RAID-6 Based Distributed Storage System

Siyue Zhang (G2204115C), Jingyi Yang (G2204056A), Ziqi Yin (G2202468F)

***Abstract*—The demand for large-scale, high-speed, and reliable storage systems has dramatically increased with the rapid growth of data volumes. Among different storage techniques, RAID-6 is commonly deployed for its capability to tolerate up to two-disk failure concurrently. In this project, we developed the RAID-6 storage system based on Galois Field theory and Reed-Solomon Coding using Python. Major system functions include `store`, `update`, `detect`, `rebuild`, and `retrieve`. In addition to basic functionalities, our system supports files of arbitrary size and user-defined configurations. We investigated the impacts of configuration parameters on the system performance for a long text file and an image file. The codes with read-me documents are available at https://github.com/siyue-zhang/RAID6-CE7490.**

***Index Terms*—RAID-6, Galois Field, Reed-Solomon codes**

I. Introduction

Since the conception of Redundant Array of Independent Disks (RAID) in 1988 [1], the data storage virtualization technology has been widely recognized and applied. In contrast to the concept of Single Large Expensive Disk (SLED), RAID combines multiple inexpensive and independent physical disks into one or more logical units. RAID mainly introduces three storage techniques: *Striping* splits the flow of data into chunks of a fixed size; *Mirroring* generates identical copies of data; *Parity* is calculated from data stripes by a certain function (e.g., XOR [2]) for data recovery. These techniques provide a cost-effective solution for large-capacity storage with fault tolerance, as well as simultaneous reads and writes.

Various RAID schemes have been developed to satisfy different levels of application requirements. RAID-0 [3] distributes data chunks uniformly across N disks, which is called striping. RAID-1 [4] mirrors data over N disks, which utilizes $1/N$ space but can tolerate failure of up to $N-1$ disks. By combining RAID-1 and RAID-0, RAID-10 replicates each data stripe once and thus reduces the useful storage by half. RAID-10 can tolerate 2 disk failures when the failed disks store different data. RAID-4 improves space utilization by storing data stripes in $N-1$ disks and parity in the N^{th} disk. Parity data are calculated values from data stripes, which can be used to restore the data if one of the disks fails. However, due to the fact that every data update requires writing to the parity disk, the parity disk becomes an I/O bottleneck. To tackle this issue, RAID-5 distributes the parity data over the disks so that the write load of parity data is balanced over all disks. Currently, RAID-10 and RAID-5 are widely used in real-world systems.

Siyue Zhang, Jingyi Yang and Ziqi Yin are with the School of Computer Science Engineering, Nanyang Technological University, Singapore, e-mail:(SIYUE001@e.ntu.edu.sg, JINGYI006@e.ntu.edu.sg, ZIQI003@e.ntu.edu.sg).

Project submitted on Nov 23, 2022.

While RAID-5 improves storage efficiency, the single parity systems are fragile, especially when multiple failures occur simultaneously. Besides, it may require a long rebuild time when the disk capacity is large. RAID-6 was proposed to tolerate a second failed disk by calculating two sets of parity data. There are various methods to implement RAID-6, including classic erasure coding techniques like Reed-Solomon (RS) coding [5] and more specialized codes such as EVENODD [2] and RDP [6]. The relatively recent erasure code, Mojette [7], showed significant performance improvements at the cost of a slight increase in storage overhead.

In this project, we developed the RAID-6 system based on the most popular Vandermonde-RS code using Python 3.7 and the `numpy` library. Galois Field and matrix operations were employed in the development. Six major functions were programmed for distributing data storage, updating modified data, detecting disk failure, restoring corrupted data, and retrieving data objects respectively. On top of the basic implementation, we also developed advanced features for accommodating real files of arbitrary size and supporting user-defined configurations for practical applications. The rest of the report is organized as follows: Section II gives an overview of the RAID-6 based distributed storage system developed in this project. Section III elaborates on arithmetic principles in the system design. Section IV introduces the experiment setup and analyzes the experiment results. Section V concludes the project with key takeaways.

II. Overview and System Workflow

A. Overview

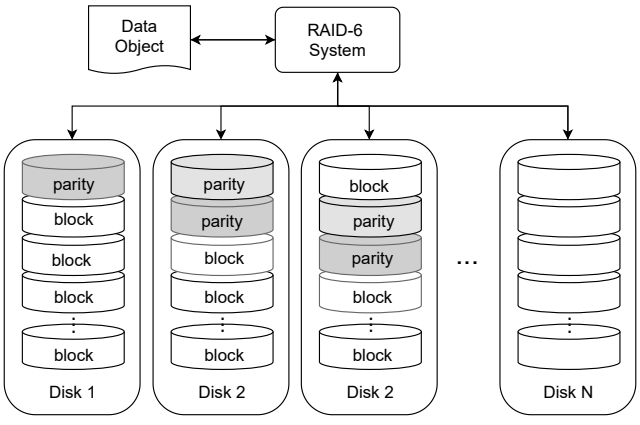

Fig. 1: Overview of RAID-6 storage system

An overview of the RAID-6 storage system is shown in Figure 1. In the system, we created the Galois Field $GF(2^8)$ operations according to the most popular RS coding [5]. The

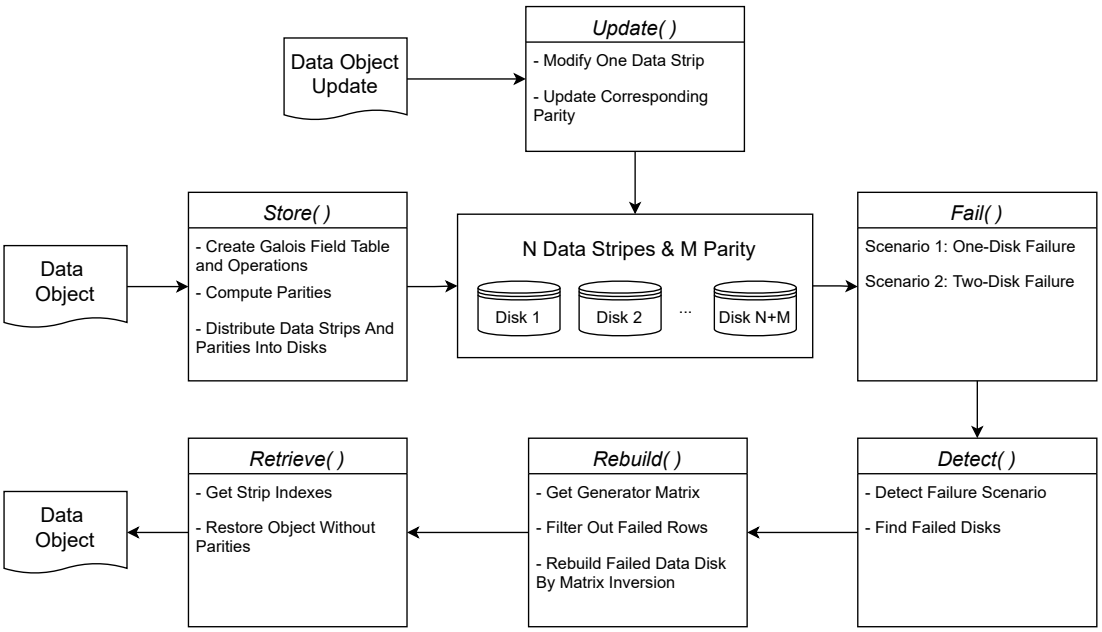

Fig. 2: RAID-6 system workflow

RAID-6 system is the interface between the user and the persistent storage for storing, modifying, and retrieving data. Our system has the following functionalities:

- Store and access abstract "data objects" across storage nodes using RAID-6 for fault-tolerance.
- Address the small-write bottleneck by evenly distributing parity data.
- Include mechanisms to determine the failure of storage nodes.
- Carry out the rebuild of lost redundancy at a replacement storage node.
- Accommodate real files of arbitrary size, taking into account issues like RAID mapping, etc.
- Support mutable files, taking into account updates of the content and corresponding parities.
- Support larger and configurable set of configurations, e.g., number of data and parity disks, chunk size, etc.

B. System Workflow

The RAID-6 storage system workflow is shown in Figure 2, which consists of 6 major functions. `Store()` is responsible for receiving data objects, data striping, parity calculation, and distributing data and parities into disks evenly. Users can use `Update()` to modify the data object where the function will automatically locate data stripe changes and update corresponding parities. `Fail()` is designed to introduce failure scenarios for experimenting, e.g., one-disk corruption and two-disk corruption. `Detect()` identifies and locates the failure that occurred. `Rebuild()` calculates the corrupted data by solving the remaining linear equations (e.g., using Matrix inversion). `Retrieve()` assembles the data stripes for the data object requested by users.

III. System and Workflow

A. Galois Field

The Galois field, i.e, the finite field, is a field of a finite number of elements. The Galois field has the following properties: 1). the operations of multiplication, addition, subtraction, and division are defined and satisfy certain basic rules. 2). the computation results of adding and multiplying the elements in the field remain in the field, 3). the calculation satisfies the commutative law, the associative law, and the distribution law, 4). there exists unit elements for multiplication and addition, i.e, for any element x in the field, there always exists an element y in the field satisfies $x + y = 1$ and another element z in the field satisfies $x \times z = 1$ if $x \neq 0$, 5). the number of elements in the field is limited. The most common examples of Galois fields are given by the integers mod p when p is a prime number.

Example 1: $GF(3)$'s element is $\{0, 1, 2\}$, it is a Galois field. The addition operations are like $1 + 0 = 1$, $2 + 2 = 4\%3 = 1$, and $0+1 = 1$. The multiplication operations are like $1 \times 1 = 1$, and $2 \times 2 = 4\%3 = 1$.

The number of elements of a Galois field is called its order or its size. A Galois field of order q exists if and only if q is a prime power p^k, where p is a prime number and k is a positive integer. In the application, we usually set $p = 2$ and use different k values. For the field $GF(2^w)$, if w is too small, then the small field will limit the maximum number of disks. On the other hand, if the w is too large, then the large field would require extremely large tables. Here we choose the commonly used field of $GF(2^8)$ which allows for a maximum of 255 data disks.

Example 2: $GF(2^2)$'s element is $\{00, 01, 10, 11\}$, the addition operations are like $00 + 01 = 01$, $01 + 00 = 01$,

$10+11=01$, and $11+10=01$. The multiplication operations are like $01\times01=01$, $10\times11=01$, and $11\times10=1$.

How to conduct the addition operation and multiplication operations in the Galois field efficiently is a critical problem. Here we make use of polynomials to conduct the addition operation and multiplication operations.

Example 3: $GF(2^2)$'s element is $\{00,01,10,11\}$, the addition operation $10+11=01$ can be viewed as $(x)+(x+1)=0x+1=1$. Similarly, the multiplication operation $10\times11=01$ can be viewed as $x\times(x+1)=(x^2+x)\%(x^2+x+1)=1$. x^2+x+1 is the primitive polynomial.

The primitive polynomial is a minimal polynomial of an extension of finite fields, serving the role of module integer, like the prime integer 3 in the $GF(3)$ finite field.

Here we introduce how addition, subtraction, multiplication, and division operations are done efficiently by hardware and software.

Definition 1: (Addition) The addition field operator $\bigoplus$ is performed by XOR, which is very fast via hardware acceleration.

Definition 2: (Subtraction) Due to the binary value property of polynomials, the subtraction field operator $\bigominus$ is performed in the same way as the addition operation.

Definition 3: (Multiplication) The multiplication $\bigotimes$ is performed by bitwise AND with two logarithm tables. The details are stated as follows.

Definition 4: (Division) The Division can be performed by multiplication with an inverse, *i.e.*, $A/B=A\bigotimes B^{-1}$

We can infer from the definition that the addition operation satisfies the commutative law and the associative law. The multiplication operation also satisfies the commutative law and the associative law. The proof is simple so we omit it here.

We further explain how to perform the multiplication operation and the division operation in the $GF(2^8)$ field. It is obvious that the addition and subtraction operations are very fast thanks to the hardware acceleration of the XOR operation of vectors. The basic idea is that we want to make use of the addition and subtraction operations to implement the multiplication operation. The pseudocode of the two tables' initialization is shown in Algorithm 1. The $gfexp$ table is used to map the indices from 1 to 255 to its logarithm in the Galois Field. And the $gflog$ table is used to map the indices 0 to 254 to their inverse logarithm in the Galois Field. The max indicates the maximum number in the Galois Field, which is 2^w. The $\wedge$ is the XOR operation here and the $\&$ is the AND operation. The $b\&max$ operation always equals 1 when b is less than max, so it is used to check whether b is greater than the maximum number max. (line 7) The modulo operation can be executed by the XOR operation since $A=B+C$, then $A\%B=C$, which is the same as $A-B$ and $A\wedge B$ in the Galois field. If b & max, then we have b is greater than the maximum number, we assign b to be $b\%modulus$, where the modulus is derived from the primitive polynomial. (line 8)

Obviously, $A\bigotimes B=gfexp[gflog[A]+gflog[B]]$ and $A\bigotimes B^{-1}=gfexp[gflog[A]-gflog[B]]$. The two tables reduce the computation cost of multiplication and division operations to that of addition and subtraction operations.

Algorithm 1: Initialize logarithm tables

Input: $w=8, modulus=0b100011101$
Output: two logarithm tables

```
1 max = 1 << w;
2 b = 1 ;
3 for log ← 0 to max do
4     gflog[b] ← log;
5     gfexp[log] ← b;
6     b ← b << 1;
7     if b&max then
8         b ← b ∧ modulus;
```

B. Reed-Solomon Code

The Reed-Solomon Code is an error-correcting code with a wide range of applications in digital communications and storage. A typical Reed-Solomon code system takes digital data as input and then generates redundant parity bytes to assure we can still get original data even if we lost some data during the transmission. We refer to the process of adding the extra bytes to ensure the data reliability as the encoding process, and the system that carries out this process the encoder. When the transmission is over, the Reed-Solomon code system processes the data and attempts to correct errors and recover the original data. We refer to the process of restoring the original data as the decoding process, and the system that carries out this process the decoder.

Typically, RAID-6 consists of N data disks and $M=2$ parity disks. It is also worth noting that, in order to improve the reliability and efficiency of the storage system, we usually distribute the original data and parity data into different disks to ensure every disk has both original data and parity data, which is shown in Fig. 1. We can see that parity blocks are distributed among all disks.

The basic principle of RAID-6 can be understood as we have $N+2$ equations with N variables. There might be some information lost during the transmission, resulting in some equations' coefficients being corrupted. RAID-6 satisfies that if two equations are corrupted, the remaining equations are still able to solve the equations. This leads to the critical problem of how to design the coefficients. In other words, how to construct an irreducible matrix. Here we introduce the Vandermonde matrix, which is an irreducible matrix.

The encoder process is stated as follows, where F is the Vandermonde matrix shown in equation 2. As shown in equation 1, we use the Vandermonde matrix to encode the original data D, and P is the encoded data.

$$P=FD \tag{1}$$

$$P=\begin{bmatrix}1 & 1 & \cdots & 1\\ 1 & 2 & \cdots & N\\ \vdots & \vdots & \ddots & \vdots\\ 1 & 2^{M-1} & \cdots & N^{M-1}\end{bmatrix} \tag{2}$$

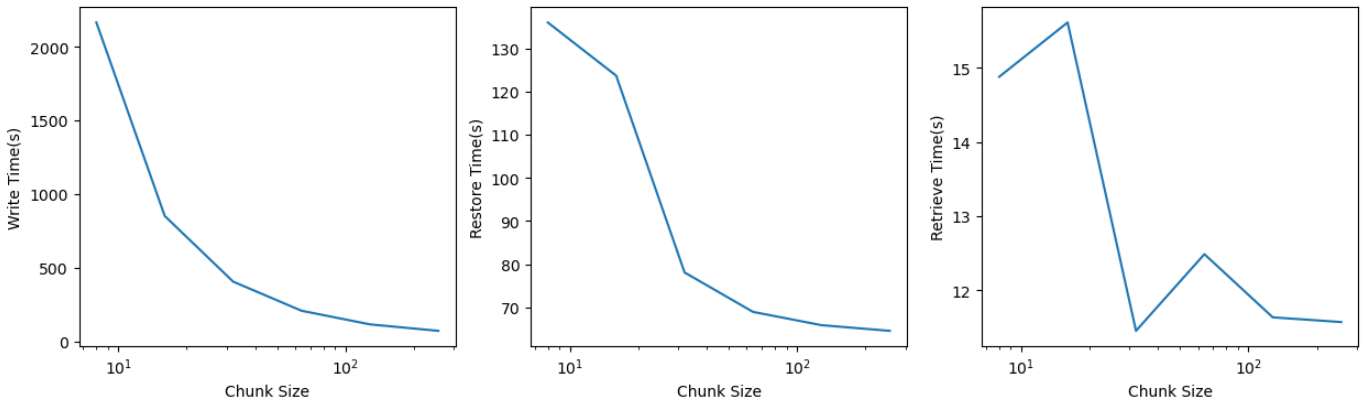

Fig. 3: RAID-6 system performance with varying chunk sizes on text data

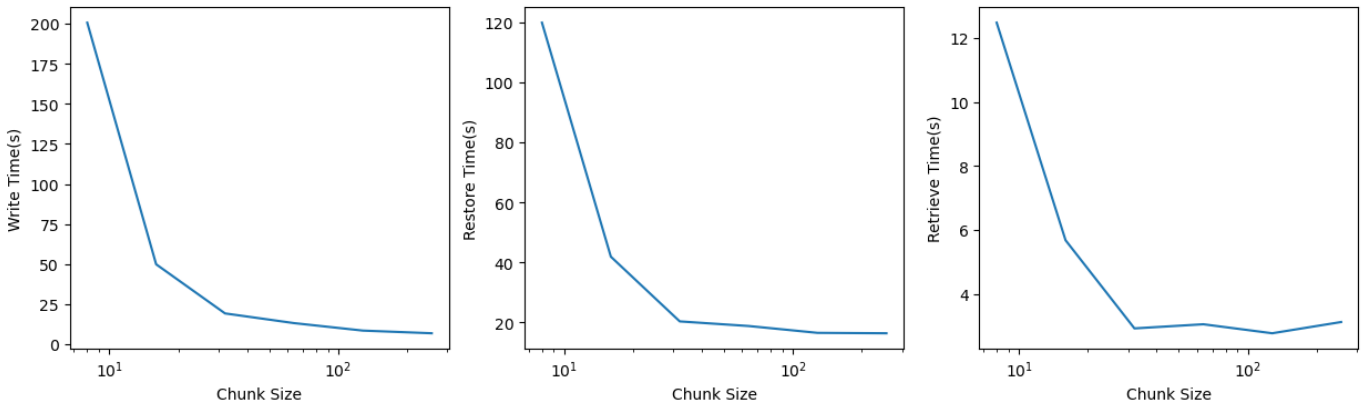

Fig. 4: RAID-6 system performance with varying chunk sizes on image data

The decoder process is stated as follows, we define the matrix A=$\begin{bmatrix} I \\ F \end{bmatrix}$ and E=$\begin{bmatrix} D \\ P \end{bmatrix}$, where I is an identity matrix. Then we derive the following equation 3 according to equation 1.

$$AD = E \tag{3}$$

When disk failures are detected in the RAID-6 system, the remaining data are used for restoring the original data D. The matrices A' and E' are obtained by deleting the corresponding corrupted rows in A and E. Then we have the equation 4.

$$A'D = E' \tag{4}$$

To obtain the matrix D, the inversion of A' is multiplied to both side of the equation, which gives us equation 5. With the restored original data D, we compute parity data P following the equation 1 and write both original data and parity data into each disk.

$$D = (A')^{-1}E' \tag{5}$$

IV. Experiments

A. Experiment Setup

In the experiments, we studied the impact of the configuration parameters i.e., chunk size, on the RAID-6 system performance. We experimented with a chunk size of 8, 16, 32, 64, 128, 256 respectively, and record the write time (the execution time of `Store()`), restore time (the execution time of `Rebuild()`), and retrieve time (the execution time of `Retrieve()`).

For all the experiments, we use a total of $N + M = 8$ disks, including $N = 6$ data disks and $M = 2$ parity disks. We conduct the experiments on two data files, a text file of Shakespeare's plays, as well as an image file. The two file sizes are 5.46MB and 1.43MB. The image file is shown in Figure 5.

B. Experiment Results

Figure 3 and Figure 4 show the performance of the RAID-6 system with varying chunk sizes. We observe that for both data files, the write time decreases exponentially as the chunk size increases. This is because the data are written to the disk through striping, and the total number of writes is inversely proportional to the chunk size. Similarly, the restore time also decreases exponentially as the chunk size increases, as the data

Fig. 5: Image file used in the experiments

are recovered stripe-by-stripe. For the image file, the retrieve time also decreases exponentially, as the data are read stripe-by-stripe. However, for the text file, the retrieve time slightly decreases as the chunk sizes increases, and plateaus after the chunk size reaches 32. This is because, for large data files, the retrieve time is dominated by disk I/O, which is independent of the chunk size.

V. Conclusions

In this project, we further deepen our understanding of the RAID-6 system and develop our coding ability. We conduct the experiments in python 3.7 under a ubuntu environment. Instead of using existing code, we implement the basic operations in the Galois field by ourselves. Based on the of Reed-Solomen Coding, the RAID-6 system is able to support all basic functions such as data storage, failure detection, and failure recovery. Our experiment results on several datasets demonstrate that the RAID-6 system is a reliable system and can be applied to real applications.

Acknowledgments

We would like to thank Prof. Anwitaman Datta for teaching the course CE7490 Advanced Topics in Distributed Systems and for the resources provided in the lecture.

References

[1] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[2] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on computers*, 44(2):192–202, 1995.

[3] Dina Bitton and Jim Gray. Disk shadowing. In *VLDB*, volume 88, pages 331–338, 1988.

[4] Peter M Chen, Garth A Gibson, Randy H Katz, and David A Patterson. An evaluation of redundant arrays of disks using an amdahl 5890. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, 1990.

[5] James S Plank. A tutorial on reed–solomon coding for fault-tolerance in raid-like systems. *Software: Practice and Experience*, 27(9):995–1012, 1997.

[6] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. San Francisco, CA, 2004.

[7] Dimitri Pertin, Alexandre Van Kempen, Benoît Parrein, and Nicolas Normand. Comparison of raid-6 erasure codes. In *The third Sino-French Workshop on Information and Communication Technologies, SIFWICT 2015*, 2015.