[17],[18]. However, suffering the following three limitations, these approaches are not suitable for edge storage systems.

·***Limitation-1 : Deduplication Granularity***:

The main idea Shared by many of these systems is to identify and reduce common data blocks shared by different storage nodes in the system. This idea may also be implemented in ESSs. However, the downside of thi side a is that data blocks have to be fetched from different storage nodes (a fewhundred or thousands of data blocks) for data reconstruction in response to users'data retrieval requests[19]. It takes time and doesnot align with users' requirements for lowlatency in MEC environments.

·***Limitation-2 : Deduplication Scope***

In pursuit of a high Reduplication ratio, cloud deduplication approaches try to minimize the number of replicas of any data blocks in the system to only one [20]. It is feasible in cloud storage systems because data blocks can be fetched from any storage nodes in the system for data reconstruction. However, in an edge storage system, if an edge server has to fetch from other edge servers far away in the system for data construction, the delays introduced in users' data retrieval are undesirable.

·**Limitation-3: Deduplication Control: Data deduplication**

techniques for cloud storage systems are designed based on central control and global storage information. Implementing these techniques for edge storage systems is impractical for two main reasons. First, central control and global storage information provided from the cloud introduce an inevitable delay in data deduplication at the edge [2]. This also goes against the low-latency requirement in MEC environments. Second, even if the delay is acceptable,maintaining the global storage information in the cloud demands constant updates from edge servers. This is inconsistent with MEC'so there goal ,i.e.,to reduce traffic over the backhaul network [21].

To over come these limitations, this paper presents Ripple,a novel distributed latency-aware approach for edge data dedupli-cation.It builds a local index for each edge server,enabling them to probe redundancy information,i.e.,whether or not its detais
also stored on nearby edge servers. Ripple offers two unique features for edge data deduplication:

1.·Edge servers can remove duplicate data from their storage to reduce data redundancy in the system without central control and global storage information.

2.·Edge servers can remove duplicate data based on app vendors' latency requirements without introducing subsantial delays in users' data retrieval.

We implement Ripple on a edge storage system comprised of 30storage nodes,and conduct a series of trace-driven experi-ments to evaluate its performance. The results demonstrate that it achieves a high deduplication ratio while ensuring low data retrieval latency, striking a significantly better tradeoff between them than state-of-the-art approaches.

II. BACKGROUND AND MOTIVATION

Popularity plays a crucial role in commercial recommendation systems [22]. This phenomenon can lead to substantial data redundancy within edge storage

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

systems, as the same popular content is repeatedly downloaded and cached across multiple users. For instance, when a specific product is prominently recommended, numerous users may access it simultaneously, thereby necessitating the storage of multiple identical data copies caching systems also lead to data redundancy.We implemented three recent caching algorithms designed for different dis-tributed caching systems, ran them on an edge storage system for three days based on the popularity of the TikTok videos we collected (Section I), and inspected the data redundancy in the system. Fig.2 demonstrates the results.We can see that there is a high level of data redundancy in the system. We can also see that the redundancy level increases as we expand the scope of inspection.

Data deduplication is an effective redundancy reduction technique and has been widely used in cloud storage systems in the past decade [17], [23]. It aims to maximize the deduplication ratio by removing al lthe duplicate data blocks in the system, as shown in Fig.3.There is a large body of work on *cloud data*

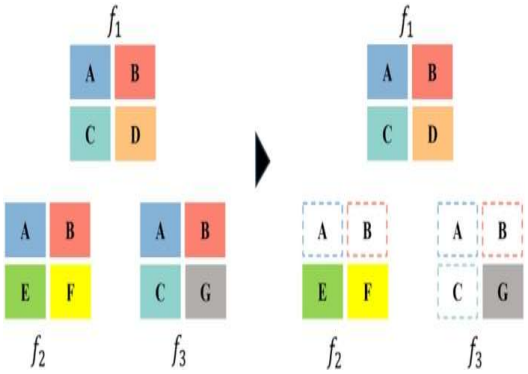
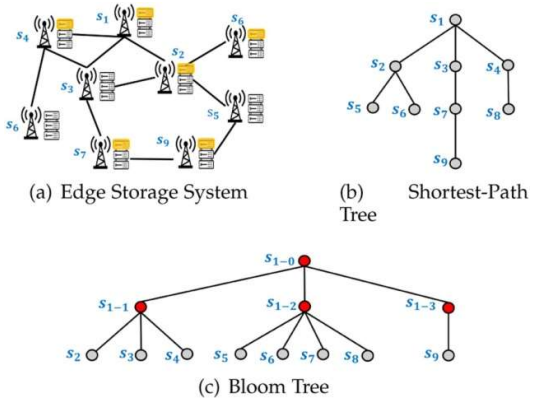


Fig.3.Cloud data deduplication. It aims to maximize the deduplication ratio by removing al lthe duplicate data blocks in the system, as



shown in Fig.3.There is a large body of work on cloud data

THEORETICAL PROBABILITY OF UPDATES ON GLOBAL STORAGE INFORMATIONIN AN EDGE STORAGE SYSTEM

p	System Size				
	2	4	8	16	32
0.1	0.19	0.34	0.57	0.81	0.97
0.2	0.36	0.59	0.83	0.97	1.00
0.4	0.64	0.87	0.98	1.00	1.00
0.8	0.96	1.00	1.00	1.00	1.00

Deduplication (CDD) [16], [24]. In general, they go through three main steps: 1) The files in the system are partitioned into data blocks. A hash value is calculated for each of the data blocks as its fingerprint with hash functions such as SHA-1 or SHA-256. 2) A central metadata server in the system builds a global fingerprint index for identifying duplicate data blocks across all the data in the system. 3) Duplicate data blocks are removed until there is only one copy for each data block in the system. This CDD process is feasible in cloud data centers because the central metadata server has access to the global information about data storage across all the nodes in the system. However, the implementation of this process in an edge storage system is ineffective, if not infeasible. The main reasons are threefold: Strategy Timeliness: Unlike the nodes in a cloud storage system, the edge servers in an edge storage system are geographically distributed. None of them have access to the global data storage information. A possible solution is to maintain the global data storage information with a central metadata server in the remote cloud. It formulates globally-optimal data deduplication strategies for individual edge servers and sends them to the corresponding edge servers for implementation. This takes time. Take BEDD-O [25], the state-of-the-art approach for edge data deduplication (EDD) for example. When the system size (measured by the number of edge servers in the system) reaches 30, BEDD-O takes about 18 seconds to formulate an EDD strategy. However, during this period of time, the data storage in the edge storage system may have changed, which will render the EDD strategy obsolete - the data demands at the edge vary greatly at different locations [26] over time [27]. Information Up-to Dateness: To pursue a maximum deduplication ratio, the meta data server must ensure that its global data storage information is always up-to-date. To achieve this objective, the edge servers in the system must upload data update information to the metadata server so that it can update its global information accordingly. This incurs massive traffic over the backhaul network, which contradicts the objective of MEC, i.e., to minimize backhaul network traffic [21]. Take an edge storage system comprised ofnedge servers for example. Let us suppose that the probability of data changes occurring on an individual edge

Data retrieval latency versus deduplication ratio. In the experiment, we randomly removed duplicate data in the system until the target deduplication ratio was reached. server in a unit of time is p. The

Fig. 4.Data retrieval latency versus deduplication ratio. In the experiment,we randomly removed duplicate data

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

in the system until the target deduplicationratio was reached.

probability of a data update in the system during a unit of time is 1 − (1 − p)n. It is also the probability that the metadata server has to update its global storage information during each unit of time. Table I provides more information about such probabilities in edge storage systems of different sizes. It is evident that when the system size or p increases, the probability of global updates surges. For example, when the system size is only 8 and p is 0.4, the probability of global updates is close to 100%. Apparently, it is communication-expensive to maintain the up-to-dateness of the global storage information on the metadata server. Deduplication Ratio: Most, if not all, CDD approaches aim to maximize the deduplication ratio [16], [18], [23]. However, in an edge storage system, there is a balance between deduplication ratio and data retrieval latency. When the replicas of a data item d are removed from nearby edge servers, an edge server has to retrieve d from distant edge servers, resulting in high latency. To validate this, we conducted an experiment on an edge storage system comprised of 30 edge servers, where data replicas in the system are removed to pursue different deduplication ratios. Fig. 4 summarizes the results. We can see that an increase in the deduplication ratio from 25% to 50% will result in a surge in the average data retrieval latency from 42.5 milliseconds to 70.1 milliseconds. This tells us that excessive deduplication conflicts with the demand for low latency in MEC environments. Our Motivation: The above analysis tells us that it is impractical to perform globally-optimal data deduplication for edge storage systems from the cloud. Another solution is for edge
servers to perform data deduplication individually at the edge instead of from the cloud. Briefly speaking, an edge server can delete its own duplicate data if there are replicas on nearby edge servers (referred to as neighbors hereafter for ease of explanation). When it needs one of those replicas to serve its users, it can retrieve a replica from its neighbors via the links between them [7], [8], [10]. Design Goals: To implement the above idea in practice, there are two main design goals: 1) To determine which duplicate data can be deleted, an edge server si needs to find out whether any of its neighbors have replicas that si can retrieve on demand. 2) In MEC environments, latency constraints differ from one application to another. An edge server si needs to be able to delete duplicate data without violating these latency constraints. For example, if an application demands that the data retrieval latency must not exceed 2 hops,1 si must not delete its data if there are no replicas within 2 hops in the system. 3) Data deduplication must not compromise data availability. More specifically, if an edge server sj could retrieve a data item d from within the system, it must still be able to retrieve it from within the system after data deduplication. This is referred to as the principle of data availability. Our Solution: Our solution is to build a local index for each edge server in the system.

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

An edge server si's index maintains the information about data storage on si's neighbors. With this index, si's can look up a data item and find out whether it is stored on any of its neighbors. In this way, si's can delete redundant data, i.e., those that can be retrieved from neighbors, to achieve deduplication. This tackles the first challenge in Section I. To tackle the second challenge, the index needs to facilitate latencybased deduplication. An edge server si's must be able to use the index to find out whether there are any replicas of a specific data item d within a certain number of hops in the system.

III. INDEX DESIGN

Ripple employs a tree structure to index the data stored on an edge server's neighbors. By inspecting the tree index, an edge server can quickly find out whether a data item d is stored within h hops and where it is stored. A. Primary Solution As discussed at the end of Section II, when an edge server si needs to free up its storage resources, it can remove data from its own storage that is also stored on its neighbors. In the meantime, it needs to be able to retrieve the data from its neighbors later on without violating latency requirements. To achieve these goals, a straightforward solution is to build a shortest-path tree for organizing its neighbors. In this shortest-path tree, the paths from the root node to other tree nodes represent the shortest paths between si and the corresponding edge servers in the 1.

For easy explanation, the latency between two edge servers in this paper is measured by the number of hops between them [2], [10]. In practice as well as in our experiments (Section V), the edges between edge servers are annotated with weights that represent the actual latency between them

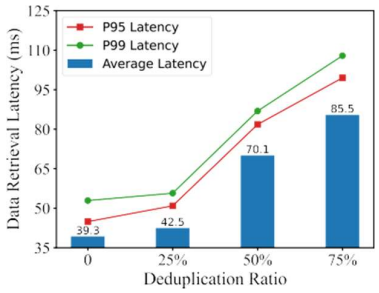


Fig.5.BuildingaBloomtree.Itsharesthesamestructure,e.g., size depth,with the Ripple tree

edge storage system. Fig. 5(b) illustrates the shortest-path tree constructed from the edge storage system in Fig. 5(a). Each leaf node in the shortest-path tree corresponds to one of si's neighbors and indexes the data stored on it. Bloom filter and its variants are widely used in distributed systems to find out whether a data item is in a dataset [28]. Its low storage overheads and high query efficiency align with the characteristics of MEC environments, i.e., constrained resources on edge servers and mandatory low service latency. It supports data

insertions and queries, both operating at O(k), where k is the number of hash functions used. However, due to the binary nature of the bits in the Bloom filter (either 0 or 1), it does not support deletion operations. Fortunately, the counting Bloom filter (CBF) does and offers O(k) deletion operations. Using CBFs as nodes in the shortest-path tree, edge server si can build a shortest-path Bloom filter tree (referred to as Bloom tree hereafter for simplicity). Fig. 5(c) demonstrates the Bloom tree generated from the shortest-path tree in Fig. 5(b). In this Bloom tree, each leaf node indexes the data stored on one of si's neighbors and a non-leaf node is also a counting Bloom tree generated through bitwise addition of its child nodes. It supports three types of queries: By inspecting the root node, si can find out whether a data item d is stored on its neighbors within h hops (h = 3). By inspecting the nodes at the next level, e.g., s1−1, s1−2, and s1−3, si can find out whether d is stored within a certain number of hops. For example, by inspecting s1−2, it can find out whether d is stored within 2 hops, i.e., on s5, s6, s7, and s8. By inspecting the leaf nodes, si can locate d and retrieve d from the corresponding edge server in the system. Index Initiation and Maintenance: An edge server can collect the CBFs (instead of complete data storage information) from its neighbors to initialize its index tree. When there is a data update on an edge server sj in the system, e.g., a data item inserted, deleted, or modified, sj can broadcast the data update information in the system, and its neighbors can update their tree indexes to ensure its up-to-dateness. This may result in frequent information exchange between edge servers.

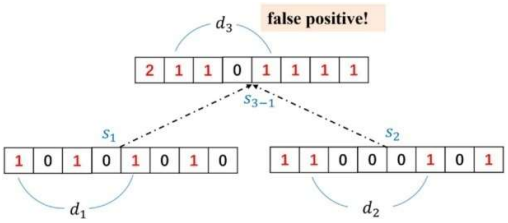


Fig.6.False positive in counting Bloom filter tree.

Fortunately, the traffic resulting from data change updates is stringently regulated by the parameter h; specifically, edge servers engage in communication solely with their h-hop neighbors, ensuring that the overall network traffic remains insignificant .This will be discussed in Section V-D.

Challenges in Practice: By design, the Bloom tree achieves the two design goals discussed in Section II. However, when we implement it in practice, two challenges present: Uniform Node Size: In the Bloom tree, all the nodes must be of the same size to facilitate the bitwise aggregation of child nodes for generating the parent node. They share the same capacity, measured by the number of data items that can be indexed without violating the target accuracy. Briefly speaking, a larger-sized CBF can index more data items without many false positives [29]. In the Bloom tree, the nodes at an upper-level index more data than those at a lower level. For example, the root node indexes the data stored on all si's neighbors while the leaf nodes index the data stored on their corresponding edge servers. Thus, the nodes at the upper level are prone to a lower accuracy caused by a higher false positive rate. Fig. 6 showcases an instance where the aggregation of two CBFs results in a false positive. Apparently, a node size that suffices for lower-level nodes does not ensure the same accuracy for upper-level nodes. In addition, a node size that suits upper-level nodes may cause significant memory wastage on lower-level nodes that are not desired by resourceconstrained edge servers. It is a challenge to strike a proper tradeoff in between Fixed Node Sizes: The size of a CBF cannot be modified at runtime [29], making it unsuitable for dynamic MEC environments. As discussed in Section II, memory wastage. If a large
size was set for these Bloom filters to ensure their accuracy, as discussed above, the memory wastage incurred would be even more profound.

Ripple Tree:

To tackle the above challenges, Ripple employs a new index structure named the Ripple tree.

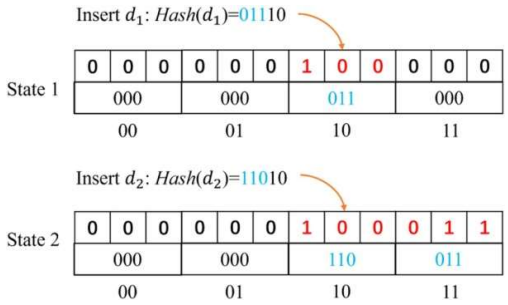


Fig.7.Example of data insertion in quotient filter.

Instead of counting Bloom a Ripple tree is comprised of quotient filters [30]. The nodes in a Ripple tree can be sized and adjusted flexibly. In this section, we first introduce the quotient filter briefly. Then, we discuss its aggregation and expansion. Quotient Filter: The quotient filter offers the same data query functionality as the Bloom filter and counting Bloom filter. It is inherently a hash table, and consists of multiple components referred to as slots. Each slot stores a N-bit entry of a data item computed with the Robin Hood hashing mechanism [31]. Ripple employs the last M bits of the N-bit slot as its address, and the X bits of the remaining bits as the fingerprint of the data item. As an example, Fig. 7 presents a 12-bit quotient filter comprised of four 3-bit slots and the insertion of three data items. These items are first hashed to 5-bit hashes. The last two bits of a hash represent the address of the corresponding data item in the quotient filter. The other three bits represent its fingerprint. Take data item d1 for example. It is first hashed to 01110. The last two bits, i.e., 10, is the address

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

of d1 in the filter, and 011 is its fingerprint. Similar to Bloom filters and counting Bloom filters, collisions may occur when data items are inserted into a quotient filter. Take d2 in Fig. 7 for example. It is hashed to 11010. Its address in the quotient filter is 10, which is the same as d1. The insertion of d2 into the filter will collide with d1. To avoid the collision, the quotient filter moves d1 to the next slot addressed 11 and saves 110 (the fingerprint of d2) in the 10 slot. Next, the quotient filter marks d1 as "is_continuation" to indicate that d1 shares the initial slot address as its left neighbor in the filter. In addition to "is_continuation", there are two other labels, i.e., "is_occupied" and "is_shifted". The former indicates whether the slot is an initial slot for one data in the filter and the latter indicates that the data item was moved. The quotient filter employs a 3-bit indicator to mark each data item in the filter, in the order of "is_occupied", "is_continuation", and "is_shifted". In particular, a data item marked "is_continuation" belongs to the same "run" as its left data item in the filter. All the data items in the same run share the initial slot addressed. Take Fig. 7 for example. After d1 and d2 are inserted, d1 is marked 100 and d2 is marked 011. They belong to the same run. With the ability to move data items, multiple quotient filters can be aggregated to produce a bigger quotient filter that indexes all the data items in the original quotient filters. Fig. 8 illustrates the aggregation of two quotient filters qf1 and qf2 into a bigger

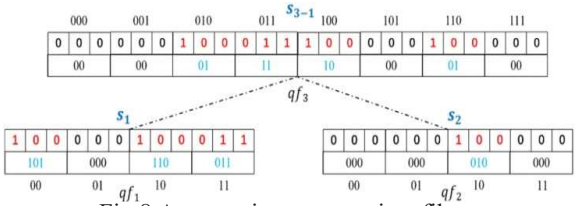


Fig.8.Aggregating two quotient filters.

quotient filter qf3. The aggregation process inserts the data items from qf1 and qf2 into qf3, following the same process for data insertion described above. Leveraging this feature of the quotient filter, Ripple can replace the counting Bloom filters in its Bloom tree with quotient filters, and aggregate child nodes, following the filter aggregating process below, to produce flexibly-sized quotient filters as non-leaf nodes. The outcome is a Ripple tree. It overcomes the challenge of fixed node sizes (Section III-A). Data Query: A data query for a data item d in a quotient filter starts from the slot matching the address bits of d. 1) If 'is_occupied = 0', there is no run starting from this slot, return false as the query result; otherwise, initiate a counter and go to the next step. 2) Go left and increase the counter by 1 every time when a slot with 'is_occupied = 1', until the first cluster comprised of multiple consecutive runs is found. 3) Begin a scan rightward and decrease the counter by 1 upon encountering a slot with the 'is_continuation = 0' until the counter reaches 0. The slot where the counter becomes zero is the one hosting d. 4) Compare the fingerprint of d with those within the entire run, starting from the identified slot. This comparison will determine if the data is present in the dataset. Data Deletion: The deletion of data d from a quotient filter goes through five steps below: 1) Remove the fingerprint from the initial slot in the filter that corresponds to the hash of d. 2) If this slot is the start of a run, move through the run until you find a matching fingerprint. 3) Remove the first fingerprint encountered during the deletion operation; 4) Shift all the subsequent fingerprints in the cluster to the left by one slot; 5) Adjust the corresponding flags for each of these shifted fingerprints. Filter Aggregation: When creating a non-leaf node in the Ripple tree, its initial size must not be arbitrarily determined. It needs to be sufficiently large to accommodate inserted data properly without compromising its accuracy. In the meantime, it must not be overly large because it would cause significant memory wastage. When aggregating multiple child nodes into a parent node, an intuitive solution is to make it just large enough to accommodate the data from the child nodes. Let us try to aggregate two quotient filters with 100 data items each for example. They are sized 16 bits to ensure a false positive rate lower than 0.001. If the parent quotient filter is given 17 bits, based on the calculation given in [32], its false positive rate will be 0.003, 2× higher than its child nodes' false positive rates. Thus, the non-leaf quotient filters in the Ripple tree must be properly sized to ensure their accuracy, including their M and X values.

IV. EQUATIONS AND SUBSTITUTION

A proper value of M must ensure that the capacity of the parent quotient filter hosted by an edge server si, i.e., the maximum amount of data it can store, is greater than or equal to the sum of the capacities of the child nodes:

$$2M \geq j \in N(si)\ 2Mj \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

where N(si) is the number of the child nodes, i.e., the number of si's neighbors. The value of X must ensure that the false positive rate of the parent node does not exceed the minimum false positive rate of the child nodes:

$$2-X\alpha \leq \min\{2-Xj\alpha \mid j \in N(si)\} \ldots\ldots\ldots\ldots\ldots(2)$$

By solving (1) and (2), we can obtain the optimal values of M and X for the parent node that satisfy (3) and (4), espectively

$$M \geq \max\{M_j, j \in N(si)\} + \log_2|N(si)|\ldots\ldots\ldots\ldots\ldots(3)$$
$$X \geq \max\{X_j, j \in N(si)\}\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(4)$$

Filter Extension: As discussed in Section II, users' data demands vary greatly over time at the edge. Upon an unexpected data demand increase in an area, the Ripple trees maintained by the edge servers in the area may suffer from decreased accuracy. To tackle this challenge,

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

Ripple extends the quotient filters in a Ripple tree on demand to ensure the accuracy of these filters and the Ripple tree. As introduced before, the data2 in a quotient filter that conflict with the inserted data fingerprint will be moved to the right in the filter. This fills up the filter as more data are inserted. According to an in-depth analysis of the quotient filter [33], its accuracy (measured by false positive rate) would degrade significantly when its occupancy rate exceeds 75%. When an edge server si uses a Ripple tree comprised of overfilled quotient filters to identify and delete duplicate data, a high false positive rate will result in incorrect deletion of data that are in fact not redundant. Ripple tackles this challenge by extending the quotient filters in a Ripple tree when its occupancy rate exceeds 75%. The key idea is to extend the address bits of the slots in the filter with the lower bit(s) taken from the fingerprint bits. Fig. 9 illustrates the extension process. We take the last bit from the fingerprint bits and concatenate it to the left of the address bits. This doubles the capacity of the filter, increasing the number of available addresses from 4 (22) to 8 (23). Then, Ripple transfers the data from the original filter to the new filter based on their new addresses. Take d1 for example. Its fingerprint 101 becomes 10 and its address 00 becomes 100. Thus, it is transferred to the slot addressed 100in the new quotient filter. Data that belong to a run, i.e., those that were moved when inserted into the original filter, are transferred to the new filter based on its new address produced based on its initial address in the old filter. Take d3 in Fig. 9 for example. Its current address in the original filter is 11 and its initial address is 10 (Fig. 7). It is transferred to the slot in the filter addressed 110 (1 ⊕ 10) rather than 111 (1 ⊕ 11).

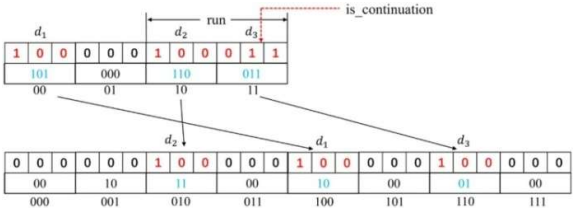


Fig.9.Extending a4-slot quotient filtert oa n8-slot one.

Remark: Fig. 9 demonstrated a 2× extension, which took 1 bit from the fingerprint bits. When a more significant extension is needed, Ripple can take more than 1 bit from the fingerprint bits to extend a quotient filter. However, there is a limit on how much it can be extended. In the example presented in Fig. 9, there are three bits in total in the fingerprint bits. A maximum of two bits can be taken to extend the address bits to quadruple the size of the filter. As discussed before, the size of the fingerprints in a quotient filter is X and its optimal value can be obtained with (2). To offer high extension flexibility, Ripple can employ a larger X.

Filter Contraction: In this study, the expansion operation will be carried out when the occupancy ratio of the filter reaches or exceeds θe = 75%. Accordingly, it employs the threshold for filter contraction to θc = θe/4, also following [33]. When the occupancy ratio of the filter drops to θc or below, it contracts. This setting prevents filter oscillation between expansion and contraction.

A quotient filter contraction goes through each data item in the filter as follows:

- Based on the three flags provided, identify the actual slot address where the data item should have been inserted. This step is crucial because the data might have been forcibly moved from its original position.
- Use the fingerprint and the actual slot address identified in the previous step, extract the original data from the filter.
- Create a new filter with half the capacity of the original filter.
- Modify the fingerprint hash value by shifting it left by one bit.
- Set the highest bit of the new filter's address to match the lowest bit of the shifted fingerprint.
- Utilize the remaining n − 1 bits to determine the new address for the data item in the new filter.

- Insert the modified fingerprint hash value into the newly determined address of the new filter.

A. BASIC DATA DEDUPLICATION Through inspecting its Ripple tree, an edge server can find out which of its own data can also be retrieved from its neighbors. If the answer is yes, it can delete these duplicate data to save up storage resources. A. Basic Data Deduplication A simple solution is for edge servers to deduplicate their own data individually. The pseudocode of the core procedure can be found in Algorithm 1. It ensures that edge servers do not retrieve data from distant neighbors for users. Otherwise, users' quality of experience would be compromised.

The following steps to deduplicate:

1) For each data d, inspect its Ripple tree and find out whether d exists within h hops in the system.
2) Put together a list of duplicate data in its storage. 3) Send a request that, asking them whether it can delete each of these data.
4) The responses from its neighbors and delete d only when they all say yes to deleting

B. Latency-Aware Data Deduplication Following the basic data deduplication strategy presented above, edge servers can remove their own duplicate data individually. It is an easy strategy to implement. However, it has a shortcoming that is not easy to notice but may compromise the performance of the edge storage system implementing the strategy. If si is a hub node, meaning it is connected to many other edge servers in the system,

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

these nodes and their neighbors can retrieve data fromsi with low latency. Such hub nodes are critical in ensuring system performance. Following the basic strategy, the deletion of si's data may compromise system performance significantly. To tackle this shortcoming, Ripple employs a latency-aware data deduplication strategy. The pseudocode of the latencyaware data deduplication strategy can be found in Algorithm 2. Following this strategy, edge server si goes through the same steps as it would follow the basic strategy. The only difference is that si will include a number li in the message sent to si's neighbors in Step 3 above to indicate the possible impact of deleting the data of the list from its storage. It is calculated as li = h l=1 l · Ni(h), where h is the latency constraint, and Ni(h) is the number of si's neighbors within h hops. Upon the receipt of si's message, each of si's neighbors denoted as sj , will determine whether to answer yes or no to si deleting each of the data on the list, say d.

There are two cases:

Case 1: Edge server sj does not have d. It will determine whether to say yes or no, following the same steps in the basic strategy.
Case 2: Edge server sj has d. It will compare lj against li. If the former is larger, it returns yes to si. Otherwise, sj implements the basic strategy to determine whether to delete d. If it deletes d, it answers no about d to sj , and yes otherwise.

(a) Deduplication Ratio vs. h

V. EXPERIMENTAL EVALUATION.

We implement Ripple in a testbed edge storage system and evaluate its performance in deduplication ratio and service latency.

1. Experiment Setup .

(b) Latency vs. h

Trace: The experiments are conducted on a public Telecom dataset [35]. It includes over 7.2 million records from 9,481 mobile users accessing the Internet through

(b) FPR vs. h

3,233 base stations over a six-month period. Based on these traces, we cache the 1000 most popular TikTok videos collected in September 2023.

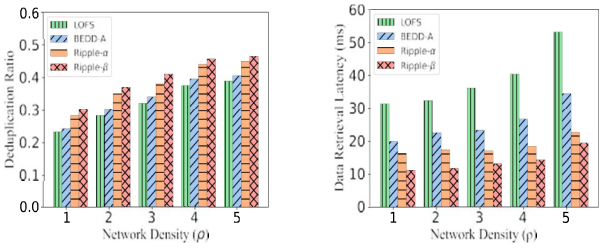


Fig.10. Performance versus network density(ρ).

In our edge storage system running MagNet [7], the state-of-the-art data caching algorithm for edge storage systems. Benchmarks: We implement two benchmarking approaches and run them to deduplicate data in the system under the same latency constraint (h) as Ripple.

In comparison with **LOFS** and **BEDD,** both Ripple versions achieve a higher deduplication ratio, securing a 1.72× improvement over LOFS and BEDD-A on average. This indicates the advantage of Ripple in minimizing data redundancy.

Compared to **Ripple-α, Ripple-β** almost halves the data retrieval latency.

2. Overall Evaluation.

The overall performance of LOFS, BEDD A, Ripple-α, and Ripple-β, measured by deduplication ratio and post deduplication data retrieval latency. We can see that:

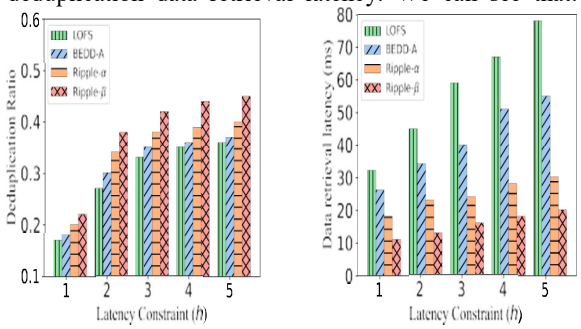


Fig.11. Performance versus latency constraint (h).

(b) False Deletion Rate vs. h

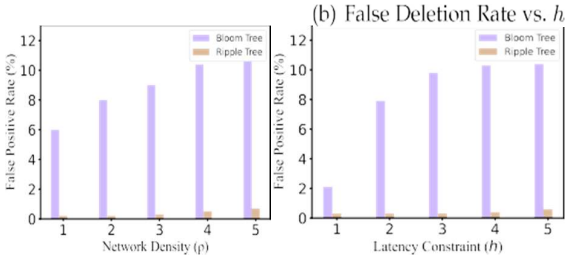


Fig.12 .False positive rate :Bloom Tree versus Ripple Tree.

Fig. 11 illustrates the results Overall, the phenomena shown in this figure are similar to Fig.10. An increase in h relaxes the latency constraint and includes more distant edge servers in individual deduplication processes, similar to an increase in network density ρ. As a result,

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

all four approaches manage to achieve a higher deduplication ratio, as shown in Fig. 11(a). Again, Ripple-α and Ripple-β manage to maintain data retrieval latency at much lower levels than LOFS and BEDD-A, as can be seen in Fig. 11(b).

3. Further Evaluation.

This section evaluates the performance of Ripple further. False Positive Rate (FPR): The false positive rate is a critical performance indicator for probabilistic indexes. The primary solution designed based on the Bloom tree suffers a high positive rate. To find out whether Ripple

(a) Average Communication Time

can properly tackle this challenge, we conduct an

(b) Communication Traffic

experiment to compare the performance of Ripple in two cases:

1) edge servers index data with Bloom trees; and
2) edge servers index data with Ripple trees.

Fig.13 demonstrates the Ripple tree's tremendous advantage over the Bloom tree in maintaining a low false positive rate.

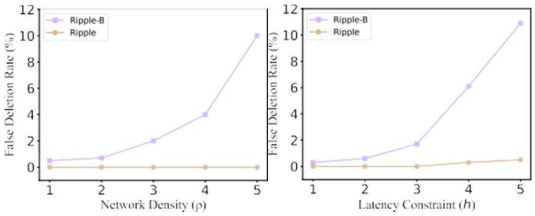
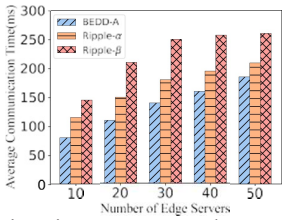


Fig.13. False deletion rate: Ripple versus Ripple-B.

In this experiment, the Bloom tree's false positive rate ranges between 6.03% and 10.62% while the Ripple tree's false positive rate always stays below 0.67%. On average, compared with the Bloom tree, the Ripple tree can reduce the false positive rate by approximately 26.17 times. When network density ρ increases, there are more leaf nodes in the Bloom tree than in the Ripple tree, indexing more data in total. Aggregated from these leaf nodes, the non-leaf nodes in these trees also have to index more data. This leads to an unavoidable rise in their false positive rates, which can be observed in Fig. 12(a). Similar phenomena can be observed in Fig. 12(b) because the latency constraint relaxed by a larger h also includes more leaf nodes in the edge servers' Bloom trees and Ripple trees. False Deletion Rate: When the Bloom tree or the Ripple tree produces a false positive, edge server si is led to believe that there are data replicas within h hops, which is actually not true. It will mistakenly delete its own replica, which compromises system performance. We measure the false deletion rates of Ripple and Ripple-B (Ripple with Bloom trees). Fig. 13 shows the results. Two main findings can be drawn. First, Ripple always beats Ripple-B. Second, when ρ or h increases, Bloom trees produce more false positives, as shown in Fig. 12. Misled by these false positives, edge servers will delete more data mistakenly, leading to compromised system performance. In fact, false data deletions may violate the principle of data availability, the third design goal discussed in Section II. Thus, the Ripple tree does an outstanding job at data deduplication in practice with a minimum false deletion rate.

Fig.15.Communication over heads. Average



communication time represents the average time taken by an edge server to make a deduplication decision.Communication traffic represents the amount of data that needs to be communicated for the system to perform deduplication operations.

4. Deduplication Overheads.

As a decentralized approach, Ripple incurs communication overheads, because information exchange is required between edge servers to facilitate data deduplication. Fig. 15 illustrates the communication overheads introduced by Ripple as the system size scales up in terms of the number of edge servers. It can be observed that with an increase in system size, Ripple's

(a) False Deletion Rate vs. ρ

communication overhead rises due to the growing number of neighboring servers each edge server needs to interact with on average. However, we observe that as the system size continues to increase, Ripple's communication overheads do not significantly escalate and instead remain nearly constant, demonstrating its excellent scalability. As the number of edge servers increases, the network topology of the edge server network gradually tends to saturate, resulting in more complex connectivity among servers. However, within the latency constraints (such as within a 2-hop range), the number of edge servers that can communicate remains relatively stable and limited. This limitation causes the overall communication latency to stabilize with the growth in the number of edge servers. Overall, the traffic generated by data change updates is not substantial.

Conclusion And Future Work.

This paper presented Ripple, the first decentralized approach for deduplicating data in edge storage systems in practice. It employs a new data structure named the Ripple tree to index the data stored on an edge server's neighbors in the system. Based on the Ripple tree, Ripple can deduplicate edge data without compromising data availability or violating data retrieval latency constraints. Its tree expansion/contraction mechanisms minimize the memory overheads effectively. Compared with the state-of-the-art approach, Ripple improves the deduplication

https://doi.org/10.63665/IJMEC.1104s.11
ISSN: 2456-4265
IJMEC 2026

ratio by up to 16.79% and reduces data retrieval latency by an average of 60.42%. In the future, in-depth communicationoptimized mechanisms will be studied for Ripple.

REFERENCES.

[1] W. Shi, G. Pallis, and Z. Xu, "Edge computing," Proc. IEEE, vol. 107, no. 8, pp. 1474–1481, Aug. 2019.

[2] AWS, "AWS Wavelength for media & entertainment," 2021. [Online]. Available: https://d1.awsstatic.com/Wavelength2020/AWS-Wavelengthfor-Media-Entertainment-SolutionBrief-Feb2021-Final.pdf

[3] G. Zhu et al., "Pushing AI to wireless network edge: An overview on integrated sensing, communication, and computation towards 6G," Sci. China Inf. Sci., vol. 66, no. 3, 2023, Art. no. 130301.

[4] Q. He, Z. Dong, F. Chen, S. Deng, W. Liang, and Y. Yang, "Pyramid: Enabling hierarchical neural networks with edge computing," in Proc. ACM Web Conf., 2022, pp. 1860–1870.

[5] H. Jin, R. Luo, Q. He, S. Wu, Z. Zeng, and X. Xia, "Cost-effective data placement in edge storage systems with erasure code," IEEE Trans. Serv. Comput., vol. 16, no. 2, pp. 1039–1050, Mar./Apr. 2023.

[6] W. Xiao, Y. Hao, J. Liang, L. Hu, S. A. Alqahtani, and M. Chen, "Adaptive compression offloading and resource allocation for edge vision computing," IEEE Trans. Cogn. Commun. Netw., to be published, doi: 10.1109/TCCN.2024.3400820.

[7] J. Peng et al., "MagNet: Cooperative edge caching by automatic content congregating," in Proc. ACM Web Conf., 2022, pp. 3280–3288.

[8] L. Yuan et al., "CSEdge: Enabling collaborative edge storage for multiaccess edge computing based on blockchain," IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 8, pp. 1873–1887, Aug. 2022.

[9] R. Li, Z. Zhou, X. Zhang, and X. Chen, "Joint application placement and request routing optimization for dynamic edge computing service management," IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 12, pp. 4581–4596, Dec. 2022.

[10] X. Xia, F. Chen, Q. He, J. Grundy, M. Abdelrazek, and H. Jin, "Online collaborative data caching in edge computing," IEEE Trans. Parallel Distrib. Syst., vol. 32, no. 2, pp. 281–294, Feb. 2021.

[11] J. Zhou, F. Chen, G. Cui, Y. Xiang, and Q. He, "FEUAGame: Fairnessaware edge user allocation for app vendors," IEEE Trans. Parallel Distrib. Syst., vol. 35, no. 8, pp. 1429–1443, Aug. 2024.

[12] X. Ge, S. Tu, G. Mao, C.-X. Wang, and T. Han, "5G ultra-dense cellular networks," IEEE Wireless Commun., vol. 23, no. 1, pp. 72–79, Feb. 2016.