Fault-tolerant and distributed data storage with RAID6

Jeremy Gerster, Simon Menke

***Abstract*—In this project, we propose an implementation of RAID6, a method for distributed, fault-tolerant storage by utilizing dual parities and Reed-Solomon codes. Hereby, in our RAID6 implementation, we aim at simulating disk failure and testing for recovery. In addition, we implement remote storage support through cloud services. In our subsequent experiments, performed on our system, we find that choosing larger chunk sizes leads to exponentially faster processing times, likely due to the reduced number of I/O operations and lower overhead associated with handling fewer, larger chunks. Furthermore, we observe that adding files to our RAID6 system takes up more time than deleting files.**

***Index Terms*—RAID6, distributed, data**

I. Introduction

In the modern era of digital transformation, the demand for efficient storage, management, and availability of vast amounts of data continues to grow rapidly. Distributed systems play a central role as they are the basis of many modern applications, ranging from cloud computing to big data as well as highly available databases and data centers. In distributed systems, data is often dispersed across multiple physical storage devices to achieve high availability and low processing times. However, with this comes the trade-off between efficiency, cost, and failure tolerance. An important component of reliable storage solutions in distributed systems is the concept of "Redundant Array of Independent Disks" (RAID) as first proposed in [4]. RAID technologies were developed to improve the performance and availability of data storage by using a combination of multiple physical disks. Through the use of different RAID levels, companies and organizations can fulfill different demands for speed as well as data redundancy and integrity. RAID systems use methods such as data striping, parities, and mirroring for improvements such as increased data transmission rate or better storage efficiency.

RAID6 is one of the better-known and widely used RAID levels, particularly in situations that require both high fault tolerance as well as efficient use of storage space. It expands the functionality of RAID5 by storing two independent parity blocks on different physical disks. Double parity enables a fault tolerance of two disks without losing data or impacting integrity. This high fault tolerance makes it ideal for critical applications where data safety and availability are of great importance such as cloud storage systems. RAID6, however, also comes with its own challenges. Through the computation of an additional parity, the compute needed increases steeply. Recovering large data sets can also become quite time-consuming [3].

In the scope of this project, we implement RAID6 and evaluate its performance. This work focuses on technical aspects of our implementations with descriptions of our architecture and explanations of critical functions.

II. Background

In the following, we cover the key concepts behind RAID 6, necessary for understanding its implementation. First, we will explain the core principles of RAID 6, including how data is distributed across multiple disks and how parity calculations ensure protection against data loss. Next, we will discuss the arithmetic behind the parity calculation used in RAID 6, known as Galois Fields, a specific type of finite field. Finally, we will describe how data recovery is performed using Reed-Solomon codes, which are built on Galois Fields.

A. RAID6 Overview

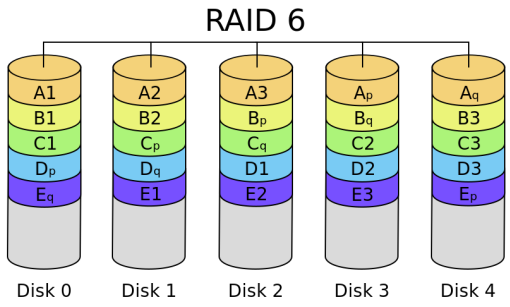

Fig. 1. RAID6 chunk allocation [6]

As mentioned before, RAID6 uses two parities to ensure data can be recovered even if two disks fail. It divides the data into chunks, based on a user-defined chunk size, and distributes these chunks across a set number of disks. The allocation process begins by storing the first chunk on the first disk, the second chunk on the second disk and so on and so forth until all disks contain a chunk. The allocation now repeats by saving the next chunk on the first disk again. This process repeats until all chunks are stored. One layer of chunks is called a stripe as visualized in Fig. 1, where one stripe is indicated in the same color. In the event of disk failure, RAID6 enables the recovery of lost information by using the remaining data inside a stripe. To be able to recover up to two disks, two parities are computed for each stripe. These parities are stored on independent disks as well, where their positions typically shift by one disk for each consecutive stripe. The first parity, denoted as p, gets computed using a simple XOR operation of

all data chunks inside a strip. This parity enables the recovery should only one disk fail. For the case of two failed disks, a second parity q is based on Galois fields, a mathematical structure used in finite field arithmetic [1].

B. Galois Fields

This section is mainly based on [2].
A Galois Field is denoted as GF(2^n) where n represents the degree of the field extension, indicating that the field contains 2^n elements. In the following we talk about GF(2^8), i.e. a finite field that consists of 256 elements which get represented as bytes. Operations in GF(2^8) have multiple properties:

- **Addition and Subtraction**: These operations are equivalent and are performed using a bitwise XOR. Through this property, we can simplify calculations:

$$a + b = a - b = a \oplus B \tag{1}$$

  where a and b are elements in GF(2^8).
- **Multiplication**: Multiplication in GF(2^8) is defined modulo an irreducible polynomial, typically $x^8 + x^4 + x^3 + x^2 + 1$. For example, multiplication by $\{02\}$ can be implemented using a linear feedback shift register (LFSR), which performs bitwise shifts and XORs:

$$\begin{aligned}(x \cdot \{02\})_7 &= x_6,\\ (x \cdot \{02\})_6 &= x_5,\\ (x \cdot \{02\})_5 &= x_4,\\ (x \cdot \{02\})_4 &= x_3 + x_7,\\ &\dots\\ (x \cdot \{02\})_0 &= x_7\end{aligned}$$

  This operation is important for performing fast and efficient multiplication that are needed in error correction algorithms.
- **Inverse and Division**: For any non-zero element $a \neq \{00\}$, there exists a multiplicative inverse a^{-1} such that:

$$a \cdot a^{-1} = \{01\}$$

  Division in GF(2^8) is defined as multiplication by the inverse:

$$a/b = a \cdot b^{-1}$$

  where $b \neq \{00\}$.
- **Exponentiation and Logarithms**: Galois Fields also support exponentiation, where raising an element to a power is done modulo 255 (because there are 255 non-zero elements in GF(2^8)):

$$a^{256} = a, \quad a^{255} = 1, \quad a^{254} = a^{-1}$$

  Logarithms in GF(2^8) are used to simplify multiplication and division by converting them into addition and subtraction operations.

C. Reed-Solomon Codes in RAID-6

This section is mainly based on [5].
Reed-Solomon codes have a wide range of use cases, however here we will focus on the application to RAID6. Reed-Solomon codes are a type of error-correcting codes used in RAID-6 to provide data redundancy and enable recovery from the failure of any two disks. RAID-6 achieves this by computing parities, p and q, all data disks:

- Parity p is simply the XOR parity of all data disks:

$$p = d_0 +_1 + d_2 + \ldots + d_{n-1}$$

  This is equivalent to summing all data bytes using XOR. If a single disk fails, the missing data can be reconstructed using the p parity.
- Parity q is computed as a weighted sum of the data disks, with weights being powers of a generator g of the Galois Field:

$$q = g^0 \cdot d_0 + g^1 \cdot d_1 + g^2 \cdot d_2 + \ldots + g^{n-1} \cdot d_{n-1}$$

  Typically, $g = \{02\}$ is chosen, which is a generator of GF(2^8). The q parity is a more complex parity check that allows RAID-6 to recover from the failure of either one or two disks.

D. Recovery from Disk Failures

When a disk fails, RAID-6 uses p and q to reconstruct the missing data:

- **Single Disk Failure**: If one data disk, say D_x, is lost, it can be reconstructed using p:

$$d_x = p - (d_0 + d_1 + \ldots + d_{x-1} + d_{x+1} + \ldots + d_{n-1})$$

  Since addition and subtraction are the same in GF(2^8), this reduces to:

$$d_x = p + (\text{sum of all other data disks})$$

- **Two Disk Failures**: If two data disks, say d_x and d_y, are lost, both parities are used:

$$p_{xy} + d_x + d_y = p$$
$$q_{xy} + g^x \cdot d_x + g^y \cdot d_y = q$$

  Here, p_{xy} and q_{xy} are computed assuming that d_x and d_y are zero. These equations are then solved simultaneously for d_x and d_y by using properties of Galois fields such as multiplication by inverses and logarithms.

III. System architecture

Our RAID6 implementation mainly follows the basic ideas described in the above background. The main idea of our implementation consists of simulating disk failures to test for successful recovery of our system afterward. However, in our project, we also introduce some additional configurations and features.
One of them is that we offer the possibility to either create new RAID6 setups or interact with existing ones. Moreover, our system is specifically designed to support binary file structures

as they are one of the most widely used. Currently, this consists of the file formats JPG, MP3, and PDF.
Furthermore, our project is adaptable to either local as well as remote storage through cloud servers. The former setup supports both fully variable chunk sizes and number of disks. The cloud version, which is based on the local setup, in addition, distributes the disks across multiple virtual servers. However, in contrast to the local version, the cloud version is constrained by our available resources. As a result, the system currently supports a maximum of 8 nodes. Of these 8 nodes, up to 5 are data disks, 2 are used for storing parities, and 1 serves as the master server, managing the distribution across the other servers. Here, we employ both a local client as well as a server.
In the following, we explain the system architecture in more detail, consisting of how the user interacts with the RAID system, the actual underlying RAID6 implementation, as well as the server logic, necessary for our cloud storage configuration.

A. User interaction with the RAID6 system.

To access our RAID6 system, the user runs the main file. This provides multiple functionalities for the user, such as creating new RAID setups, loading existing setups, and doing file operations such as adding or deleting files. In either case, in the end the goal consists of testing the modified system for recovery. For that, the user can choose 1 or 2 disks to delete. Subsequently, the system tries to recover the missing data, following the described procedure in the background section. The corresponding flowchart is depicted in Fig. 2:

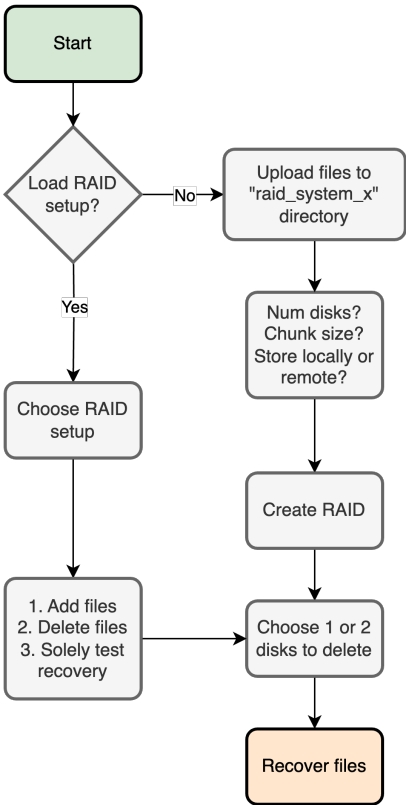

Fig. 2. Flowchart of user interaction with RAID6 system

So once the script is executed, the user can either choose between loading an existing RAID setup or creating a new one.

```
.
├── Initial_distributed_files
├── Recovered_files
├── Reloaded_Initial_distributed_files
├── disks
├── files
└── metadata.json
```

Fig. 3. Directory structure of "`raid_system_x`"

1) Creating a new RAID6 setup: If the user creates a new setup, the script subsequently creates a new RAID6 setup directory "`raid_system_x`" (the "x"th system) and asks for files to be uploaded in the format of either JPG, MP3 or PDF to the files folder within this directory. An overview of what the directory looks like is provided in Fig. 3. Once the user uploads the desired files to be stored, he is asked to provide the necessary configuration parameters. These include the number of disks, the chunk size, and whether data should be stored locally or remotely. Given these parameters, the RAID setup process will be executed, which consists of distributing the data on the disks and then testing recovery from simulated disk failure(the specific process is further explained in B).

2) Loading an existing RAID6 setup: On the other hand, if the user decides to interact with an existing setup, then he will be provided with 3 options. He can either 1. add new files, 2. delete files, or 3. just test for recovery e.g. delete disks and recover data from the existing raid storage setup. In the case of option 1. and 2. failure and recovery will also be subsequently simulated.
In the following section, we give an overview and further elaborate on how our RAID6 implementation is designed to achieve these functionalities.

B. RAID6 implementation

Our RAID6 implementation includes several key functions for managing data distribution, erasure, recovery, and maintaining existing RAID setups. The main functions, in their order of execution, are as follows:

- `_init` – Initializes the RAID 6 setup by setting up necessary variables and parameters such as chunk size, number of disks, Galois Field, etc. If an existing configuration is provided, it loads the corresponding metadata.
- `distribute_data` – Distributes files data across the RAID disks, while ensuring that parities are properly calculated and stored.
- `compute_parity` – Computes the P and Q parity values based on the distributed data for error correction.
- `delete_disks` – Simulates disk failures by removing specific disks, allowing the system to test data recovery capabilities.
- `rebuild_data` – Reconstructs the lost data by using the remaining disks and the parity information.

Furthermore, to support existing RAID setups, the system stores metadata for each configuration. This metadata contains all the essential information needed to reconstruct the

RAID matrix from the available disks. The stored information includes the chunk size, number of disks, number of stripes, name of previously stored files, as well as where each file is saved on the disks.
Additionally, ensuring functionality for existing setups requires the following functions:

- `load_metadata` – Loads the metadata necessary for initializing a pre-existing RAID6 configuration.
- `load_existing_data` – Loads the data from the disks into the RAID6 matrix according to the metadata.
- `update_file_metadata` – Updates location of files (start, end stripe in the matrix) when files get deleted.
- `recalculate_parity_locations` – Recalculates the P and Q parity locations when files get deleted.
- `save_metadata` – Saves the new updated metadata after the final reconstruction process.

1) Analyzing `distribute_data` and `rebuild_data`: In the following part, we provide a more detailed explanation of two key components of the system: `distribute_data` and `rebuild_data`. These functions offer a comprehensive overview of the system architecture and illustrate how various functionalities interact with each other:

`distribute_data` is responsible for distributing data across all available disks in our RAID6 system.
Here, again we differentiate between the scenario of a new setup and an existing one.
New RAID6 setup:
For a new setup, in order to manage how to distribute the data, we follow the approach described in Figure 1 interpreted as a matrix. Therefore we initialize a matrix of size $m \times n$ where m equals the number of stripes and n represents the number of disks available. The number of stripes is calculated by dividing the file data into chunks based on the user-defined chunk size. Two chunks in each stripe are hereby reserved for the q and p parities for recovery in case of disk failure. Then in a first step, the locations of the P and Q parities are calculated, following the same logic as shown in figure 1. In the next step, the rest of the matrix is filled with the data chunks. After that, `compute_parity` is called. This function calculates the P and Q parities, following the same approach described in the background, considering the data chunks on the same stripe in the matrix. After that, the matrix is filled. Therefore in the next step, each column of the matrix is saved to a disk. This can be done either locally or remotely. Additionally, to check the correctness of the distribution, we also implemented a debugging mechanism where the data gets reloaded from the disks and the files recreated, stored in `Initial_distributed_files`.
Existing RAID6 setup:
If a system already exists, then in a first step, the old matrix is reloaded by calling `load_existing_data` and using the stored disks' data as well as available metadata. Then again the three cases are differentiated:
If only the recovery of the system is tested after simulated disk failure and there are no new files, then, once the matrix is reloaded, no additional steps are required.
If there are new files added, then for each file, there will be individual matrices initialized and populated with the corresponding file data, following the same approach as creating a new setup. Afterward, they are combined with the reloaded matrix to form the updated matrix.
If files are deleted, then the function identifies the associated stripes of these files and removes them from the matrix. Additionally, due to the removed files, the file metadata, meaning the information where each file is stored on the stripes of the matrix, as well as the Q and P location information, are updated.
Finally, in either of those cases, once the new matrix is created, each column of the matrix is saved to a disk. For debugging purposes, the reconstructed files can then again be found under `Initial_distributed_files`.

`rebuild_data` is responsible for reconstructing missing data from an existing RAID6 setup when either one or two disks fail. First, it rebuilds the matrix by reading the content of the remaining disks. When the content of a disk is missing due to failure, it initializes the corresponding matrix cells with zeroes. When all available disk data is being read, the function continues by reconstructing the missing data using the q and p parities. The implementation follows the approach described in the background section. Hence, it rebuilds the files, stores them in `Recovered_files`, and in the end calls `save_metadata`.

C. Server logic of the cloud implementation

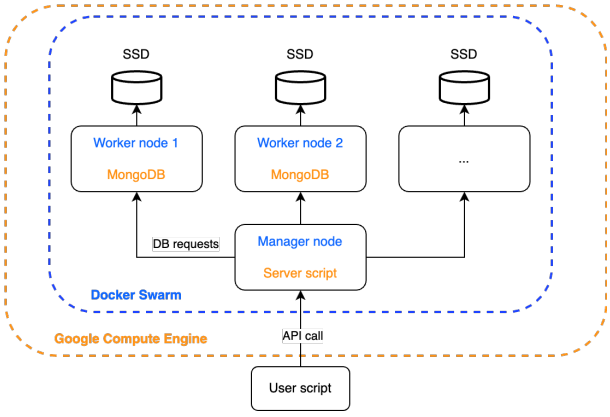

Fig. 4. Server architecture

Although the remote storage procedure is based on the previously described architecture used for the local system, the data is accessed and stored differently. Thus, in the following, we will explain how the server architecture is set up and how the client interacts with the server.
To achieve truly distributed, remote storage, we set up cloud infrastructure with independent virtual machines (VMs). As seen in Fig. 4, it consists of a manager node that contains the `server.py` script, responsible for running the server, as well as multiple worker nodes containing MongoDB instances.

Through Docker Swarm, we get the manager-worker relationship needed for RAID6. The manager node controls the data nodes and can distribute files after they have been processed. The data nodes merely have to implement a database and be able to receive requests. As we set up the VMs in Google Compute Engine with a free trial, we could only get 8 instances leading to a maximum of 7 data nodes and therefore a maximum of 7 disks in a RAID setup. The server acts as an interface between the client application and the distributed storage system, constructed using FastAPI for high-performance API handling. It provides endpoints for uploading, retrieving, and deleting file chunks on simulated disks, each represented by a separate MongoDB instance utilizing GridFS for file storage. Through the `api_client.py` file, the local script interacts with three endpoints: `/upload` for uploading file chunks to a specified disk, `/data` for retrieving chunks from a disk, and `/reset` for deleting files from a given database server. The server connects to the appropriate MongoDB instance based on the disk number provided in the request. During the upload process, it decodes the base64-encoded chunk received from the client, stores the binary data in GridFS, and returns a file ID after successful storage. For data retrieval, the server converts the provided file ID to a BSON `ObjectId`, retrieves the file from GridFS, and streams it back to the client. The deletion process involves removing the specified file from GridFS on the corresponding MongoDB instance. Error handling is integrated to provide meaningful feedback, returning specific HTTP status codes and messages for issues like missing files or general exceptions.

IV. Runtime Experiments and Results

In our experiments, we evaluated the performance of our RAID6 setup by measuring the runtime from the moment where the files are distributed to the simulated disk failure and the subsequent recovery process. For all experiments, this runtime was measured with respect to the chunk size. Furthermore, for each experiment, we included multiple more variables to analyze their specific impact on the runtime.
It is worth noting, that in our experiments, we did differentiate between the usage of the different available data formats JPG, MP3, or PDF, as these are all binary files, thus handled equivalently in our implementation.
In the following we further elaborate on the different experiments and the results we obtained:

A. Disk failures

In the first experiment, we measured the runtime based on chunk size as well as the number of failed disks. The number of failed disks is set to either 1 or 2 disks while we used chunk sizes of 32, 64, 128, and 256 bytes. For each combination of chunk size and number of failed disks, a new RAID setup is created, following the process described in III-A1. Hereby, an MP3 and PDF file totaling 2.3MB are distributed in remote storage. Furthermore, we used 7 disks to distribute the data on our cloud storage setup.
Fig. 5 illustrates our results for the experiment. As the chunk size increases from 32 seconds to 256 seconds, the runtime

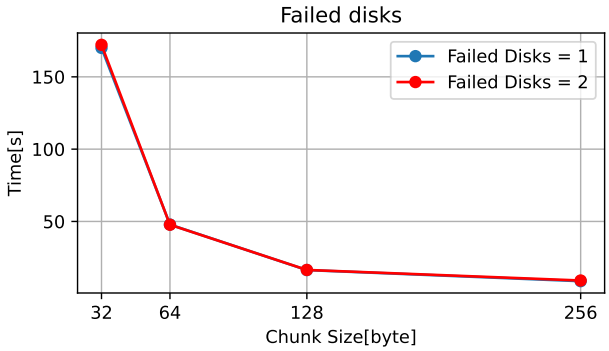

Fig. 5. Runtime comparison for one and two failed disks

exponentially decreases for both one and two failed disks. For two failed disks, the time decreases from around 150 seconds at a chunk size of 32 bytes to approximately 40 seconds at a chunk size of 64. Further increases in chunk size to 128 and 256 bytes result in less reduction in time of around 35 and 30 seconds, respectively. Moreover, it is apparent, that as the runtime for one and two failed disks does not greatly differ, making the graphs nearly indistinguishable. Although, for a chunk size of 32, the deviation becomes slightly more visible.

B. File size

In this experiment, we measured, how the runtime behaves with respect to the chunk size as well as different file sizes. For this, we used chunk sizes of 16, 32, and 64 bytes. The file size are chosen to be from 400KB to 1000KB in steps of 100KB respectively. It is worth noting that for this experiment, we reduced the maximum considered chunk size, as high chunk sizes result in a short runtime that can not be properly compared, as the differences mostly stem from server latency and processor scheduling.

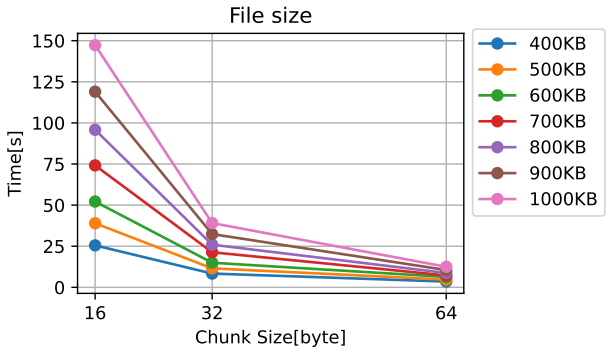

Fig. 6. Runtime comparison for different file sizes

Fig. 6 shows the results of this experiment and, like the previous experiment, we observe an exponential decrease in runtime as the chunk size increases. Notably, this trend is visible across all file sizes. Moreover, it is apparent, that a higher file size leads to a longer run time. Furthermore, we noticed, as seen in the figure, that the lower the chunk size,

the more visible there is a difference in runtime between the file sizes. So for example for chunk size 16, there is a bigger difference between the runtime for the file sizes, than it is for 64. Furthermore, this difference in runtime for a fixed chunk size seems to increase exponentially for different file sizes. Thus, for instance, the 800KB file has a 3-4 times longer processing time than 400KB which is only half the size. Likewise, 1000KB takes more than three times the runtime compared to half the file size, 500KB.

C. Local and remote storage

For this experiment, we test whether there is a noticeable difference when storing a file on a local machine or on virtual machines in the cloud. Here, we evaluated this by distributing a 1000KB file, where the number of failed disks is set to 1, and the number of disks is to 7. Upon examining Fig. 7,

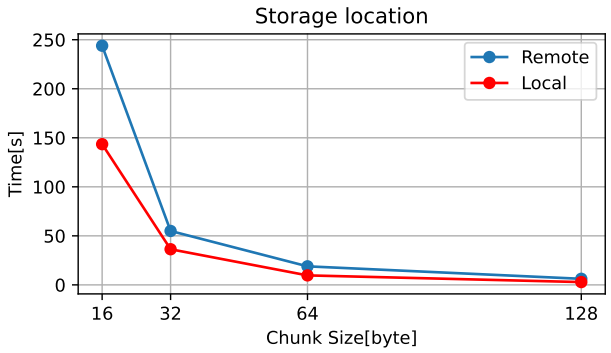

Fig. 7. Comparing processing time based on storage location

it is apparent that the runtime is consistently higher when using remote storage compared to local storage for all chunk sizes. At a chunk size of 16 bytes, the processing time for remote storage is significantly higher than for local storage. However, as the chunk size increases, this difference becomes less apparent. Thus, again, we observe a similar exponential trend for both local and remote storage, akin to the experiments before. This significant difference in processing time for small chunks is likely due to the fixed network latency associated with remote storage. As chunk size increases, the network latency cost becomes less pronounced compared to the actual data transfer time, which might explain the convergence in processing times for larger chunk sizes.

D. Add and delete operations

In this experiment, we compared the impact of add and delete operations on processing time. We used a 500KB file to add or delete data from an existing RAID6 system.

Fig. 8 suggests that, like previous experiments, processing times for both operations decrease exponentially as the chunk size increases. Additionally, it can be seen that adding files consistently requires more time than deleting across all chunk sizes. This might be because adding files could imply a longer recovery time, as there is more data to be processed. In contrast, deleting files reduces the load on distribution,

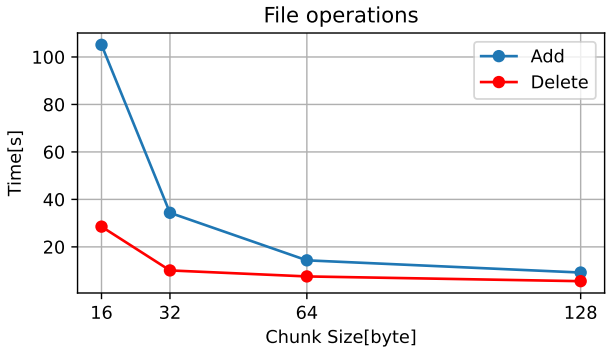

Fig. 8. Runtime comparison for add and delete file operations

and therefore, the runtime is potentially lower. Moreover, the performance gap between adding and deleting files decreases as chunk size increases. This is likely because larger chunks reduce the number of I/O operations required for adding and deleting, making the processes more efficient overall. Thus, this exponential trend observed for both operations as chunk size increases might reflect the overall reduction in processing overhead when managing fewer, more extensive data blocks.

V. Conclusion

In our experiments, we evaluated the performance of our RAID6 implementation under various conditions. Our results suggest a consistent trend of longer runtime for smaller chunk sizes, irrespective of parameters such as failed disk or file sizes. This is likely due to the reduced number of I/O operations and lower overhead associated with handling fewer, larger chunks, which significantly impacts the overall performance. Moreover, in the disk failure experiment (IV-A), we unexpectedly observed that the number of failed disks had a minimal impact on processing time, particularly at larger chunk sizes. This suggests that the distribution process and the recovery most likely take up most of the processing time, whereas the actual data recovery calculation runs considerably faster. Furthermore, experiment IV-B implies that a linear increase in file size is most likely correlated to exponentially longer processing time. Moreover, our experiment on storage location (IV-C suggests that as chunk size increases, the network latency cost becomes less pronounced compared to the actual data transfer time, which might explain the convergence in processing times for larger chunk sizes. When looking at the file operations in experiment IV-D, namely add and delete, we observe that adding files could imply a longer recovery time, as there is more data to be processed. In contrast, deleting files reduces the load on distribution, and therefore, the runtime is potentially lower. Moreover, the performance gap between adding and deleting files decreases as chunk size increases. This is likely because larger chunks reduce the number of I/O operations required for adding and deleting, making the processes more efficient overall.

VI. Further Work and Lessons Learned

We noticed some areas where our implementation could be further extended. For instance, the storage of our MongoDB

instances is currently handled within the default storage of a Docker container instead of a dedicated, persistent storage solution. Adding multiple persistent storage solutions per node would also enable a higher number of disks inside the RAID system and reduce the cost of infrastructure and network bandwidth. Furthermore, when a connection to the server or between the nodes fails, there is currently no proper mechanism for handling it gracefully. Instead of an error message, the program could work with checkpoints to avoid restarting the whole process. Another improvement would be supporting more file formats, such as plain text files. However, we noticed that opposed to binary files, they can require significantly different processing mechanisms, making it challenging to implement. Moreover, we recognized that our code could be simplified and optimized on multiple occurrences. For example, creating separate functions for disk writing and checking for file format could have simplified the code and offered more flexibility. Furthermore, in terms of performance analysis, another idea for the experiments would be to evaluate the runtime based on varying numbers of disks. Lastly, adding or deleting files could have been done more dynamically, where the system automatically recognizes any changes made to the uploaded files and starts the distribution process by itself.

As for lessons learned, we recognized that distributing data efficiently across disks can be complex, especially when balancing performance, fault tolerance, and space utilization. Furthermore, we got hands-on experience with RAID6, which greatly helped us solidify our theoretical understanding and, as mentioned, taught us the complexities of actually implementing this system in practice. In addition, taking our storage system to the cloud gave us valuable experience in server infrastructure.

REFERENCES

[1] H Peter Anvin. The mathematics of raid-6, 2007.

[2] Emil Artin and Arthur Norton Milgram. *Galois theory*, volume 2. Courier Corporation, 1998.

[3] Patrick McClanahan. Raid. https://eng.libretexts.org/@go/page/82927. Accessed: 2024-09-19.

[4] 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*, SIGMOD '88, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery.

[5] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[6] Wikipedia contributors. Raid — Wikipedia, the free encyclopedia, 2024. [Online; accessed 19-September-2024].