DRS: A **D**eduplicated Scheme in **R**edundant **S**torage for Cloud with Client-side Data Encryption

Sabbir Ahmed[1], Md Nahiduzzaman[2], Tariqul Islam[3] and Faisal Haque Bappy[4]
[1,2] Institute of Information Technology, Jahangirnagar University, Dhaka, Bangladesh
[3,4] School of Information Studies (iSchool), Syracuse University, Syracuse, NY, USA
Email: {*sabbir.iit.ju@gmail.com, nahidsamrat2@gmail.com, mtislam@syr.edu, fbappy@syr.edu*}

***Abstract*—With the growing popularity and quick development of cloud storage technologies, an increasing number of enterprises are beginning to migrate their valuable data to cloud storage platforms. However, cloud storage is limited and digital data is growing exponentially, cloud service providers need to ensure storage efficiency but in contrast, they also need to ensure high availability and reliability which incurs storage costs. Furthermore, the problem is exacerbated if the storage server saves multiple copies of the same data without an effective technique for detecting and eliminating duplicates. This can significantly increase storage space usage while also wasting network bandwidth. In this paper, we have proposed a cloud storage scheme that simultaneously achieves storage efficiency, security, reliability, and high availability. Our scheme first ensures specific fault tolerance by creating subsets of data blocks and then dispersing these subsets to optimum servers while ensuring data deduplication. To ensure high availability we have proposed several algorithms that create erasure coding style data partitions and dispersal while also maintaining duplications in multiple server scenarios. We have also proposed an intermediate processing system that handles security and reliability for the client devices while also minimizing query complexity in cloud servers. Both theoretical and experimental results show that the system can achieve high fault tolerance for single, group, or public users with file or block-level deduplication by using a less storage scheme.**

***Keywords*:** Security, Reliability, Authentication, Deduplication, Merkle Tree, Erasure Coding.

I. Introduction

In this big data era, cloud computing provides a revolutionary mechanism for data owners to share their outsourced data with authorized users in which enterprises, hardware, and software designs and procurements are shifting. Customers expect to be able to use on-demand cloud services at any time as the amount of data in the cloud grows, while providers must maintain system availability and handle a significant volume of data. Providers want a method to drastically reduce data volumes to save money while maintaining massive storage systems

Data deduplication allows CSPs to store a copy of data while deleting duplicate copies, achieving the objective of saving storage space and network bandwidth [1]. But this approach can lower availability and reliability, the most fundamental cloud services. In cloud computing, redundancy is the best way to ensure that our data is accessible, safe, and secure regardless of what happens to individual servers. Therefore, we must maintain redundant data to assure increased availability. The cost of storage can, however, go up if there is too much redundant data. So deduplication increases storage efficiency while redundancy is kept to increase availability. So, our goal is to find a solution where we can balance the degree of deduplication and redundancy to design an efficient, highly reliable storage system.

Data redundancy is used in a cloud storage system to intelligently distribute data among clouds. As a result, the redundant data distribution method is crucial for storage availability, storage efficiency, and effectiveness. Data recovery from a subset of clouds has been made possible by reliability-enhanced technologies (such as replication or erasure coding), even if the other clouds are inaccessible. RACS [2] uses RAID-like techniques which are used by the disks and file systems at the cloud storage level that can transparently spread the storage load over many providers. RACS lowers the one-time cost of changing storage providers in return for more communication overhead. By utilizing techniques adopted from the cryptographic and distributed-systems communities, HAIL [3] uses e Proofs of Retrievability (PORs) as building blocks that depend on a single trusted validator to maintain file integrity and availability across a number of servers or different storage providers. NCCloud [4] employs regenerating codes to tolerate cloud failures with significantly less storage overhead while maintaining the same level of data redundancy and storage requirements as traditional erasure codes (e.g., RAID-6). While the previous three methods are all based on the erasure code or the network code, DuraCloud [5] uses replication to duplicate user data to multiple distinct cloud storage providers to improve availability. Furthermore, it guarantees that all copies of user material are synced. However, consumers must pay more for the increased bandwidth and storage space that DuraCloud needs. DEPSKY [6] also uses the replication strategy to improve availability by combining the Byzantine quorum system protocol and cryptographic secret sharing. HyRD [7], first integrates erasure coding with a replication strategy which is different from these approaches, and tried to achieve storage efficiency with high availability guarantee. However, his design does not remove duplicate data blocks on the network.

According to workload research carried out by Microsoft [8] [9], and EMC [10] [11] suggests that about 50% and 85% of the data in their production primary and backup storage systems, respectively, are redundant. According to IDC research [12], about 80% of the organizations assessed were considering data deduplication technology in their storage systems to remove duplicate data and therefore boost storage efficiency and lower storage costs. Douceur [13] proposed convergent encryption as the first solution for secure and effective data deduplication. This concept developed a variety of wide uses, with numerous methods based on convergent encryption being deployed or planned [14] [15].

Recent research, such as RACS [2], HAIL [3], NC-Cloud [4], DuraCloud [5], DepSky [6], and HyRD [7], show that replication-based schemes are more performance-friendly for ensuring improved availability and reliability, but deduplication-based schemes [14] [15] are more cost-effective. Because unwanted redundancies cost money, enterprises would profit greatly from client-side data deduplication before outsourcing their data to the cloud. To overcome this issue, DAC [16] combines a replication technique to save data blocks with a high reference count and erasure codes to store the remaining data blocks across various cloud storage providers. It uses fixed-size blocks with SHA1-based or MD5-based hash computing algorithms but this scheme lacks security and is vulnerable to attackers as they didn't use any encryption protocol. They also failed to mention the exact working procedure of how the chunking algorithm, erasure coding scheme, server selection, and effective data structure for working deduplication. So, we have found the following question that should be addressed to achieve a storage efficient highly reliable cloud system. **RQ1:** Can we create a balance between deduplication and redundancy that can achieve storage efficiency and higher availability simultaneously? **RQ2:** Can we create a rating mechanism during data dispersal that can choose the best server among all available servers to keep duplication as low as possible? **RQ3:** Can we create optimum grouping of blocks while reducing data servers in such a way that faul tolerance in maximized?
Since the existing algorithm for each task works independently and inversely related to each other, the balance required to ensure both deduplication and redundancy for storage efficiency and higher availability, which is the most crucial part of the solution that has yet needed to be solved. Besides, during data dispersal to multiple storage servers, an optimal server selection mechanism is needed to choose the best servers among all to ensure user-defined redundancy. Thus, in this article, we propose several algorithms for ensuring fault tolerance, including optimum subset creation from data blocks and redundancy factors and selecting appropriate servers based on rating calculation. We have also proposed data structures, dispersal, and restoration algorithms to ensure data deduplication with inconsistency checks and security.

Objectives. Our research aims to present a distributed storage solution that combines security, higher availability, and storage efficiency.

Contributions. Following are the main contributions of our work:

- We propose a novel secure deduplication strategy that successfully reduces duplication at both the file and block levels while assuring user-defined availability and greater fault tolerance in multiple redundant server scenarios.
- We propose a novel algorithm that may select several optimal servers from a pool of maximum assigned servers to provide user-defined redundancy. We arrange distinct data blocks into subsets in this approach to guarantee maximum fault tolerance.
- We propose a custom data structure using an ordered map and matching algorithm, that can work as a load balancer by calculating a duplication rating (server load, server distance, redundancy factor) for each subset to ensure minimal duplication in redundant servers.
- We provide security analysis and performance evaluation of our scheme, and the results show that our scheme is secure and efficient.

The rest of the paper is organized as follows: In Section II, we review some preliminaries and cryptographic primitives along with a few well known security algorithms and protocols. In Section III, we describe our system model. In Section **??**, we present our proposed scheme in detail, followed by theoretical performance analysis and experimental evaluations in Section V and Section VI respectively. We present some related work in Section VII. Finally, we conclude the paper in Section VIII.

II. Preliminaries

A. Convergent Encryption and Deduplication

Convergent Encryption is a cryptographic operation that produces an identical ciphertext from duplicate files. A convergent key generated by computing the cryptographic hash value of the data's actual content is used by a CE scheme to encrypt or decrypt the data copy. If a user wants to upload duplicate data, the server discards the data and returns an ownership pointer to the uploader. When storing deduplicated data, CE seeks to provide data confidentiality by encrypting a message F using a message-derived key K. A convergent encryption system can be described formally as follows:

1) $KeyGen_C E(1_k, F) \rightarrow K_C$: A randomized cryptography key generation algorithm requires a security parameter together with a message F as input and generates a convergent keyK_C ;
2) $Encrypt_C(K_C, F) \rightarrow C$: A randomized symmetric key encryption process utilizes the convergent key K_C and the message file F as input and outputs the ciphertext C;
3) $Decrypt_C(K_C, C) \rightarrow F$: This decryption technique uses the convergent key K_C as well as the ciphertext C, and outputs the original plaintext file F;

B. Merkle Hash Tree

Merkle hash tree is a type of hash-based data structure that is used to authenticate digital data while requiring less computation and communication cost. Every internal node in the structure stores a hash value that is the concatenation of the internal node's left and right children's hash. We divide a file into blocks, couple the blocks, then hash each pair with a collision-resistant hash function to create a Merkle tree. The hash values are again paired off and each pair is further hashed. This procedure is repeated until there is only one hash value left. The Root of the tree is the last single hash value.

C. Hashmap

Data structures with indexes are hash maps. When creating an index with a key into an array of buckets or slots, a hash map uses a hash function. The bucket with the relevant index is linked to its value. The key is distinct and unchangeable. Implementing a hash map relies primarily on hash functions. It accepts the key and converts it to the bucket index from the key. A separate index should be generated via ideal hashing for each key. But crashes can happen. We may easily utilize a bucket for multiple items by adding a list or by rehashing when hashing yields an existing index. Dictionaries are an example of hash maps in Python. Hash maps include the following functions:

1) $SetValue$(key,value): A key-value pair is inserted into the hash map. This function updates the value if it's already there in the hash map;;
2) $GetValue$(key): If there is no mapping for the given key in this map, this method returns ”No record found” or returns the value to which the given key is mapped;;
3) $SetValue$(key): Deletes the mapping for a certain key if it is present in the hash map;;

III. System Model and Design Principles

A. System Model

Our proposed system consists of the following three entities as illustrated in Fig 2.

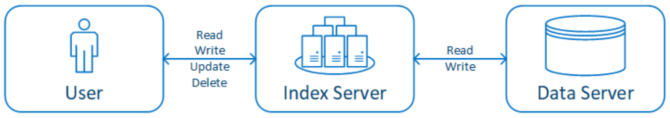


Fig. 1: System Model

Cloud Users. *Cloud Users* is the data owner of the files. Cloud users (also known as group members) are authorized users of the cloud. When they initially connect to the system, they must register with the index server. They can upload and download their own outsourced data by providing proof of ownership. Cloud users can be a total of 3 kinds: public, group, and private users. *Group User:* A standard key will be generated for all group users though they can have a different key, a keygen algorithm will yield the same CE key. So, we assume that they will not have any collision. *Public User:* For the public user, they upload a file using their own encryption key which is generated from the data. *Private User:* A private user gets a unique key and acts like a group itself.

Index Server. *Index Server* is responsible for system parameter generation, user registration and revocation. Before a file or a block of a file is outsourced to the cloud, it stores the corresponding file and blocks authentication tags. An Index Server (IS) communicates with the user and cloud server. An IS distributes data chunks to multiple cloud servers based on redundancy. The other task includes data deduplication check, file index management, and record and maintenance of abnormal events.

Assumption 1: The cloud server cannot be trusted, but the index server can. Users can submit the primary file to the IS unencrypted in a trusted situation. The file is then used by IS to create blocks and subsets before being sent to the cloud server. It is impossible for any attackers to obtain the file since the data blocks are randomly distributed among several cloud servers based on the redundancy factor and only IS preserves the user file. To ensure more security, users can upload encrypted data to the IS.

Assumption 2: IS and Cloud Server are both unreliable. In this scenario, users will create blocks and encrypt them on their end before uploading cipher data to the IS. However, as it serves as the convergent key, users must save the hash value of each block to his/her end. This may increase the computation overhead but will ensure security and reliability.

Cloud Server. The CS provides data storage services with extensive storage backup facilities for its client on a pay-as-you-go basis over the internet. A CS would like to delete duplicate files to minimize its storage cost. On the other hand, a single file will be stored on more than one CS to ensure higher availability.

IV. Deduplication in Redundant Server - DRS

In this section, first, we define several aspects, structures, and functionality of the proposed *DRS* system, which will be later utilized by various user-level operations to perform Read, Write, update and delete operations. Deduplication is used to reduce data redundancy, whereas redundancy is kept to ensure the higher availability of cloud services. In this system there are several key components that transform the user data into block-sized duplication checked redundant services. These components include some custom data structures to keep track of users, files, and data servers, a data processing system for fragmentation and tag generation, optimum subsets of data, and server selection. The main working procedure is as follows:

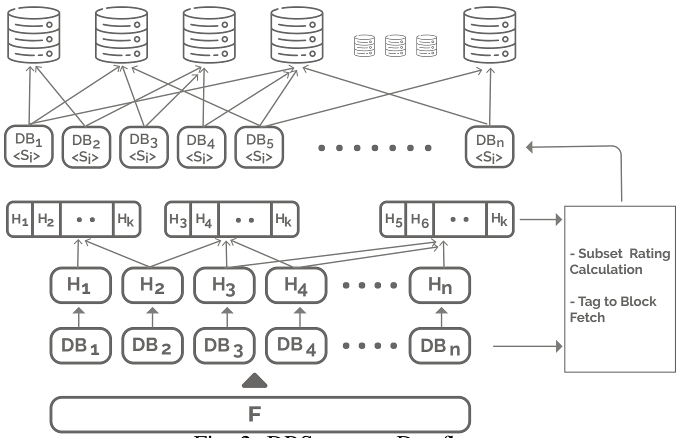


Fig. 2: DRS system Dataflow

A. System Initialization

Notation Table:

Notation	Description
U_{id}	User Id
Ma	Memory allocation
DS_{id}	Data server Id
AVL	Availability
IS	Index Server
F_j	Files
F_{id}	File id
H_k	Hashing
DB	Data Block
$DBCon$	Concatenated Data Block
$HCon$	Concatenated Hash
MHT_{root}	Merkle Hash Root
K_{CE}	Convergent Key
C_y	Ciphertext
S_{max}	Maximum allocated servers
S_{opt}	Optimum servers
$divs_i$	Divisors
AVL_{Space}	Available Space
S_l	Server List
R_f	Redundancy Factor
DB_{ss}	Data Block Subset
H_{ss}	Hash Subset
H_{DB}	Hash of Data Blocks
DD	Deduplication
DP	Dynamic Programming

The system is initialized with parameters for the client, index server, and data server..

Data Server Construct: Data Server (Ds) has a separate but defined number of memory allocation (M_a) with unique Identification (Ds_{ID}), with another property that determines the availability (AVL) of that server. When the data server is online, only the Index Server (Is) can access this information

Client/ User (property) in his class: User will also be initialized with a unique ID (U_{id}). Each member of the same group is given a user id, which is used to construct the convergent key. User Registration: First, users will register with an Index Server. Then IS will give a group key to its registered users in a similar group. A user can be register without any key. Authentication: A challenge-response protocol between user and cloud server. Authorization: Each user is provided several degrees for read, write, update and delete.

B. Data Structure

The Index server additionally keeps track of a collection of information regarding users, data servers, and previously uploaded data in addition to the characteristics stated in the initialization section. We have presented three data structures, primarily based on hash maps, to access them in the shortest amount of time feasible.

User-File Users (U) is a complex unordered map structure consisting of several User (U_{id}) which contain several files (Fj). Again for each file, lists of hash (H_k) are arranged in a particular order so that, a sequential read of hashes eventually creates the targeted file. Each user has a unique ID (U_{id}) which is used as the key of Users (U_i), where the data section consists of several unique (F_j). Thus any hash can be accessed by querying the user's data structure with $U - id$, $F_i d$, and H_k in O(1) time, where to access the file just need to specify the U_{id} and F_{id}. Similarly, any number of U_{id}, $F_i d$, or Hk can be inserted and replaced or deleted in constant time complexity. To add a new U_{id}, there must be no similar U_{id} in the existing structure which is also true for $F_i d$, although H_k does not require that same criterion because many files can consist of the same data blocks which result in the same hash.

Algorithm 1: Data Processing: (F, DB_s)

```
input  : $F, DB_s$                          ▷ file and block size
output: $< H_k, DB_i >$                     ▷ tags and blocks
1 $K_c \leftarrow$ KeyGen(seed);            ▷ group key
2 $K_{ce} \leftarrow$ Hash($K_c$);          ▷ convergent key
3 $Cy \leftarrow$ Encrpyt($K_c$, $F$);
4 $< DB_i > \leftarrow$ Fragment($Cy$, $DB_S$);
5 for each $DB_i \in < B_i >$ do
6     $H_k \leftarrow$ TagGen($DB_i$);      ▷ Tag for blocks
```

Data Server The data server (DS) is a similar unorder data structure residing in the index server, to keep track of the data server available and used memory address (Ma), where multiple servers (DS_i) is identified with a unique key ($DsID_i$). In each DS_{id}, a boolean list is created with the corresponding memory location (Ma_j) as key, whereas is boolean value True or False delineates whether the memory address is available or not. When a new DS_i is initialized, all of the memory addresses are added on the respective $DsID_i$ by keeping the availability True. A new data server can be added to data server structures although the Memory location can not be added or deleted.

Hash Map Our Hash Map (HM) is a really simple but effective unorder map structure with a hash(H_k) as key, where each H_k is authentication and also a general purpose tag for Data block (DB_i). For each H_k several pairs of ($DsID_j$), Ma_j can be assigned to describe the exact memory location and server identification for data retrieval. Since each data is saved on multiple servers, from any H_k we can determine whether it exists or the corresponding data blocks need to be updated. Any number of $DsID_j$, Ma_j pair can be added, replaced, or removed from corresponding H_k in O(1) time complexity for one operation.

MHT Construct To verify the system's integrity, we build a Merkel Hash Tree (MHT) using our structure. First, we use the Merkle tree technique to create a Merkle root MHT_root from file F. The user then determines whether or not the root matches his stored root. If a match is found, the user may be certain that the integrity of his or her file has not been compromised; if not, they can notify the service provider.

C. Data Processing

Key Generation: As we use convergent encryption, the key will be generated by hashing the file itself. We use the symmetric AES encryption technique so that the same key will be used for encryption and decryption.

Encryption/decryption: After the key is generated from the file, we will apply the padding operation to fragment the data into fixed-size blocks. Then the file will be encrypted using AES 256 symmetric cryptography algorithm. As we are using convergent key, same data will return the same ciphertext. $enc(m) = E(H(m), m)$

Fragmentation: Data is fragmented into data blocks(DB) based on the block size (DB_s) specified. The number of data blocks will be $len(data)/DB_S$.

Tag Generation: We will generate an authentication tag for each block by hashing the block using the SHA256 algorithm. We also keep a file tag by keeping the hash of the entire file to imply a file level integrity check.

D. Data Redundancy

1) Optimum Servers and Subsets: Since blocks can be send to any number of available DS, the exact number of optimum servers from max-user-server and fixed redundancy is needed to calculated.

Algorithm 2: GetServerList($N, \{S_{max}\}$)

```
input : $DB_N$, $\{S_{max}\}$   ▷ number of blocks and maximum available server list
output: $\{S_i\}$, $S_{opt}$   ▷ list of available servers and number of optimal servers, respectively
1  $\{divs_i\} \leftarrow 1$;
2  $size \leftarrow$ sqrt($DB_N$) + 1;
3  for $i \leftarrow 1$ to $size$ do
4      if $DB_N$ mod $i$ == 0 then
5          if $i <=$ length($\{S_{max}\}$) then
6              $\{divs_i\} \leftarrow i$;
7          if $DB_N/i <=$ length($\{S_{max}\}$) then
8              $\{divs_i\} \leftarrow DB_N/i$;
9  $S_{opt} \leftarrow$ max($\{divs_i\}$);
10 $\{S_i\} \leftarrow$ init();
11 for $i \leftarrow 1$ to $S_{max}$ do
12     $AVL_{Space} \leftarrow$ Available($i$);
13     if $AVL_{Space} >= DB_N$ then
14         $\{S_i\} \leftarrow i$;
15 return $\{S_i\}$, $S_{opt}$
```

Problem Definition: Given a list of all servers (S_L) for any particular user, with number of total data blocks (N) for specific file and redundancy factor (R) find out the optimum number of servers(S) that can used to store these blocks.

Lemma 1: Since N data needs to equally distributed among S server, N must be divided by S, in other words S must be divisor of N. Proposition: The largest value of S when lemma 1 is fulfilled, which is less then the number of maximum server (Smax) should be the optimum answer.

Proof : , Let $divs_N$ is the set representing the divisors of N. Then,

$$S = Max(divs_i < S_{max}) \qquad (1)$$

is the largest possible value of S that satisfied the lemma 1 and less than maximum possible value of S. Thus the largest possible number of servers (S) will ensure the maximum distribution of data which in turns maximize the fault tolerancy which is the optimum result in this case. To find out the divisors (N_div), the algorithm of sieve prime factorization can be used, where we can check all the number (X) from 1 to SQRT(N)+1 to find out if that number divides N or not. If X divides N then both X and N/X are both divisors of N (Algorithm: 2 line 1-8). Then we can use Eq1 to find out the Optimum number of server S. After that S_{opt} server ID will be chosen from S_{max} based on availability (line 10-14)

2) Maximum fault tolerance subset: Data blocks may have several combination of subset, in this section how to do that in most efficient way.

Problem Definition (Application): N Data with R copies need to be stored in S servers. What is the most efficient and Fault Tolerant way to do that?

Assumption (Application): If we can spread data evenly among servers, that could reduce the points of failure. The length of data sent to each server must be the same to produce that even distribution.

Problem Definition (Mathematical): How to make an S number of fixed size subset from N numbers, where each number occurs precisely R times?

Observation and proposition: Let's choose the length of the subset as L. For simplicity, also constraints that L is less Than N is size.

Then there could be Sub= ${}^{N}C_L$ possible subsets, and from them, we need to choose an exact S where the summation of occurrence of each number is strictly R. So the number of possible combinations is Sub(C) S. Now for any N, that's an NP-hard problem to calculate: Lets Choose N= 1000, L=500 Then, sub= ${}^{N}C_L$ = 1000C500= 2.70288E+299 Now we have Sub(C) S Choice= (2.70288E+299)C(100) = Uncountable in modern computer

Corollary: As a combinatorics problem producing this massive number of subsets, memorizing(DP) and combining them in problem-defined ways is NP.

Lemma 1: Since we need to distribute N numbers with R redundancy, total numbers are N*R

Lemma 2: Since we need even distribution, i.e., the length of each subset is equal, then (N*R) must be divisible by S.

Lemma 3: For any arbitrary value of S, N, R, there are two scenarios possible and impossible. That is, there are some cases where even distribution or equal subsets are impossible to make.

Lemma 4: For any arbitrary value of S, N, R, the Possibility can be checked by (N*R)%S==0. (3 line 4-5) Otherwise, S can be chosen from the set of the proper divisor of (N*R). This will remove the checking steps.

Postulate : Place 1 to N numbers, then repeat R times sequentially, take only (N*R)/S numbers at a time (3 line 10-11). That will result in exactly S subsets. (Here S, N, and R are chosen such that they fulfill lemma 1-4)

Proof : To prove the optimality of the solution we have to fulfill two separate conditions. First, each data block must occur exactly R times. Since the whole data is copied the exactly R time, so each of the data blocks must occur exactly R times in a concatenated state which will be $DBCon$ (3 line 7-9). Similarly $< H >$ will be also concatenated as $HCon$. Secondly, two or more same data blocks must be placed at the highest distance possible, thus two same blocks may not be placed in one subset. If two or more same data blocks are placed in one subset, then by losing one subset, that part of the data is lost, which will reduce the fault tolerance. Since there are N data blocks, and they are arranged sequentially, so the maximum distance between two same data can be N, which is achieved in this solution.

Algorithm 3: `FTSubset`($\{DB_i\}$, $\{H_k\}$, S_{opt}, R)

```
input : $\{DB_i\}$, $\{H_k\}$, $S_{opt}$, $R$    ▷ blocks, tags, # optimum servers and redundancy factor
output: $\{H_{SS}\}$,$\{DB_{SS}\}$    ▷ subsets of tags and blocks
1  $size \leftarrow$ length($\{H_k\}$);
2  $DBCon[\,] \leftarrow$ init();
3  $HCon[\,] \leftarrow$ init();
4  if $((size \times R) \bmod S_{opt})\ !=0$ then
5      return "Error";
6  $ssz \leftarrow (size \times R)/S_{opt}$;    ▷ subset size
7  for $i \leftarrow 1$ to $R$ do
8      $DBCon$.Add ($\{DB_i\}$);
9      $HCon$.Add ($\{H_k\}$);
10 for $j \leftarrow 0$ to $(size \times R)$ do
11     $\{DB_{SS}\} \leftarrow DBCon[j : j + ssz]$;
12     $\{H_{SS}\} \leftarrow HCon[j : j + ssz]$;
13 return $\{H_{SS}\}$, $\{DB_{SS}\}$;
```

E. Rating Calculation

DB_{ss} subsets of data blocks need to be distributed in S_max data servers where $DB_{ss} <= S_max$, where two subsets can not be assigned to the same data server (DS) to assure fault tolerance, for each S_DBi select DSi that maximizes the overall Score

Duplication Matching For each data block DB_i in DB_{ss} subsets, the corresponding tag or hash (H_k) is also calculated in the data processing section (Tag Generation). Again, (4 line 1-6) for any Key Hk, Hashmap (HM), gives the server ID ($DsID_i$) and memory location (MA_j) pairs to retrieve the data block (DB_i) of (H_k). The unavailability of H_k in HM also can query in O(1) time. For each Subset DB_{SSj}, duplication for each data server DS_i is calculated and stored

Algorithm 4: `Rating`($< H_{SS}\}$, $< S_ei\}$)

```
input : $\{H_{SS}\}$,$\{S_i\}$    ▷ tags subset, Servers list
output: $\{H_{SS},\ DB_{SS}\}$    ▷ subsets of tags and blocks
1  $< DP\}, < DD\} \leftarrow 0$;
2  for each $Hsi \in < H_{SS}\}$ do
3      for each $Dj \in < S_i\}$ do
4          $< DD_{Hsi,Dj}\} \leftarrow$
               $< DD_{Hsi,Dj}\}$+findLocation($H_k \in Hsi$);
5          $< RC_{Hsi,Dj}\} \leftarrow$ Available($Dj$);
6          $< Sl, Qs, Dis\}_{Hsi,Dj} \leftarrow$ GetVal($Dj$);
7  $< R_{i,j}\} \leftarrow$ WeightedRating($< DD_{i,j}\}$,
       $< RC_{i,j}\}$, $< Sl_{i,j}\}$, $< Qs_{i,j}\}$, $< Dis_{i,j}\}$ );
8  for each $i \in < H_{SS}\}$ do
9      for each $j \in < S_i\}$ do
10         $< DP_{i,j}\} \leftarrow < R_{i,j}\}$ +
               findMax($DP_{i+1,j+1}$, $DP_{i+1,j-1}$)
11 $< DS_i\} \leftarrow$ PathPrint($< DP\}$);
12 return $< DS_i\}$;
```

in tabular form, where the DB_{SSi}'th row and DS_i'th column represent the duplication for corresponding DB_{SSi} and DS_i.

Other Factors From DS structure in any IS, the number of True value that is available storage in any DS_i can be counted to measure how much storage any DS_i have available, similarly by counting the false value, used storage space can also be calculated. Available free space for each DS_i can be calculated by counting the availability from IS. Other factors such as server load, server distance, and bandwidth have been set as constant for each DS_i since the test environment does not comply with those requirements.

Weighted Rating Weighted final rating (4 line 7) can be computed by taking the weighted sum of all of the scoring criteria. For the system, the constant that would be multiplied with each criterion is set as a decreasing geometric series, such that the first criterion contains the most weight in scoring. Since these criteria can vary depending on system requirements ie: some systems might require higher weight on server load than it can be placed first.

$$TotalScore = R_S \times a + S_L \times b + Q_S \times c + D \times d + R_C \times e \quad (2)$$

where deduplicated Redundancy Score = R_S, Server load = S_L, user-specified query size = Q_S, Distance = D, Remaining Capacity of the server = R_S. Here a,b,c,d,e Can number from series like $alpha * (1/2)^x$ or any arbitrary number.

The total rating forms a Rating[Subset][servers] size 2D matrix where each of the R_{ij} represents the rating for sending ith subset to jth data server and $0 < i <= number$ of subsets(SS_n), $0 < j <= number$ of data servers(DS_n). To get the combination with maximum score, lets assume DP(i,j) = max(0 to i, 0 to j) + Rating(i,j) + dp(i+1,j+1) or dp(i+1,j-1). The time complexity evaluation of the recurrence is $O(SS_N * DS_n)$. That is there are $SS_n * DS_n$ states where we have take the maximum pairs only once.Thus we calculates all of the ratings for server and subsest and memorize that on DP(SS_n, DS_n) table (4 line 8-10). To find out the optimal server i for each subset j, then we need follow the path for maximum value of dp[0][0], which that path will eventually give unique i,j pairs such that each i and j will be only ones.

F. Data Dispersal

When a user wants to upload a file F, there can be two scenarios: i) the file is being uploaded for the first time, or ii) the file has already been uploaded by another user from the same group.

First Upload. To write any file, the user first takes an Authentication key using the key generation function, then encrypts the data using CE with AES. The encrypted data is sent to the index server (IS) with U_{id}, F_{id} with some optional parameters like the Redundancy Factor R_f, maximum server $\{DS_{max}\}$, etc. The IS authenticates U_{id} and checks if it has any write privileges or not. Then IS invokes a check to find if that F_{id} exists or not. If it exists then the update method takes over, otherwise it is sent to the fragmentation and tag generation process to partition the data into data blocks DB_n and create corresponding Hash H_k. Then based on the length of DB and maximum data server (DS_{max}) allocated for the user, the number of optimum servers (DS_{opt}) is calculated using algorithm 2. DS_{opt}, DB_i, H_k, and R_f are used by the subset creation algorithm 3 to make the optimal subsets (DB_S) that maximize the fault tolerance. Here subsets fulfill three conditions, 1) same data block placement ensures maximum distance such that two same blocks never occur in one subset. 2) the number of subsets strictly equal to the number of optimal servers. 3) In all subsets each block occurs exactly at R_f time. Then Rating calculation is invoked (algorithm 4) with DB_S, H_k, and DS_{opt} to find out which subset should be sent to which data server. Rating calculation takes several factors: duplication, server storage, server load, etc using eq:2, and creates an overall weighted sum of rating. Then for each DB_i, one DSi is selected such that all of the DB_i have different DS_i and the overall combination maximizes the total rating. Thus now we need to send each DB_i to the DS_i by checking if H_k is already in the hash map (HM). Then IS finds out the empty memory location (MA_j) for that DS_i and requests DS_i with MA_j to store DB_i. Then IS Store DS_i, MA_j pair with key H_k in HM, update the data server map by making MA_j address unavailable, and also store H_k in user map for further recreate the file. If sever IS is involved in the process all of them update their internal data structure similarly.

Update Users who want to update their file must first initiate four fundamental data processing processes (key generation, encryption, block creation, tag generation). After that, the Index server will find the updates that need to be done by users. IS will achieve this by comparing old and new block hashes and saving the differences in an array called $diff_{hash}$. Following that, the write function is invoked, and data is stored depending on the redundancy factor and optimal server rating calculation. After completing the write operation, the data server's availability status will be adjusted so that data subsets are distributed to all servers and overwrite is avoided. The index server will then need to update the new subset location in the server as well as the associated server id in the hash table. Finally, the user table will be updated to reflect the ownership of new block hashes.

Algorithm 5: FirstUpload(F,DB_S)

```
input : $F$,$DB_S$                    ▷ File and blockSize
output: $< H_k, DB_i >$               ▷ tags and blocks
1  $< DB_i, H_k > \leftarrow$ Data Proccessing
     ($< H_k >, < DB_i >$);
2  $< S_i >, S_{opt} \leftarrow$ GetServerList($DB_N$,
     $< S_{max} >$);
3  $< H_{SS}, DB_{SS} > \leftarrow$
     faultTolerantSubset($< DB_i >$,$<$
     $H_k >$,$S_{opt}$,$R$);
4  $< DS_i > \leftarrow$ Rating($< S_n, S, S_{total} >$);
5  for each $< DB_i, H_k >$ do
6      $< isDup_i > \leftarrow$ HashMap($< H_k >$);
7      if $< isDup_i >$=TRUE then
8          return blockPointer($H_k$);
9      $< MA_j > \leftarrow$Check Availability($DS_i$);
10     Upload($DB_i$,$MA_j$,$DS_i$);
11     DataServer[$DS_i$][$MA_j$]$\leftarrow$True;
12     Update Hash Map;
13     Insert Tag in User File;
```

Algorithm 6: Update(F,DB_S)

```
input : $< F, DB_S >$                 ▷ Hash and blocks
output: $< H_k, DB_i >$               ▷ tags and blocks
1  $diff_{hash} \leftarrow$
     SetDifference$< new_{hash}, old_{hash} >$;
2  for each $< B_i, H_i >$ do
3      $< isDup_i > \leftarrow$ Duplicate($< C_i >$,$F$);
4  if $< isDup_i >$=TRUE then
5      return blockPointer($C_i$);
6  else
7      Check Availability;
8      Upload($C_i$,$K_{C_i}$);
9      Update Availability Status;
10     Update Hash Map;
```

Data Restoration For a read request, the user requests the IS with U_{id} and F_{id}. Each ISj validate the request, and using UID and FID find out the sequential Hk from user map data structure. Then for each Hk, IS read $< DS_i, MA_j >$ pair and request DSi to send data stored at MAj location. Since for each Hk there are several DSi, MAj pair, so several DBij collected from different DSi. The IS then invoke the integrity and inconsistency check, where among all DBij maximum matching DBi is selected. Again from DBi using tag generation a new tag is generated, and matched with existing

Hk to check that both of them are same. If both criteria are satisfied then the DBi is appended to the data. This process continue for all Hk in FID, and create the whole data from DBi. The concatenated data is sent to the user, where user can decrypt and access the original data that was stored previously.

Algorithm 7: Read $< U_{ID} >, F_{ID}$

```
input : $U_{ID}$,$F_{ID}$                 ▷ User ID, File ID
output: $F$                               ▷ Complete File
1  $H \leftarrow$ GetTags($U_{ID}, F_{ID}$) ;
2  for each $H_k \in H$ do
3      $< DS_i, MA_j > \leftarrow$ LocationOf($H_k$);
4      $< DB_i > \leftarrow$ GetData($< DS_i, MA_j >$);
5      $< DB_i > \leftarrow$ RemoveNone($< DB_i >$);
6      $DB \leftarrow$ MaximumVote($< DB_i >$);
7      $H_{DB} \leftarrow$ Hash($DB$);
8      if $DB$ Not = None and $H_{DB}$ == $H_k$ then
9          $F_k \leftarrow$ Add($DB$) ;
10     else
11         $F_k \leftarrow$ Add("ErrorMsg") ;
12 return $F$;
```

Block-level Deduplication: When calculating and uploading ratings for two instances, the block-level duplication is checked. The index server keeps track of all the tags or hashes of blocks that have been uploaded using this index server in a HM data structure. After completing Algorithms 1,2,3,then Algorithm 4 checks each of the tags for the $< DB_i >$ to determine the degree of duplication. It does this by utilizing a HM_i, which can check for the presence of any given tag in O(1) time and return the MA_j and S_{ID} of that tag. The $< DB_i >$ must be saved if these return values are empty, so algorithm 5,6 sends that data block to the appropriate server.

File-level Deduplication:

When an Index Server (IS) receives a file upload request, it first checks to see if the server has the corresponding file authentication tag. If so, the server considers it a duplicate file upload request and prompts the user to use the user id to verify file ownership. If the verification does not succeed, the server terminates the upload activity of the file since the user is not permitted to access it. As the block size of our system varies and is user-defined, if the user sets the block size as same as the file size, our system can perform file-level deduplication also in comparatively less time.

V. Security and Performance Analysis

A. Security Analysis:

Confidentiality: On the cloud servers, information is kept in ciphertext form. Furthermore, they are randomly stored, making it impossible for a malicious attacker to obtain actual data from the storage server without knowing the correct memory location sequence and the matching server id. It is considered that the Index Server is not necessarily to be trustworthy. We can say that our design ensures confidentiality because only Index Server is aware of the server id and memory sequence and they are regularly auditable.

Authentication and Integrity: We store the hash root of each file and data block, which can effectively maintain data integrity and public auditing.

Fault Tolerance: Our system is secure from hackers because it runs on multiple servers. Our system can withstand attacks well because data is distributed randomly and encrypted across a number of servers. In our design, it is always possible to recover a piece of the data unless all servers are under attack.

Attack Resilience: We may state that our system is immune to brute force assaults because we implemented Convergent Encryption with the AES-256 encryption algorithm. Using brute-force methods, AES 256 is essentially impregnable. While a 56-bit DES key can not be hacked in a day, with current computer capability, AES would take billions of years to crack. Hackers would be unwise to undertake such an attack.

VI. Implementation and Evaluation

To implement our scheme, we have built a prototype of DRS in Google Colab [17] using python language. We use SHA-256 as our hash function and AES-256 for encryption. We conduct all cryptographic protocols using the pycryptodome library. We have a total of four variables to measure our performances and they are block size, redundancy factor, file size, and maximum allocated servers. For each test run, we have fixed any three variables and show the effect of changing another variable. First of all, in figure 3a, we have shown the effect of different block sizes where the file size was 128 MB, the Redundancy factor was set to 3, and we disperse the data maximum to 40 servers. Then we measure the performance of upload time, download time, and MHT-based integrity checking time. As we can see, there is a negative relationship between file upload time and block sizes whereas data download time and integrity checking time nearly remain constant for block sizes ranging from 8 KB to 8 MB. However, once the block sizes are 32 KB or more, the uploading time likewise stays fairly consistent. On the other hand, when file size fluctuates but the other 3 parameters stay the same in figure 3b, we have seen an increasing tendency. It takes less than 0.8 seconds to upload, download, and validate the integrity of a file up to 128 MB, while it takes 11.26 seconds to upload a 2GB file. For figure 3c, the redundancy level has been increased from 1 to 8, with a maximum server count of 20. The file size is 128 MB, and the block size is 64KB. As we can see, redundancy has no influence on download time or integrity checking time, but it has a nearly linear relationship with upload time.
According to our simulation, block and file size have no influence on fault tolerance, but the number of maximum allotted servers and redundancy factor has an upward tendency on a system's fault tolerance. In figure 3d, We raised the redundancy from 1 to 8 while maintaining the maximum server count of 20. We discovered that by simply keeping one copy of data, our system can withstand up to four server breakdowns, indicating that the system is 20% fault tolerant

which means data was dispersed to 16 servers out of 20 maximum allocated servers. When we maintain 4 copies of user data, our system can tolerate up to 16 server failures, and when we raise the redundancy to 8, only 2 servers out of 20 can recover the original data, giving the system a 90% fault tolerance. Although this is an upward trend the relation between redundancy and fault tolerance is logarithmic, that is the changing rate of fault tolerance reduces when redundancy increases. A similar trend can be found when we fixed the redundancy but increased the maximum allocated servers gradually in figure 3e. We have kept the redundancy at 5 and file size 128 MB and the block size fixed to 32 KB. Then we conduct the simulation for various file sizes and check fault tolerance by randomly shutting down a portion of the total servers. We have found that our system can give 87.5% fault tolerance when the maximum server is set to 80 which means our system only needs 10 servers to recreate the original data. This change percentage becomes nearly flat when we use more allocated servers like 120, 240, and 400 and the fault tolerance is 96.88%, 98.05%, and 98.44% respectively.

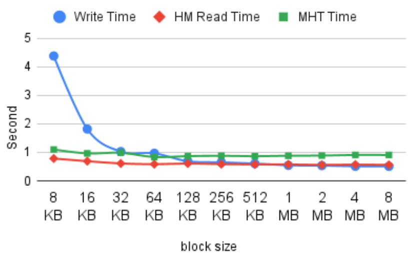
DRS calculated matching for all blocks for subsequent upload can also be found in any DB_i present among the current HM. Thus the exact number of matching blocks depends on the existing files on the IS. For this reason, we have emptied the full server, uploaded files with different sizes, then changed the data by a certain percentage and computed the time for subsequent update operations. Since the update has separate block-by-block matching, it took more time than the first write. Figure 3f represents times needed for a subsequent upload with 1 to 100% changed data, and for file size 64,128 and 256 MB. The redundancy factor for this experiment was fixed at 5, block size 32KB, and 20 max server. The time requires increases with the percentages of data changes with some fluctuation due to processing power. Another intriguing pattern is that data upload durations for a given file ID with 100% change are doubled compared to the initial upload.

In figure 3g, We have shown the comparison of batch auditing time between our proposed hash map with MHT. We have fixed the maximum number of servers to 40 and the block size to 64KB while keeping the redundancy factor at 3. We put a challenge to the servers to check the integrity of randomly chosen 1% data blocks for batch audit and the response time of the servers is comparatively low in our hash map while it takes considerably more time with MHT response. The bigger the file size, the response time is exponentially growing in MHT-based auditing protocols, while this time is less than half in our design.

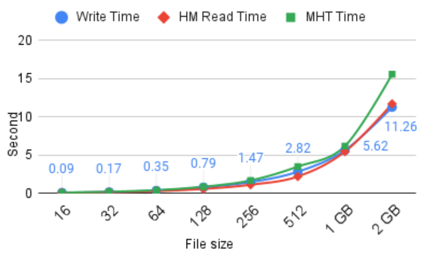
Redundancy, maximum server capacity, file size, and block size were the four factors that were kept constant in the studies, while one variable was changed over a range of values. To comprehend each variable's impact, computations of time required and fault tolerance are made for each. Since each server is mimicked using a different class, our experiment revealed that the number of servers has no bearing on the read, write, or upload operation times. However, since they don't alter the number of redundant blocks, file size and block size have no bearing on fault tolerance. Block size has an inverse relationship with computation time because it affects the total number of blocks. In contrast, computation time directly correlates with file size because it affects the total number of blocks. Redundancy only causes an increase in write time because more duplicate files must be saved in the data server. How fault tolerance is calculated is the percentage of servers that can be offline while still allowing for complete data recovery.A mathematical function was attempted to be combined with the data by producing more outputs.A mathematical function was attempted to be combined with the data by producing more outputs. Fault tolerance and redundancy have related either inverse exponentially($y = a \times x^{-b}$) or hyperbolically ($y = a + x \div b$).

VII. Comparison With Related Work

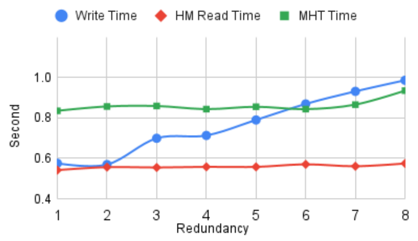
Using a Permutation Ordered Binary (POB)-based number system to split the data into many shares and a CRT-based secret sharing scheme to distribute the key into random shares, Sing et al. [18]'s storage scheme performs deduplication. This solution addresses the single point of failure issue for convergent encryption-based deduplication schemes. This method transforms important data into shares that appear random and distribute them across other servers to achieve fault tolerance. The deduplication process is applied to each of the shares in this scheme, which does not require a third-party authenticator but requires a little bit more computation time since separate shares are produced for each data block. To facilitate dynamic user management, Yuan et al. [19] provide a secure data deduplication approach with effective re-encryption and a bloom filter-based location selection mechanism. Convergent all-or-nothing Transform (CAONT) is used in this strategy, where bits are sampled at random and data owners are compelled to re-encrypt a tiny portion of the whole package using CAONT to lower system communication overhead. The message is encrypted using an MLE method, and CAONT sends the ciphertext to packages. The cloud user concatenates the last 256 bits of the package as a collected package with the last 256 bits of the randomly chosen package after creating a file key with the CP-ABCE algorithm. The cloud user then re-encrypts the collected package using the file key. The remainder is then uploaded to the cloud along with the assembled package. Li et al. [20] proposed a secure client-side deduplication approach based on Message-locked Encryption (MLE) and Proof of Retrievability (PoR), in which the client may check the duplicity of outsourced data by communicating with the server. This technique provided a dedicated key server to aid clients in generating MLE keys, which uses a rate-limiting strategy to restrict the number of quarries of the client during each epoch to withstand a brute-force attack. To prevent illegal content distribution, the Bloom filter-based proof of ownership approach is used. CSED uses the GDH signature rather than the RSA signature since the GDH signature is shorter and the communication speed is faster. However, CSED does not defend against an adversary who impersonates a valid client in



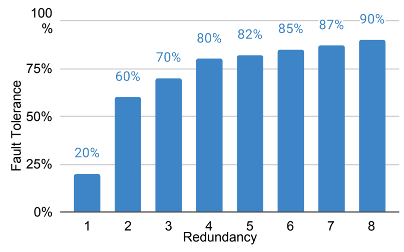
(a) Effect of Block Size



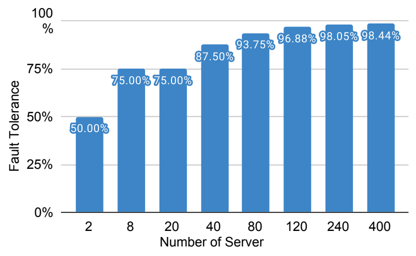
(b) Effect of file Size



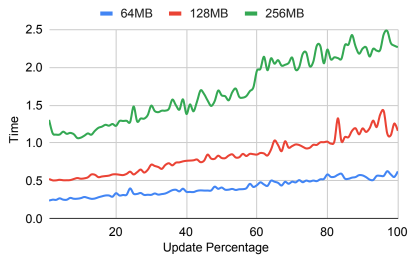
(c) Effect of Redundancy



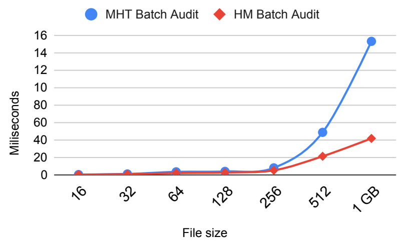
(d) Redundancy vs Fault Tolerance



(e) Servers vs Fault Tolerance



(f) Subsequent Upload Time



(g) Batch Audit Time

Fig. 3: Operation Completion Time for different Variables

order to receive the right answer values to the cloud server's challenge.

Yuan et al. [21] propose a batch auditing scheme with a secure deduplication scheme that can guarantee data consistency by implying additional consistent detection mechanisms. This approach substitutes TPA with a blockchain mechanism that automatically pays customers through a smart contract, whose data integrity is compromised, to ensure fair arbitration and removes the random masking in data auditing. But, this scheme works for the Ethereum blockchain only and does not support user revocation. Li et al. [22] propose a blockchain-based public auditing scheme that removes the TPA by storing the lightweight verification tag on the blockchain on the user side and generating proof by constructing MHT using the hashtag. To generate evidence during the auditing phase, the Data Owner (DO) must first ask a public auditor (another DO from the blockchain network) to create an MHT utilizing all of the DO's hashtags that are recorded in the blockchain. Then DO challenge the CSP in a message. For the DO in the cloud, the CSP creates the hashtags for the encrypted file blocks, and after that, it returns the evidence to the DO by building an MHT with the produced hashtags. The DO then verifies that the evidence produced by the CSP corresponds to the proof provided by the public auditor. Neelima et al. [23] proposed an Adaptive Dragonfly Algorithm (ADA) based load-balancing task scheduling scheme that can solve NP-hard optimization problems to ensure effective resource utilization in cloud computing. This multi-objective load balancing approach can provide minimum time and cost while can assist in moving work away from overburdened VMs and assigning them to underloaded VMs. They assign multiple tasks to different virtual machines using their proposed load balancing algorithm and illustrate better performance in completion time, processing cost, and load. The cost-based allocation (CBA), a technique for allocating resources that comply with data availability from redundancy models while taking into account the minimal availability level required by the user, is suggested by Goncalves et al. [24]. The system consists of a number of allocation algorithms that compute allocation costs and select the lowest-cost option. However, this system's recovery time after failure is not ideal. To solve replication issues and cloud storage governance, John et al. [25] offer a novel dynamic data replication approach based on the intelligent water drop (IWD) algorithm, which considers criteria such as bandwidth, user access, available storage space, and traffic. This approach clones the data based on the number of accesses or the data's popularity. The technique can increase storage efficiency by more than 40% over the traditional scheme. However, data restoration is a challenge for this storage design. Xue et al. [26] presented an AD-KP-ABE key policy attribute-based encryption technique for cloud data deletion that uses the MHT and attribute revocation primitives to accomplish fine-grained access control with verifiability. When a user wishes to delete a file, the attribute is changed by re-encrypting a portion of the ciphertext, guaranteeing that data is safely erased. This approach utilizes the MHT where the cloud server generates new roots following attribute revocation to validate target deletion. Although verification of deletion is obtainable in this system, a trustworthy third party is required.

VIII. Conclusion

In this paper, we have proposed *DRS*, a deduplication scheme in a redundant servers scenario. DRS ensures storage efficiency while keeping high availability and reliability by using our custom hashmap with convergent encryption. Our proposed algorithm can choose the best servers from a pool of maximum servers that have been assigned, allowing for user-defined redundancy while minimizing duplication. Our custom data structure, which employs a matching algorithm, can produce a server rating and also employs an index server, which also serves as a load balancer to maintain a balance between redundancy and duplication. Implementation of *DRS* demonstrates that our scheme is efficient in computation and communication overhead also. We measured our performance using four criteria (maximum servers, redundancy factor, block size, and file size), and we also compared our design to MHT, which shows that our scheme is efficient and takes less time than MHT in all scenarios. Besides, both file-level and block-level deduplication is achieved in our design while ensuring batch auditing to check the integrity of dispersed data. The security and performance analysis of our system reveals that it is effective in terms of storage efficiency as well as reliability.

References

[1] H. Yuan, X. Chen, T. Jiang, X. Zhang, Z. Yan, and Y. Xiang, "Dedupdum: secure and scalable data deduplication with dynamic user management," *Information Sciences*, vol. 456, pp. 159–173, 2018.

[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "Racs: a case for cloud storage diversity," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 229–240.

[3] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 187–198.

[4] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, "Nccloud: applying network coding for the storage repair in a cloud-of-clouds." in *FAST*, vol. 21, 2012.

[5] R. Steans, G. Krumholz, and D. Hanken-Kurtz, "Duracloud™ and flexible digital preservation at the texas digital librar," 2015.

[6] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," *Acm transactions on storage (tos)*, vol. 9, no. 4, pp. 1–33, 2013.

[7] B. Mao, S. Wu, and H. Jiang, "Improving storage availability in cloud-of-clouds with hybrid redundant data distribution," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 633–642.

[8] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage (ToS)*, vol. 7, no. 4, pp. 1–20, 2012.

[9] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data {Deduplication—Large} scale study and system design," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 285–296.

[10] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[11] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *ACM Transactions on Storage (ToS)*, vol. 8, no. 4, pp. 1–26, 2012.

[12] L. DuBois, M. Amaldas, and E. Sheppard, "Key considerations as deduplication evolves into primary storage," *White Paper*, vol. 223310, 2011.

[13] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proceedings 22nd international conference on distributed computing systems*. IEEE, 2002, pp. 617–624.

[14] N. Jayapandian and A. Md Zubair Rahman, "Secure deduplication for cloud storage using interactive message-locked encryption with convergent encryption, to reduce storage space," *Brazilian Archives of Biology and Technology*, vol. 61, 2018.

[15] X. Yang, R. Lu, J. Shao, X. Tang, and A. Ghorbani, "Achieving efficient secure deduplication with user-defined access control in cloud," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[16] S. Wu, K.-C. Li, B. Mao, and M. Liao, "Dac: Improving storage availability with deduplication-assisted cloud-of-clouds," *Future Generation Computer Systems*, vol. 74, pp. 190–198, 2017.

[17] E. Bisong, "Google colaboratory," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Apress, pp. 59–64. [Online]. Available: https://doi.org/10.1007/978-1-4842-4470-8$_7$

[18] P. Singh, N. Agarwal, and B. Raman, "Secure data deduplication using secret sharing schemes over cloud," *Future Generation Computer Systems*, vol. 88, pp. 156–167, 2018.

[19] H. Yuan, X. Chen, J. Li, T. Jiang, J. Wang, and R. H. Deng, "Secure cloud data deduplication with efficient re-encryption," *IEEE Transactions on Services Computing*, vol. 15, no. 1, pp. 442–456, 2019.

[20] S. Li, C. Xu, and Y. Zhang, "Csed: Client-side encrypted deduplication scheme based on proofs of ownership for cloud storage," *Journal of Information Security and Applications*, vol. 46, pp. 250–258, 2019.

[21] H. Yuan, X. Chen, J. Wang, J. Yuan, H. Yan, and W. Susilo, "Blockchain-based public auditing and secure deduplication with fair arbitration," *Information Sciences*, vol. 541, pp. 409–425, 2020.

[22] J. Li, J. Wu, G. Jiang, and T. Srikanthan, "Blockchain-based public auditing for big data in cloud storage," *Information Processing & Management*, vol. 57, no. 6, p. 102382, 2020.

[23] P. Neelima and A. Reddy, "An efficient load balancing system using adaptive dragonfly algorithm in cloud computing," *Cluster Computing*, vol. 23, no. 4, pp. 2891–2899, 2020.

[24] G. E. Gonçalves, P. T. Endo, M. Rodrigues, D. H. Sadok, J. Kelner, and C. Curescu, "Resource allocation based on redundancy models for high availability cloud," *Computing*, vol. 102, no. 1, pp. 43–63, 2020.

[25] S. Nannai John and T. Mirnalinee, "A novel dynamic data replication strategy to improve access efficiency of cloud storage," *Information Systems and e-Business Management*, vol. 18, no. 3, pp. 405–426, 2020.

[26] L. Xue, Y. Yu, Y. Li, M. H. Au, X. Du, and B. Yang, "Efficient attribute-based encryption with attribute revocation for assured data deletion," *Information Sciences*, vol. 479, pp. 640–650, 2019.