HFBT: An Efficient Hierarchical Fault-tolerant Method for Cloud Storage System

Ling Xiao[1,3], Beiji Zou[1,3], Chengzhang Zhu[1,2,3,*], Meng Zeng[1,3], Zhi Chen[1,3]
[1]*School of Computer Science and Engineering*, *Central South University*, Changsha, China
[2]*The College of Literature and Journalism*, *Central South University*, Changsha, China
[3]*Hunan Engineering Research Center of Machine Vision and Intelligent Medicine*, *Central South University*, Changsha, China
{194701006@csu.edu.cn, bjzou@csu.edu.cn, anandawork@126.com, zengmeng@csu.edu.cn, chen.zhi@csu.edu.cn}

***Abstract*—With the development of information technology and wide application of smart devices, data increases exponentially and more and more data are stored in the cloud, whereas most of them are infrequently accessed. Data temperature indicates the access frequency of data, where hot data is frequently accessed while cold data is rarely accessed after a while, also called archival data appearing in many fields, such as health care, finance, education etc. For these domains, appropriate fault-tolerant method for data of different temperature can improve the overall performance, so we propose the hierarchical fault-tolerant method based on temperature (HFBT) that provides the data placement strategy based on load balancing to improve application performance (The efficiency of storage and read) and recovery performance simultaneously and then executes encoding transformation according to data temperature to reduce storage overhead, and finally exploits the pipeline repair based on slice-level for archival data to decrease recovery time. This paper has three main contributions. (1) The reliability: HFBT realizes closely to 100% of the reliability. (2) The performance: HFBT saves 50% of storage overhead compared with 4-replicas and improves overall performance compared with other methods. Besides, the recovery speed of HFBT is up to 88.09% higher than that of RS (9, 6) code. (3) The flexibility: users can flexibly set the encoding transformation rules according to data characteristics.**

Keywords—archival data, fault-tolerance, data placement, encoding transformation, pipeline repair

I. INTRODUCTION

With the popularity of internet and the increasement of intelligent devices, huge amounts of data are generated, and how to efficiently store large-scale data has become a research hotspot. Nowadays, in addition to store data on local devices, cloud storage is an excellent solution that attracts extensive attentions. The traditional storage system has limited capacity, while cloud storage system can arbitrarily add nodes to expand storage space, which provides better scalability. In addition, one server of the cloud storage system can utilize resources of other servers in the data center to realize resource sharing and save costs, and users access data and resources in the cloud at any time through the network. In summary, cloud storage system has the advantages of scalability, economy and flexibility, but it is built based on multiple servers and hard disks in servers generally have low reliability, which is prone to failure [1]. For example, over 50 machines appear unavailable every day in the Facebook cluster with storage capacity of hundreds of petabytes because of hard disks failure or power failure [2]. These failures cause data not being correctly accessed, so cloud storage system is necessary for establishing fantastic fault-tolerant mechanism to make sure the reliability.

In general, replication and erasure code are the primary strategies to guarantee reliability for cloud storage system [3], and the former provides outstanding recovery performance, but incurs high storage expenses. The recovery performance refers that the data will be reconstructed by the fault-tolerant mechanism once data is lost, and the reconstruction procedure seriously affects read efficiency on users. The storage overhead of erasure code dramatically decreases when maintaining the same fault-tolerance, whereas it generates noteworthy reconstruction costs, of which Reed-Solomon code (RS) is the most popular, and the operations of encoding and decoding are completed based on Galois field (GF) [4]. Afterwards, researchers present massive optimized methods to remedy the imperfections, mainly including three categories: the optimization of RS code (OR), hybrid erasure code (HEC) and the combination of replication and erasure code (REC). The OR method focuses on how to reduce the recovery cost of single erasure code, such as the Local Recovery codes (LRC) [5] for reducing disk I/O and Regeneration codes (RGC) [6] for reducing network bandwidth. The HEC mechanism uses different structures or diverse kinds of erasure codes to balance storage overhead and recovery performance [7], [8], [9]. The REC method combines the complementary features of replication and erasure code to achieve better performance [10], [11], [12].

Nevertheless, these solutions still have some defects. The OR method doesn't consider the data access frequency and exploits static encoding that either sacrifices storage overhead or recovery performance. The HEC mechanism balances the storage overhead and recovery performance to some extent, but the processes of encoding and decoding have very high time complexity, which seriously affects the read efficiency on users, since erasure code with k data blocks and m redundancy blocks needs to read k blocks to recover the data block when a data block is lost. Similar to the HEC mechanism, the REC method considers the storage cost and recovery performance, and multiple replicas improve the access efficiency of hot data, whereas cold data stored with erasure code still has high recovery delay. Besides, for the HEC mechanism and the REC method, the partition of data temperature greatly affects the storage overhead, so how to identify the temperature of data and utilize appropriate fault-tolerant method for data of different temperature is very significant. Furthermore, current approaches concentrate on how to reduce the recovery delay, but ignore that the efficiency of storage and read is more intuitive and important to users. Generally, the recovery performance is only measured when data is lost and then reconstructed, and the main impact on users is the efficiency of storage and read. Therefore, how to improve both application performance (The efficiency of storage and read) and the recovery performance to maximize overall performance at the same time is a huge challenge, but the existing approaches don't take this into account.

Inspired by this insight, in this paper, taking medical data as research object, we propose a hierarchical fault-tolerant method based on temperature (HFBT). Medical treatment and clinical diagnosis generate lots of data which only are accessed when patients are in hospital. When the patient is discharged from hospital, the data related to the patient is rarely accessed again, that is archival data. However, these archival data need to be retained for a long time according to relevant medical rules, and it is essential to build fault-tolerance for these data in order to guarantee reliability. Different fault-tolerant methods have diverse storage cost and reconstruction overhead. If all data are used with the manner of the same fault-tolerant strategy, a large amount of storage space or recovery time will be wasted. By analyzing these characteristics of medical data and combining the application performance with recovery performance, on the basis of REC method, we present the data placement strategy based on load balancing to improve the overall performance (Storage, read and recovery) when placing data blocks. Then, we propose the flexible encoding transformation strategy that identifies the temperature of data to decrease storage overhead of archival data, and meanwhile utilize the pipeline repair based on slice-level which makes up for the drawback of erasure code with high recovery delay to speed up reconstruction of archival data. The key contributions are as follows.

(1) We analyze the access characteristics of medical data and propose an effective hierarchical fault-tolerant method for archival data that is rarely studied. Although taking medical data as research object, it also can be referenced by other fields, such as education, finance, etc.

(2) On the basis of the REC method, HFBT considers application performance and recovery performance simultaneously, which effectively improves the efficiency of storage, read and recovery.

(3) The flexible encoding transformation reduces storage overhead and the pipeline repair for archival data makes up for the defect of erasure code with high recovery delay while taking advantage of the benefits of replication and erasure code.

The remainder of this paper is organized as follows. In Section II, the characteristics of medical data and related works are introduced. In Section III, our methodology is described in detail. Comprehensive experiments are carried out in Section IV and this paper is concluded in Section V.

II. Background and Related Works

This section analyses characteristics of medical data, and then introduces related works of storing archival data.

A. *Characteristics of Medical Data*

In 2017, Stanford Medicine Health Trends Report predicted that the amount of medical health data will reach over 2,314 EB by 2020 [13], for this reason that on the one hand, hospitals, clinical experiments, pharmaceutical analyses and medical smart wearables devices generate a mass of medical data, on the other hand, and medical data need to be retained for a long time to provide more complete and accurate services [14], [15]. Nevertheless, certain medical data is requisite for keeping online, and most of them is timeliness, which is rarely accessed after a while. For instance, medical expenses, inspection reports, radiotherapy records, CT images, MRI images are almost no longer accessed when patients are discharged from hospital, which is called archival data in this paper.

The preservation of archival data is the very meaningful, so different countries formulate diverse proposals for archival data, whereas there has one thing in common that archival medical data is conserved for a long time, as illustrated in the TABLE I.

TABLE I. Summary of Archival Data

Country	Date	Organization	Relevant proposals
USA	2000	American Society for Radiation Oncology [16]	All radiotherapy-related records need to be kept for at least 5 years after death.
Australia	2005	The Royal Australian and New Zealand College of Radiologists & The Faculty of Oncology [17]	Records and prescriptions need to be preserved throughout the lifetime, preferably up to 5 years after death, and images are reserved for 7 years.
UK	2016	National Health Service [18]	Medical data must be conserved for 30-year or 8-year after death.
China	2017	The Ministry of Health of the People's Republic of China [19]	The records of outpatient and inpatient need to be preserved over 15 years, 30 year respectively.

Great importance is attached to the preservation of medical data, where archival data is infrequently accessed, but takes up a large proportion of storage space. Therefore, different fault-tolerant mechanisms are requisite for online data and archival data in order to decrease recovery time and storage space. Meanwhile, it is important to periodically switch the fault-tolerance of archival data to one that requires less storage overhead.

B. *Related Works*

Researchers rarely focus on archival data to improve the overall performance of storage system. Gupta et al. [20] discovered that data need to be frequently moved to archival warehouse by analyzing structural data, and proposed the proactive archiving solution to accelerate query performance, without considering storage overhead and recovery performance. Chen et al. [21] presented parallel archiving method to reduce archiving time in view of the high cost of converting three-replicas into RS code, whereas it doesn't take into account the access frequency of data. Considering storage overhead, recovery performance and access frequency, Xia et al. [7] proposed HACFS that uses two different structures of erasure codes to dynamically adapt workload changes. Wang et al. [8] utilized LRC code and Hitchhiker code for data of different temperatures. In the process of encoding conversion, some original blocks are utilized to the maximum extent, which reduces the number of transferring blocks and the reconstruction time. Qiu et al. [9] employed RS code for writing-intensive tasks and Minimum Storage Regenerating (MSR) code for reading-intensive or frequently reconstructed tasks to balance the overhead between storage and recovery, but the calculation of MSR is very complicated. Literatures [7], [8], [9] employ different structures or diverse categories of erasure codes, which takes up lots of CPU, IO resources when encoding and decoding, and the performance of hot data is very poor. Besides, if all data are stored with erasure code, the recovery delay of data seriously affects the read efficiency on users. Therefore, Ma et al. [10] presented CAROM that combines replication and erasure code where files are stored by erasure code in backup data center. When a request initiates a write operation on the file, if the requested file isn't cached, the data is reconstructed by erasure code and then the reconstructed data is cached to master data center. When the file is cached, operations of read and write are directly oriented to replicas. Mao et al. [11] put forward HyRD that utilizes erasure code for large files and replication for small files, metadata, large files frequently accessed, which achieves rapid response and effective storage. Through discrete event simulation, Gribaudo et al. proved [12] that the combination of replication and erasure code can reduce the overhead and improve the reliability to the greatest extent. However, literatures [10], [11], [12] utilize erasure code for data accessed infrequently or large files, so the recovery delay is very high. Besides, the above methods don't describe the partition of data temperature in detail, and also don't consider the application performance of practical storage system.

Different from the above approaches, based on the REC method, we consider the application performance and recovery performance simultaneously, and propose the data placement strategy based on load balancing to improve the efficiency of storage, access, and recovery. Furthermore, the flexible encoding transformation method decreases storage overhead, and the pipeline repair based on slice-level speeds up reconstruction of archival data, which makes up for the defect of erasure code with high recovery delay while taking advantage of the benefits of replication and erasure code. To our knowledge, none of current approaches combine the pipeline repair and data temperature to adapt different workloads, and most of them sacrifice the storage overhead or recovery performance.

III. Our Methodology

In this section, we briefly introduce our motivation, then elaborate each module of HFBT, and finally describe the design architecture.

A. *Our Motivation*

The performance of applications and the storage overhead of system are affected by fault-tolerant ways, and it is necessary to establish reliable mechanisms for data, so we propose an effective fault-tolerant method HFBT to maximize the system performance and reduce the storage overhead. The framework is shown in the Fig. 1 that consists of three modules (Data placement, Encoding transformation and Pipeline repair). We mainly face the following challenges.

(1) Diverse nodes have diverse hardware configurations, different performance and various workloads. How to take full advantage of the resources of every node and evenly distribute data to obtain the maximum application performance is a significant problem.

(2) Data in different periods has different access frequency, particularly for archival data which is almost no longer accessed after a period of time, but it takes up a large proportion of space. If all data always are stored by the same fault-tolerant method, there is no doubt that lots of storage spaces or recovery time will be wasted. Therefore, we need to design an encoding transformation mechanism that periodically changes the fault-tolerant way of archival data with the manner of a low storage overhead.

(3) The HEC method and the REC method with erasure code reduce storage overhead but lead to high recovery delay and affect the read efficiency on users. Some methods focus to use erasure code with higher storage overhead to obtain less recovery delay since different erasure codes have different storage costs, which fails to obtain the optimal storage overhead and recovery performance. However, it is very important to minimize storage costs while reducing the recovery delay as much as possible.

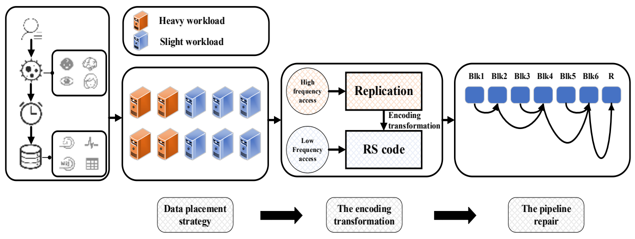


Fig. 1. The framework of HFBT.

B. *Data Placement Strategy Based on Load Balancing*

In this section, we will analyze the block placement problems caused by replication and RS code, and then introduce the data placement strategy based on load balancing.

1) *Problem analysis*

Cloud storage system generally uses replication and erasure code for fault-tolerance, where each block needs to be stored in different nodes. Especially for erasure code, once data is lost, the data needs to be reconstructed, which consumes lots of CPU, IO, memory, and results in poor application performance. For a cluster containing 12 Datanodes, 3-replicas will randomly select 3 nodes, and RS (9, 6) code will optionally choose 9 nodes (This paper mainly discusses inter-node fault-tolerance rather than rack-level). As shown in Fig. 2, there are 12 nodes where nodes n_i, n_j, and n_k have numerous calculation tasks and other nodes have slight workload. If 3-replicas or RS code selects nodes n_i,

n_j, and n_k, performance of the system will be terrible. Next, we will discuss the cost of choosing different nodes.

Assuming l data blocks $\{b_1, b_2, \ldots, b_l\}$, m nodes $\{n_1, n_2, \ldots, n_m\}$, the block size b_{size}, R_j, W_j, Y_j is the frequency of reading, writing, and recovering of n_j. FR_j, FW_j and FT is the speed of reading and writing, the seeking time of n_j. CR_j, CW_j, CY_j is the cost of reading, writing, recovering of n_j.

The cost of writing a block b_k ($k \in \{1, l\}$) and reading the b_k from node n_j ($j \in \{1, m\}$) is:

$$CW_{jk} = \frac{b_{size}}{FW_{jk}} + FT \tag{1}$$

$$CR_{jk} = \frac{b_{size}}{FR_{jk}} + FT \tag{2}$$

For RS code with d data blocks and r redundancy blocks, the cost of recovering the b_k is shown (3), where TR is the time of transferring a block from a node to another node (Assuming all nodes have the same bandwidth).

$$CY_{jk} = d(\frac{b_{size}}{FR_{jk}} + FT + TR) \tag{3}$$

The total cost of data access is:

$$CA_j = CR_{jk} \times R_{jk} + CW_{jk} \times W_{jk} + CY_{jk} \times Y_{jk} \tag{4}$$

Assuming that nodes are accessed with the same frequency and the seeking time. The cost comparison between node n_i and node n_j is:

$$CA_j - CA_i = (R + Yd)(\frac{b_{size}}{FR_{jk}} - \frac{b_{size}}{FR_{ik}}) + W(\frac{b_{size}}{FW_{jk}} - \frac{b_{size}}{FW_{ik}}) \tag{5}$$

When a node has a faster rate of reading and writing, the total cost of reading, writing, and recovering for the node is smaller.

2) Data placement strategy

The speed of reading and writing is mainly affected by the resources of node, such as CPU, memory and IO. However, the Hadoop [22] with random placement (RP) strategy doesn't fully consider the heterogeneity of nodes, which leads to the load imbalance of the system. Therefore, we put forward the data placement strategy based on load balancing. When placing blocks, we consider the resource utilization of the system and then select nodes with light workload. The workload of node is measured by the following indicators.

Memory utilization is the ratio of the memory M_u used by the process to the total memory M_t. The memory utilization M is denoted by (6).

$$M = \begin{cases} \frac{M_u}{M_t}, M_u < M_t \\ \infty, M_u = M_t \end{cases} \tag{6}$$

CPU utilization is the ratio of the CPU execution time C_u of the process to the total time C_t, which represents the running situation of the process in a certain period, the lower the CPU utilization, the less the running process. The calculation is shown in (7).

$$C = \begin{cases} \frac{C_u}{C_t}, C_u < C_t \\ \infty, C_u = C_t \end{cases} \tag{7}$$

I/O utilization is the ratio of the actual transfer rate I_u of the disk to the maximum transfer rate I_t provided by the parameter of the disk, as shown in (8).

$$I = \begin{cases} \frac{I_u}{I_t}, I_u < I_t \\ \infty, I_u = I_t \end{cases} \tag{8}$$

On the premise of considering above indicators, if all data are allocated to a node, the node network will be blocked and the reading efficiency will be affected. In order to make data evenly distributed among nodes, occupied storage space should also be considered. Storage occupancy S is the ratio of the storage space S_u used by node to the total storage capacity S_t, the calculation shown in (9).

$$S = \begin{cases} \frac{S_u}{S_t}, S_u < S_t \\ \infty, S_u = S_t \end{cases} \tag{9}$$

In order to improve the utilization of resources, blocks should be placed on nodes with slight workload. The evaluation function is constructed according to these indicators, and the workload of node is evaluated by (10). The evaluation function is based on the linear weighting method [23] that obtains the corresponding weights in accordance with the importance of each indicator. η_i represents the weight coefficient of each indicator where the larger η_i indicates the more significant impact on the node performance.

$$W = \eta_1 M + \eta_2 C + \eta_3 I + \eta_4 S \tag{10}$$

$$\sum_{i=1}^{4} \eta_i = 1 \tag{11}$$

Weight coefficients are determined through the influence of indicators on the node performance. Due to node performance is mainly influenced by the memory, CPU, I/O, and storage space, and these factors belong to different categories. Hence, the corresponding weights are determined according to the hierarchy analytic method [24] that is a hierarchical weight decision combining quantitative and qualitative analyses. The evaluation objective is decomposed, and the complex problem is divided into related factors in accordance with these evaluation indicators. Then, quantitative decision is made on the importance of each indicator through the combination of objective analyses and subjective judgments. Therefore, the value of η_i will be given in combination with experimental analyses in Section IV.

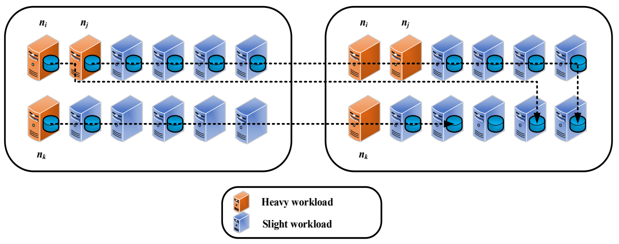


Fig. 2. The data placement strategy.

C. The Flexible Encoding Transformation of Replication and Erasure Code

In this section, we analyze the performance of replication and erasure code, and then describe in detail the encoding transformation according to the access rules of medical data.

1) The performance analyses of replication and erasure code

- The definition of erasure code

Replication and RS code are widely used for fault-tolerance in distributed file system, such as Hadoop Distributed File System (HDFS) [22] and Google File System (GFS) [25]. Generally, replication strategy copies n replicas on n different nodes to tolerate n failures, while RS code performs more complicated calculation based on Galois field to maintain the same fault-tolerance. RS code is defined with (n, k) where n and k are the number of total blocks and data blocks respectively, ($k > n$-k). The (n, k) indicates that a file is divided into k data blocks and n-k parity blocks. The process of encoding and decoding of RS code is as follows.

The definition of encoding: Given k data blocks (b_1, b_2, …, b_k) and the positive integer m (m=n-k), RS code can generate m parity blocks according to the k data blocks, as shown in (12) where each parity block can be represented by linear combinations of data blocks. Among $G_{ij} \in F_q$, $i \in (1, m)$, $j \in (1, k)$, G_{ij}, D_j and P_i are the coefficient of generator matrix, the data block, and the parity block, respectively, and F_q is the GF field with q elements. In order to ensure the fault-tolerance between nodes, the $k + m$ blocks are stored in different nodes.

$$P_i = (G_{i1}, G_{i2}, G_{i3}, ..., G_{ik}) \times (D_1, D_2, D_3, ..., D_k)^T \quad (12)$$

The definition of decoding: Given discretionary k and m, the original data can be generated according to k data blocks and m parity blocks. That is, RS code can tolerate m data block failures at most, as shown in (13), of which D is the missing data block, and λ_i , R_i are the coefficient of the decoding matrix and remainder blocks, $\lambda_i \in F_q$.

$$D = \sum_{i=1}^{k} \lambda_i \times R_i \quad (13)$$

As shown in Fig. 3 and Fig. 4, there are six data blocks and three parity blocks, and the parity block P is obtained by the generator matrix G multiplying the data block D. If the data blocks D_2, D_3, and D_5 lose, decoding algorithm inverses the encoding matrix, and then multiplies by remaining k blocks, which restores the lost data blocks.

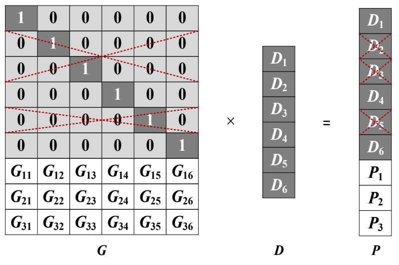


Fig. 3. The encoding of RS (9, 6) code.

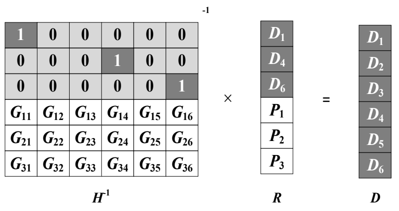


Fig. 4. The decoding of RS (9, 6) code.

- The comparison of storage overhead and recovery performance

The multiple replicas of replication occupy a mass of storage space. RS code economizes storage overhead, whereas the encoding and decoding is very complex, which leads to high recovery delay of data reconstruction. When the file size is S and the block size is S/k (n nodes, k data blocks), the comparisons of storage overhead and recovery performance of RS code and replication are shown in TABLE II.

TABLE II. THE STORAGE OVERHEAD AND RECOVERY PERFORMANCE

Method	**Fault-tolerance**	**Storage costs**	**Bandwidth of recovering a block**
RS code	n-k	nS/k	S
Replication	n-k	$(n$-k+1$)S$	S/k

It is difficult to balance storage overhead and recovery time. Hot data is frequently accessed and needs low recovery delay, while archival data occupies a large proportion storage space and has lower requirements on read performance, so replication and RS code applied for hot data and archival data can achieve better performance. However, the number of replicas and the parameters of RS code are also important to maximize reliability and reduce storage overhead.

- The analysis of reliability

When the cloud storage system has n nodes, the reliability of each node is r and the redundancy factor is d (d is the physical space size divided by logical space size of data block). The larger d indicates more storage overhead.

The system reliability P_{re} with replication is shown as (14), where d is the number of replicas.

$$P_{re} = 1 - (1-r)^d \quad (14)$$

The system reliability P_{rs} with RS (n, k) code is shown in (15), where k and n are the number of data blocks and total blocks, $d=n/k$.

$$P_{rs} = \sum_{i=0}^{kd-k} C_{kd}^{i} . r^{kd-i} . (1-r)^{i} \quad (15)$$

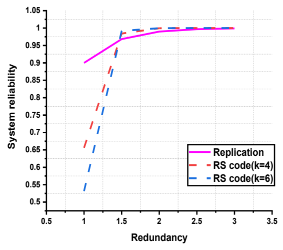


Fig. 5. The comparison of reliability.

The comparison of system reliability is shown in Fig. 5, where the x-axis is the redundancy factor d (r=0.9). When $d \geq 1.5$, the reliability of RS code is higher than that of the replication and closely to 100%. Moreover, with the increasement of the number of nodes, the reliability of RS code is higher. The reliability of replication is closely to 100% until $d \geq 3$. Therefore, 3-replicas (d=3) for hot data and RS (9, 6) code (d=1.5) for archival data can acquire closely 100% reliability and mitigate hot issues of data.

2) *The access rules of medical data*

For medical system, the majority of data access cycle is relatively short. When patients are discharge from the hospital after recovery or outpatient treatment, the previous data are scarcely accessed. Relevant statistical analyses reveal that data access accords with the Pareto distribution, also called the 80/20 rule [26], [27], which describes plentiful real-world phenomena, the probability density function as shown in (16), where x and x_m respectively are a random variable, the smallest positive number of x, $\alpha > 0$.

The analysis means that 20% data is accessed frequently, and 80% data is rarely accessed after a certain period. Besides, we collected the medical data from the cooperative eye hospital and the access characteristics also conform to this distribution. To maximize the overall performance, rarely accessed data should use the fault-tolerant approach with low storage overhead, and frequently accessed data should use the fault-tolerant method with superior recovery performance. Although data access generally conforms to the Pareto distribution, the temperature of data will change with the increase of data. The recovery performance of different fault-tolerant methods is inversely proportional to the storage cost, so the encoding transformation of data is very important to balance the storage overhead and recovery cost of the fault-tolerant system.

$$f(x) = \begin{cases} 0, & x < x_m \\ \alpha \dfrac{x_m^{\alpha}}{x^{\alpha+1}}, & x \geq x_m \end{cases} \quad (16)$$

3) *The encoding transformation*

The encoding transformation need consider the temperature variation and design the switching rule. Since archival data is almost never converted to hot data, the temperature variation only considers the cooling situation of hot data. When the amount of hot data is more than w_{hc} of total capacity, the fault-tolerant mechanism of cooling hot data is switched to RS code. w_{hc} and 1-w_{hc} are the ratio of hot data and archival data (Hot data and archived data are stored by replication and erasure code, respectively). Literature [26] demonstrates that 30% data is divided into hot data, which provides the best solution for cloud storage system. Based on Pareto distribution and [26], the subsequent experiments set the w_{hc} as 20% because the pipeline repair based on slice-level is leveraged to speed up the reconstruction of archival data, which makes up for the defect of RS code with high recovery delay. In practical scenarios, users can flexibly set w_{hc} according to data characteristics. Even if the proportion of hot data exceeds 20% at some time, this situation is temporary, moreover, and the failure of data will not happen all the time, so data keeps this distribution within the average range.

For the switching rule, common switching ideas are FIFO (First in first out), LRU (Least recently used), and LFU (Least frequently used) [28]. FIFO and LRU only consider the locality of time and don't combine with data access frequency, which is unsuitable for practical storage system. LFU describes the access frequency of data and switches data based on historical access frequency, which is reasonable for practical application scenarios because data recently accessed has greater probability of being accessed in the future. Hence, we propose the encoding transformation method based on time-slice LFU idea that switches the fault-tolerant ways where the time is divided into fine-grained slices to count more accurate access frequency, and the access frequency is counted in each time slice. Besides, the access frequency of file is not simply counted, but larger weight is given to the most recently accessed file based on LFU idea. The pseudocode of the encoding transformation is shown in algorithm 1, and definitions of relevant parameters and the calculation of access frequency of files are as follows.

- The total time $T = \bigcup_{j=1}^{t} \{T_j\}$, t represents the number of time slices.
- Data set $D = \bigcup_{i=1}^{z} \{D_i\}$, $A = \bigcup_{i=1}^{z} \bigcup_{j=1}^{t} \{A_{ij}\}$, z and A_{ij} represent the number of files, the total accesses number of file D_i in time slice j respectively.
- File size $S = \bigcup_{i=1}^{z} \{S_i\}$, S_i represents the size of file D_i.
- The accesses number of D_i in time slice j:

$$I_{ij} = A_{ij} - A_{ij-1} \quad (17)$$

- On account of the data recently accessed with greater probability of being accessed, the access frequency of file is not simply counted. The files recently accessed are given larger weight, the access frequency $f(D_i)$ of file D_i shown as (18), where P_T is the time slice.

$$f(D_i) = \sum_{j=1}^{P} (I_{ij} . 2^{-(P_r - j + 1)}) \quad (18)$$

Algorithm 1 The encoding transformation

```
Input: sets $T$, $D$, $A$, $S$, $w_{hc}$
1: set $w_1$, $S_{\text{total1}}$ ($S_{\text{total1}}$ is the total size of all files), $f_{t1}$=0, $i_{\text{index2}}$
2: if in the $j$-th time slice the file $D_i$ is accessed then
3:     the access frequency $f(D_i)$ is calculated according to (18)
4: end if
5: if in the $j$-th time slice the file $D_i$ is inserted and the size of $D_i$ is $S_i$ then
6:     sort files $D$ by access frequency and obtain descending sequence $\{f_1, f_2, \ldots, f_z\}$
7:     $f(D_i)=f_1$.
8:     set the fault-tolerance of $D_i$ by replication
9: end if
10: count new access frequency and obtain descending sequence $\{f_1', f_2', \ldots, f_{z+1}'\}$
11: $w_1= w_{hc} \times (S_{\text{total1}}+ S_i)$
12: for $i$=1 → $z$+1 do
13:     $f_{t1}=f_{t1}+f_i'$
14:     if $f_{t1} > w_1$ then
15:         set $i_{\text{index2}}=i$
16:     end if
17: end for
18: the fault-tolerance of files between $i_{\text{index1}}$ and $i_{\text{index2}}$ is converted to RS code ($i_{\text{index1}}$ is the separation index value before the data is accessed or inserted)
```

Through the switching method, only the fault-tolerant mode of cooling hot data needs to be changed according to the data access characteristics, unlike other methods ([7], [8], [9]) to perform the encoding transformation of all files, which greatly reduces the time complexity and computation complexity. The differences between HFBT and HyRD are that HyRD only uses erasure code for large files and replication for small files and metadata, and doesn't consider the data block placement of affecting application performance, the temperature of data and the encoding transformation caused by temperature change. In addition, HyRD utilizes replication for metadata to reduce access delay. However, metadata is stored in memory, which takes up more memory overhead, so this paper employs pipeline repair based on slice-level to reduce recovery delay without increasing storage overhead.

D. The Pipeline Repair Based on Slice-level

Traditional recovery method of HDFS transfers k blocks to requestor R, which causes the congested downlink of R. The pipeline method makes full use of the bandwidth of each node and transmits data blocks in parallel. However, none of current approaches combine the pipeline repair to adapt different temperature of data, and most of them sacrifice the expense of storage space or recovery time, so we propose the pipeline repair based on slice-level (PRBS) to accelerate the recovery of archival data without increasing storage overhead and make better use of the advantages of the combination of replication and erasure code to adapt to different temperature, which reduces the reconstruction time to the same as the normal read.

PRBS divides each block into smaller slices to make better utilize bandwidth resources. If there are k data blocks $\{b_1, b_2, \ldots, b_k\}$ and each block b_i ($1<i<k$) is divided into s slices $\{b_{i1}, b_{i2}, \ldots, b_{is}\}$, the transfer time of b_i is 1 timeslot and the time of transferring a slice b_{ij} ($1<j<s$) is $1/s$ timeslots. There have s paths that transmit s slices concurrently, whereas the bandwidth resource of each path doesn't conflict with each other. Each path has k blocks and requestor R, so the transfer time of every path is ks timeslots. For the first path $N_1 \to N_2 \to \ldots \to N_k \to R$, N_1 transmits $\beta_1 b_{11}$ to N_2, and N_2 combines $\beta_1 b_{11}+\beta_2 b_{21}$ to N_3, and then the process is repeated until N_k sends the first slice of all the blocks to R, that is $\beta_1 b_{11}+\beta_2 b_{21}+\ldots+\beta_k b_{k1} \to R$, where β_i is the coefficient of the decoding matrix. Therefore, the total time of recovering a block is $(s-1)/s+k/s=1+(k-1)/s$. In general, k is fixed, but s is elastic, and the total time is 1 timeslot when s is enough large. As shown in Fig. 6, the total time of recovering a block is 7/4 timeslots, which is 9/4 timeslots less than traditional recovery of HDFS.

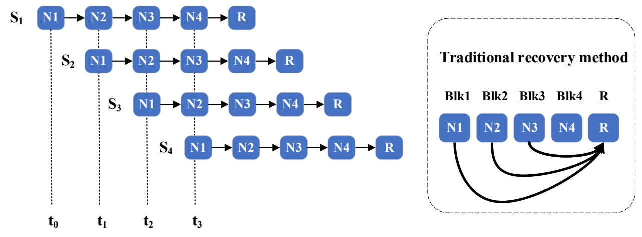


Fig. 6. The pipeline repair based on slice-level.

E. The Design Architecture

HFBT is designed for the application layer and storage layer, as shown in Fig. 7, where the application layer identifies data temperature, selects fault-tolerant methods and executes the encoding transformation according to the data access frequency. The storage layer singles out the appropriate Datanode based on the data placement strategy, and then leverages PRBS to accelerate recovery of archival data when data blocks are lost.

The data placement strategy calculates machine workloads and selects nodes with slight workload before storing blocks. PRBS mainly includes coordinator and helper where coordinator is responsible for recovery, and helper runs on each Datanode, which takes charge of dividing slices and transferring blocks. If a block is lost, the storage layer creates Requestor that transfers block id to Coordinator. Afterwards, Coordinator requests Namenode to obtain k data block for the same stripe, and then informs all helpers about the location of each block. Eventually, helpers divide blocks into slices and transfer these slices from Datanode to Requestor.

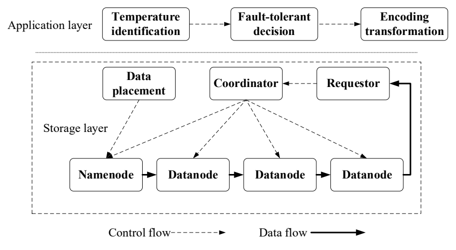


Fig. 7. The design architecture.

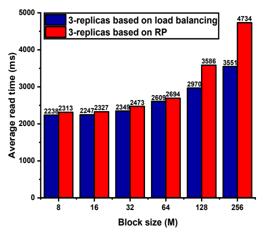
IV. Experimental Analyses

The prototype is deployed on Hadoop3.1.1 and each node is configured with Ubuntu16.04 system of Intel-Xeon-CPU@3.0GHZ, 4G memory, gigabit network, and 200G disk.

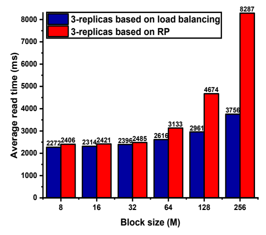
HFBT is evaluated in four main aspects: (1) Replication using the data placement based on load balancing tests different sizes of blocks compared with RP of HDFS. (2) RS code using the data placement of based on load balancing tests different sizes of blocks compared with RP of HDFS. (3) The different size of blocks and slices on pipeline repair in order to determine the best parameters. (4) The performance of HFBT is compared with that of 4-replicas, RS (9, 6) code, and the HyRD method while maintaining the same fault-tolerance, where access logs are obtained according to the PostMark benchmark [29] that creates a pool of files to generate random sized files, including text files and image files etc., with files ranging from 1 KB to 100M.

A. Replication Using the Data Placement Strategy Based on Load Balancing

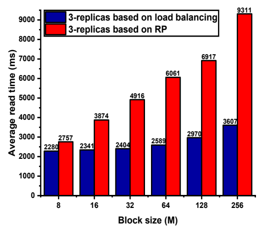
The performance of node is affected by CPU, memory, and IO, so we change one of three factors, keep the other two unchanged, and test the influence of these factors on the Datanode when the fault-tolerance is 3-replicas. As shown in Fig. 8, the cluster has 10 nodes with 1 Namenode and 9 Datanodes (node1-node9), where (a), (b) and (c) set 3 nodes with 50% CPU workload, 50% memory workload, and 50% IO workload respectively. As can be seen from the figure, 3-replicas using the data placement strategy based on load balancing achieves better performance than RP of HDFS.



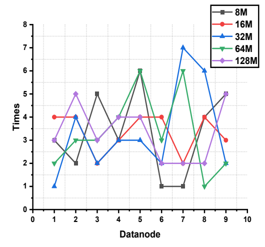
(a) CPU workload.



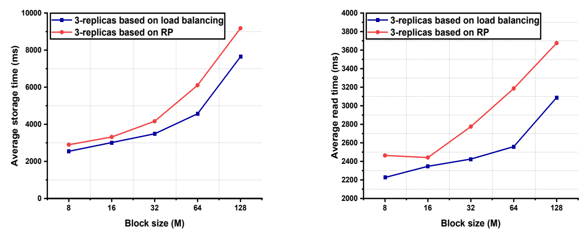
(b) Memory workload.



(c) IO workload.



(d) The replicas distribution of RP.



(e) Hybrid workloads.

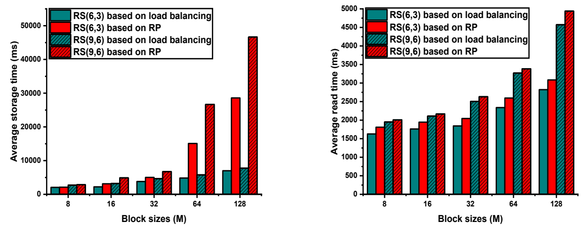
Fig. 8. The comparison between replication based on load balancing and RP.

In general, users more concern read performance, and thus we measure the weight of these indicators according to read performance. When the size of data block is larger, the performance of 3-replicas based on load balancing is approximately 1:2:3 better than that of RP, as shown in (a), (b) and (c). For example, when the size of data block is 128M, the 3-replicas based on load balancing reduces the read time than RP by 17.18%, 36.65%, and 57.06% under 50% CPU workload, 50% memory workload, and 50% IO workload, respectively, where the average results are the average time of 10 tests. Consequently, the ratios of η_1, η_2, η_3 are set to be 2:1:3 according to the hierarchy analytic method. Besides, since the storage space has feeble influence, whereas data need to be evenly distributed across each node, the ratio of η_4 is set to be equal 1. Eventually, the value of η_1, η_2, η_3, η_4 are 2/7, 1/7, 3/7, 1/7.

After determining the weights of indicators, we conduct the experiment of hybrid workloads, as shown in (d), (e), node 1 and node 2 with 50% CPU workload, node 3 and node 4 with 50% IO workload, node 5 and node 6 with 50% memory workload, node 7 with 50% CPU, 50% IO and 50% memory workloads, node 8 and node 9 without workload. As shown in (d), 10 tests are executed with 3-replicas of RP under different sizes blocks and per block of different size produces 30 distributions. The x-axis and y-axis stand for the Datanode and the distribution times of RP replicas on nodes respectively, which indicates that the block placement strategy of RP is random. Our method selects node 1, node 8, node 9 or node 2, node 8, node 9 with slight workloads. The (e) shows that the storage performance and read performance of 3-replicas based on load balancing are superior to RP under hybrid workloads.

B. RS Code Using the Data Placement Strategy Based on Load Balancing

We experiment on RS (6, 3) code and RS (9, 6) code in 10-node cluster with 1 Namenode and 9 Datanodes and 13-node cluster with 1 Namenode and 12 Datanodes, where node 2, node 4, and node 6 occupy 50% CPU workload, 50% memory workload, and 50% IO workload respectively. As shown in Fig. 9, the storage performance, read performance and recovery performance of RS (6, 3) code and RS (9, 6) code based on load balancing are better than those of RP. When the block size is 64M and 128M, the recovery time of RS (9, 6) code based on load balancing decreases by 21.38%, 37.52%, 53.82% and 24.08%, 33.97%, 49.71% compared with RP in the case of one block loss, two block losses and three block losses respectively since workloads on nodes consume lots of resources and affect the decoding process.



(a) The time of storage and read under hybrid workloads.

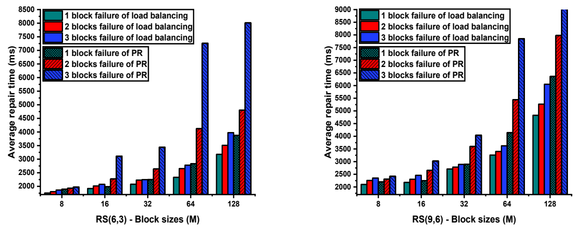

(b) Recovery time under hybrid workloads.

Fig. 9. The comparison between RS code based on load balancing and RP.

C. The Pipeline Repair Based on Slice-level

The pipeline repair based on slice-level (PRBS) is utilized to accelerate the reconstruction of archival data. We compare the recovery performance of PRBS with different sizes of slices and blocks, as shown in Fig. 10. As can be seen from the left figure, when the block is fixed at 64M, the recovery time gradually decreases with the increasement of slice size since the number of slices is larger when the size of slice is smaller, which increases the time of dividing slices. Furthermore, PRBS achieves better performance than HDFS when one block, two blocks and three blocks are lost. When the slice size is greater than or equal to 256K, the slope gradually decreases and tends to be flat. Therefore, the size of slice is selected as 256K in subsequent experiments.

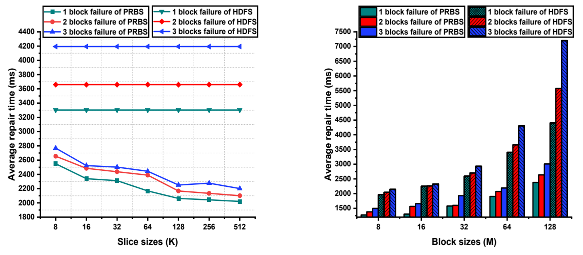


Fig. 10. Different sizes of slices and blocks.

When the size of slice is fixed at 256K, we compare the recovery performance of different block sizes, as shown in the right of Fig. 10. While one block, two blocks and three blocks are lost, the recovery time of PRBS is smaller than that of HDFS, and when the block size is 64M and 128M, the recovery time of PRBS reduces by 36.6%, 43.31%, 49.09% and 45.94%, 52.74%, 58.24% compared with HDFS. Because our file is less than or equal to 100M, the subsequent block size is set as 64M.

D. Performance Analyses of HFBT

HFBT employs different strategies for fault-tolerance and carries out the encoding transformation based on data temperature, which realizes the equilibrium of the storage overhead and recovery time, and further utilizes the block placement strategy based on load balancing to improve the overall performance and accelerates the recovery of archival data by leveraging PRBS. As shown in the left of Fig. 11, we compare HFBT (4-replicas for hot data and RS (9, 6) code for archival data), 4-replicas, RS (9, 6) code and HyRD (4-replicas for small files and metadata, RS (9, 6) code for large files) when maintaining the same fault-tolerance. When 1 node, 2 nodes and 3 nodes fail, the recovery speed of HFBT is 37.54%, 43.72% and 50.03% higher than that of HyRD, furthermore, and the recovery speed of HFBT is 69.92%, 73.57%, 88.09% higher than that of RS (9, 6) code. Although the recovery speed of HFBT is not as fast as that of 4-replicas, its storage overhead is half of 4-replicas, which saves 50% storage space. We also compare the recovery speed of HFBT based on load balancing with HyRD based on RP, when 3 nodes of 12 Datanodes are set to 50% CPU workload, 50% memory workload, and 50% IO workload respectively, as shown in the right of Fig. 11. The recovery speed of HFBT based on load balancing is 1.75X, 1.97X, and 1.98X of the HyRD based on RP when 1 node, 2 nodes and 3 nodes fail, which demonstrates that the hierarchical fault-tolerant method combined with the data placement strategy based on load balancing achieves better performance.

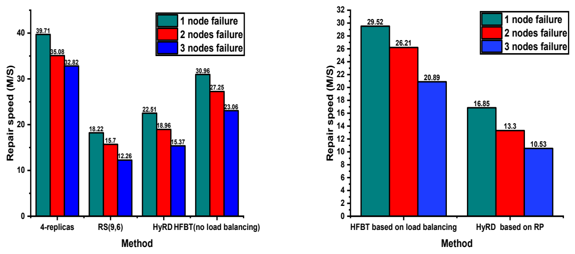


Fig. 11. Comparison of HFBT with other methods.

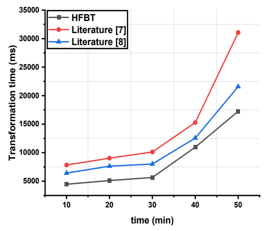


Fig. 12. The time of encoding transformation.

As temperature changes with time and data access, we simulate data access. 200M, 300M, 400M, 500M and 600M data will be inserted every 10 minutes and 20% of hot data will be accessed. The encoding transformation time between HFBT and other methods is shown in Figure 12 (There is no encoding conversion for HyRD since the encoding of HyRD is based on file size and data type) where HFBT only converts encoding of part data and its transformation time is minimal. Therefore, when the encoding transformation occurs, there has weeny impact on users, but saves massive storage overhead.

V. Conclusion and Future Work

The failure of large-scale cloud storage system is frequent, whereas replication and erasure code incur great storage overhead and recovery cost respectively. How to balance storage overhead and recovery time to achieve maximum performance is a research hotspot of cloud storage system and cloud service center. Although there are lots of improved approaches, most of them don't simultaneously combine application performance with recovery performance and sacrifice the storage cost when reducing the recovery delay. In addition, the characteristics of archival data aren't considered. Therefore, we propose HFBT which improves the performance of storage, read and recovery by the data placement strategy based on load balancing, and the flexible encoding transformation method combined with the characteristics of archival data access decreases storage overhead. Moreover, the pipeline repair based on slice-level combined with data temperature is firstly proposed to speed up the reconstruction of archival data without increasing the storage cost and make up for the defect of RS code with high recovery delay. Although HFBT maximizes overall performance and reduces storage overhead, we don't consider fault-tolerance of rack-level. The transmission consumption of across rack is conspicuous, so we will combine HFBT with fault-tolerance of rack-level, and further research how to optimize the overhead of across rack in the future.

ACKNOWLEDGMENT

This research is supported by the National Key R&D Program of China (No.2018AAA0102100)，the Fundamental Research Funds for the Central Universities of Central South University (No.2020zzts143), the Scientific and Technological Innovation Leading Plan of High-tech Industry of Hunan Province (No.2020GK2021).

REFERENCES

[1] W. Dai, L. Qiu, A. Wu, and M. Qiu, "Cloud Infrastructure Resource Allocation for Big Data Applications," IEEE Transactions on Big Data, vol. 4, no. 3, pp. 313-324, 2018.

[2] K. V. Rashmi, Nihar. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluste," USENIX Workshop on Hot Topics in Storage and File Systems, 2013, pp. 1-8.

[3] C. Aatish and M. Avinash, "Data Management in Erasure-Coded Distributed Storage Systems," IEEE International Symposium on Cluster, Cloud and Internet Computing, 2020, pp. 902-907.

[4] I. Tamo, M. Ye, and A. Barg, "The Recovery Problem for Reed–Solomon Codes: Optimal Recovery of Single and Multiple Erasures With Almost Optimal Node Size," IEEE Transactions on Information Theory, vol. 65, no. 5, pp. 2673-2695, 2019.

[5] L. Ma and C. Xing, "Constructive Asymptotic Bounds of Locally Recoveryable Codes via Function Fields," IEEE Transactions on Information Theory, vol. 66, no. 9, pp. 5395-5403, 2020.

[6] Q. Q. Xu, W. Y. Xi, K. L. Yong, and C. Jin, "CRL: Efficient Concurrent Regeneration Codes with Local Reconstruction in Geo-Distributed Storage Systems," Journal of Computer Science and Technology, vol. 33, no. 6, pp. 1140-1151, 2018.

[7] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A Tale of Two Erasure Codes in HDFS," USENIX Conference on File and Storage Technologies, 2015, pp. 213-226.

[8] Z. Wang, H. Wang, A. Shao, and D. Wang, "An Adaptive Erasure-Coded Storage Scheme with an Efficient Code-Switching Algorithm," International Conference on Parallel Processing, 2020, pp. 1-11.

[9] H. Qiu, C. Wu, J. Li, M. Guo, T. Liu, X. He, Y. Dong, and Y. Zhao, "EC-Fusion: An Efficient Hybrid Erasure Coding Framework to Improve Both Application and Recovery Performance in Cloud Storage Systems," IEEE International Parallel and Distributed Processing Symposium, 2020, pp. 191-201.

[10] Y. Ma, T. Nandagopal, K. P. N. Puttaswamy, and S. Banerjee, "An ensemble of replication and erasure codes for cloud file systems," IEEE International Conference on Computer Communications, 2013, pp. 1276-1284.

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

[12] M. Gribaudo, M. Iacono, and D. Manini, "Improving reliability and performances in large scale distributed applications with erasure codes and replication," Future Generation Computer Systems, vol.56, pp. 773-782, 2016.

[13] Stanford Medicine Health Trends Report, 2017, [Online]. Available: https://med.stanford.edu/content/dam/sm/sm-news/documents/StanfordMedicineHealthTrendsWhitePaper2017.pdf.

[14] E. Lockhart, K. Bak, L. J. Schreiner, D. C. Hodgson, E. Gutierrez, P.Warde, and M. B. Sharpe, "Best Practice Recommendations for the Retention of Radiotherapy Records," Clinical Oncology, vol. 29, no. 11, pp. 195-202, 2017.

[15] K. Gai, Z. Lu, M. Qiu, and L. Zhu, "Toward Smart Treatment Management for Personalized Healthcare," IEEE Network, vol. 33, no. 6, pp. 30-36, 2019.

[16] Proposals for storing medical data in USA, 2000, [Online]. Available: https://www.acr.org/-/media/acr/files/practice-parameters/radonc.pdf.

[17] Proposals for storing medical data in Australia, 2005, [Online]. Available: https://www.rcr.ac.uk/system/files/publication/field_publication_files/BFCO(06)2_retention_of_records(2011).pdf.

[18] Proposals for storing medical data in UK, 2016, [Online]. Available: http://webarchive.nationalarchives.gov.uk/20161101131024/http://systems.digital.nhs.uk/infogov/iga/rmcop16718.pdf.

[19] Proposals for storing medical data in China, 2017, [Online]. Available: http://www.nhc.gov.cn/fzs/s3576/201808/7a922e4803fa452f99d43a25ec0a3d77.shtml.

[20] R. Gupta, H. Gupta, U. Nambiar, and M. Mohania, "Efficiently querying archived data using hadoop," ACM international conference on Information and knowledge management, 2010, pp. 1301-1304.

[21] Y. Chen, Y. Zhou, S. Taneja, X. Qin, and J. Huang, "aHDFS: An Erasure-Coded Data Archival System for Hadoop Clusters," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 11, pp. 3060-3073, 2017.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," IEEE Symposium on Mass Storage Systems and Technologies, 2010, pp. 1-10.

[23] J. Watts and S. Taylor, "A practical approach to dynamic load balancing," IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 3, pp. 235-248, 1998.

[24] T. L. Saaty, "How to make a decision: the analytic hierarchy process," European journal of operational research, vol. 48, no. 1, pp. 9-26, 1990.

[25] S. Ghemawat, H. Gobioff, S. T. Leung, "The Google file system," ACM symposium on Operating systems principles, 2003, pp. 29-43.

[26] Y. Hsu, R. Irie, S. Murata, and M. Matsuoka, "A Novel Automated Cloud Storage Tiering System through Hot-Cold Data Classification," IEEE International Conference on Cloud Computing, 2018, pp. 492-499.

[27] Y. Li, B. Shen, Y. Pan, Y. Xu, Z. Li, and J. C. S. Lui, "Workload-Aware Elastic Striping With Hot Data Identification for SSD RAID Arrays," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 5, pp. 815-828, 2017.

[28] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's Time to Revisit {LRU} vs.{FIFO}," USENIX Workshop on Hot Topics in Storage and File Systems, 2020, pp. 1-7.

[29] PostMark, [Online]. Available: http://www.shub-internet.org/brad/FreeBSD/postmark.html.