Research on improving disk throughput in EC-based distributed file system

Dong-Jin Shin[1], Jeong-Joon Kim[2]*

[1] Department of Computer Engineering, Anyang University, South Korea
[2] Assistant Professor, Department of ICT Convergence Engineering, Anyang University, South Korea
Email: [1]djshin@ayum.anyang.ac.kr, [2]jjkim@anyang.ac.kr

ABSTRACT

The development of the Fourth Industrial Revolution resulted in an increase in data type and size, and distributed file systems emerged to store them. Among them, Replication techniques divide the data that you want to store into certain blocks and replicate the divided blocks to store them distributed across multiple nodes. However, there was a problem with increasing the disk's capacity to store the replicated blocks. Thus, the Erasure Coding technique emerged, and the EC-based distributed file system improved the space efficiency issue over the Replication technique because it creates blocks to be stored through encoding and parity blocks to be used for recovery. However, EC-based distributed file system has caused disk disk write throughput problems to access a number of disks, causing system performance degradation. Therefore, this paper proposes Buffering and Combining techniques to improve disk write throughput problems in EC-based distributed file systems.

Keywords

Disk Write Throughput, Big Data, Erasure Coding, HDFS
Article Received: 10 August 2020, Revised: 25 October 2020, Accepted: 18 November 2020

Introduction

In the distributed file system, Hadoop is a representative method of distributing and storing data to be stored. Hadoop has two types of distributed file storage technology and parallel processing technology, and in this paper, only distributed file storage technology is mentioned. Hadoop's distributed file storage is called HDFS (Hadoop Distributed File System), and it uses a replication technique that divides the data to be stored into blocks of a certain size, replicates and stores it [1].

However, the replication technique requires a large disk to store copies of divided blocks. In particular, for companies that store and process large amounts of data, large costs are incurred in building and managing because the scale of the system increases a lot. To solve this space efficiency problem, the erasure coding technique (hereinafter referred to as EC) has begun to be applied to HDFS [2,3].

In the EC technique, original data is striped and stored into K data cells and M parity cells through encoding. In the replication technique, blocks are replicated and stored, but distributed storage through the encoding of the EC technique is superior to the replication technique because only parity cells are added from the existing data cells [4].

However, by encoding, data cells and parity cells are distributed stored across multiple data nodes, resulting in one disk data switching to multiple smaller Cells. At this point, a number of small Cell can result in a performance degradation of the overall system, the larger the volume of K and M that produces data cells and parity cells, and the smaller the stripping size, the lower the disk write throughput performance [5,6].

Starting with the introduction, this paper explores the replication techniques and EC techniques used in HDFS, and the performance degradation factors that occur in EC-based HDFS through two related studies. Chapter 3 introduces solutions to disk write throughput problems that arise during encoding. Chapter 4 compares and analyzes existing EC-based HDFS and systems that apply solutions and concludes this paper with the conclusion of Chapter 5.

Related Works

2.1 Replication based distributed file system structure

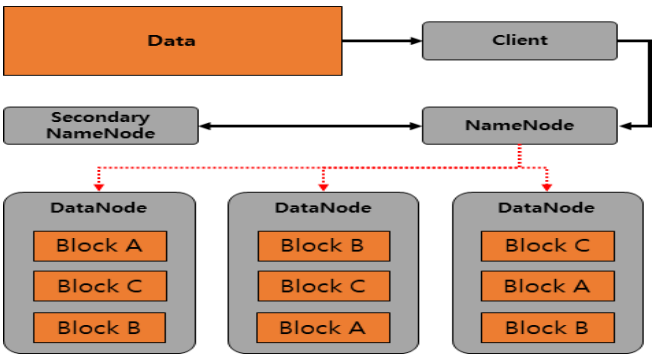

Fig 1. HDFS Basic Structure

Figure 1 illustrates the structure of a replication-based distributed file system used in Hadoop. Hadoop has HDFS(Hadoop Distributed File System) technology that distributes and stores large amounts of data and Map-Reduce technology that supports distributed parallelism. In this paper, we focus only on HDFS.

The process in which the data is stored is first ordered by the client to store the data in HDFS, and then the NameNode divides the data entered by the client into 128 megabytes blocks, which are designated as default values. When Hadoop is configured, the value set when dividing the initial block is 128 megabytes, which can be modified to the size desired by the user [7].

However, if more data is stored, it is necessary to increase the space on the physical disk to store the data. In other words, as the number of replicated blocks increases, the spatial efficiency of data storage is reduced.

For example, in Figure 1, we split one data into three blocks to store it and stored it on multiple data nodes. However, assuming that one data has a 100% spatial efficiency, we divide it into blocks and store two more data through replication, which eventually requires an additional 200% spatial efficiency. To improve this, EC techniques emerged and began to be applied to distributed file systems.

2.2 Erasure Coding based distributed file system structure

In a replication-based distributed file system, we use a simple and convenient storage method using block replication, but we look at the disadvantages of increasing data reducing space efficiency and increasing maintenance costs for the file system itself. The EC techniques used six of the RAID levels to improve the existing RAID techniques, and the structure of the RAID level 6 was shown in Figure 2.

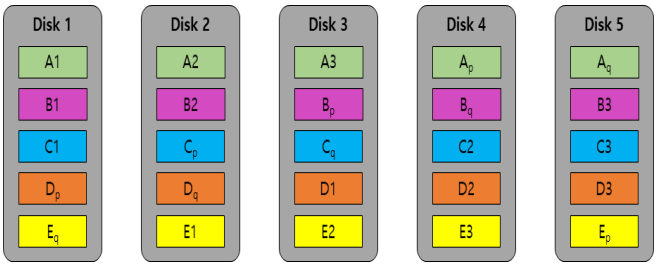

Fig 2. Hard Disk RAID 6 Level Storage Structure

Figure 2 shows a RAID 6 level storage structure that is divided into five disks when files A, B, C, D, and E are present. Data blocks A1, A2 and A3 that make up file A are stored on different disks due to RAID 6 level techniques, and two parity blocks, Ap and Aq, are configured in case of problems with the data blocks that make up file A. This parity block is stored on different disks to avoid overlapping with the data block, and the original data can be recovered by XOR operations of the parity block and the data block when the source data block is in trouble [8].

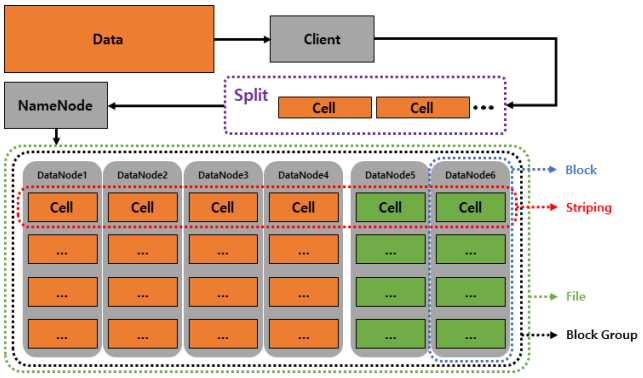

Fig 3. 4+2 EC Volume Storage Structure

Figure 3 illustrates the 4+2 EC volume storage structure of a data node storing four data cells and a data node storing two parity cells. When a client commands the data to be stored, the data is not replicated in EC-based HDFS but divided into data cells through the Split process. We then encode stored data cells to generate parity cells, which are used for recovery [8].

In other words, data nodes 1 through 4 represent data nodes that are stored by fragmenting the original data they want to store into data cells, while data nodes 5 and 6 represent data nodes that store parity blocks generated by encoding data cells.

Stripping refers to a set of data cells and parity cells associated with a single encoding operation as an encoding unit. A block is a file stored on each data node in a unit of storage. A block group refers to a set of blocks where a single stripping is divided and stored, and one file consists of one or more block groups [9].

2.3 EC-based HDFS System Structure Degradation Factors

This section provides a detailed look at the disk in/out problems encountered with EC-based HDFS when storing data. We first describe the basic of write process in EC-based HDFS.

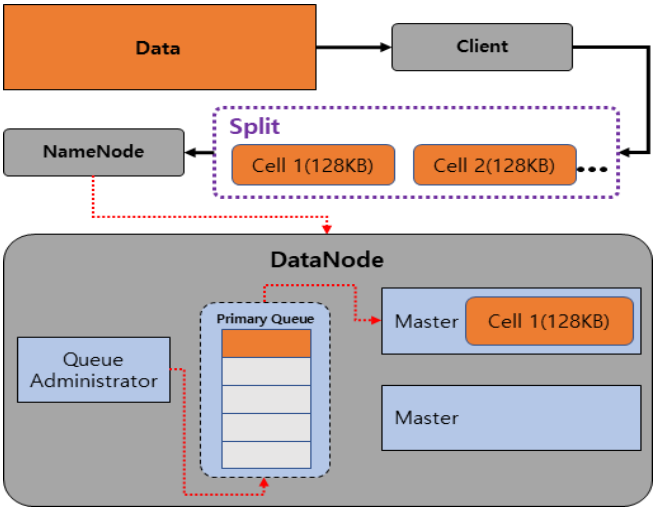

Fig 4. Single write process for EC-based HDFS

Figure 4 shows the basic process of EC-based HDFS and shows the process of importing a file layout from a client through a name node to process write through a data node. File layout refers to file configuration information about the data that the client wants to store. In addition, the data node has a Queue Administrator and a worker to perform encoding services to accept requested writes and reads, and the worker is changed to Master and Slave as needed.

An EC-based HDFS client sends a data storage write request to the name node. The Queue Administrator on the data node puts the write that occurs when a client requests it into the Primary Queue. There are two workers in Figure 4, but there are many workers in the data node. Many workers take events from the Primary Queue and call the appropriate service function during Master or Slave upon their request, processing them, and returning the results. Two requests (Cell 1, Cell 2) have occurred, but only one is processed because a single write request is dedicated to one worker for concurrency control.

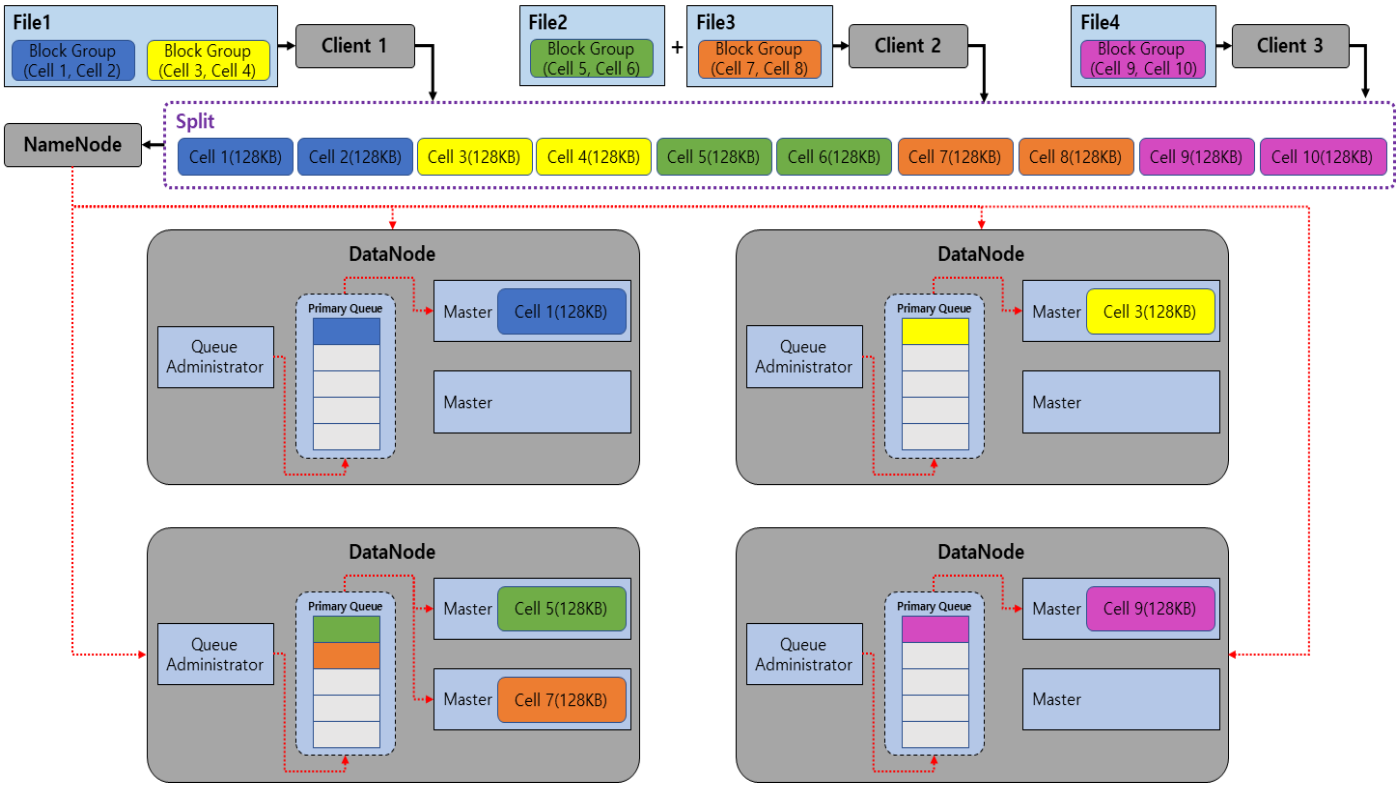

Fig 5. 4+2 Multiple write process for EC-based HDFS

Different block groups can be processed simultaneously, as illustrated in Figure 5. However, as mentioned in Figure 4, requests for the same block group for concurrency control are dedicated to one worker, so only a total of four workers are allowed to operate, and the rest are stacked in Primary Queue.

Therefore, if there is a lot of write for minority files, the primary queue fails to process, resulting in more waiting.

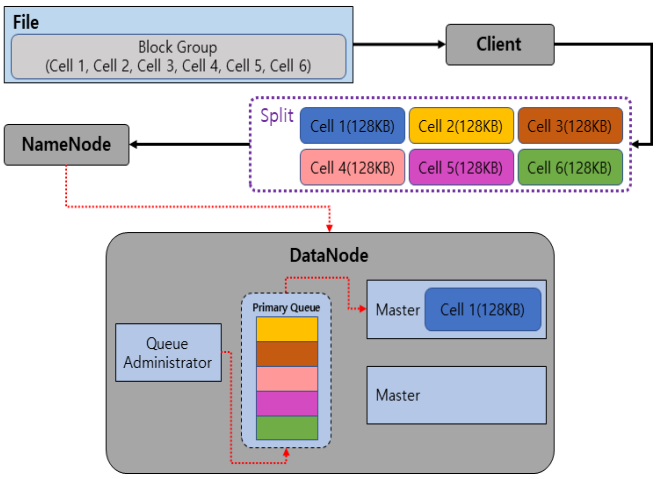

Fig 6. EC-based HDFS queue processing delay

Figure 6 shows an example consisting of a single block group for File1, and since it is a single block group, the representation of the block group is omitted. It shows that write processing is not handled due to internal processing delay problems and is accumulated in Primary Queue, which can cause overall system performance degradation.

Efficient Data Storage Plan

This chapter introduces Buffering and Combining measures to improve the problem of write degradation arising from EC-based HDFS as described in Chapter 2.

3.1 Buffering

The Buffering step refers to the step of creating a secondary queue for the block group processing write, rather than waiting for the same basic block group write request during write processing. Figure 7 shows the processing procedure of the Buffering step.

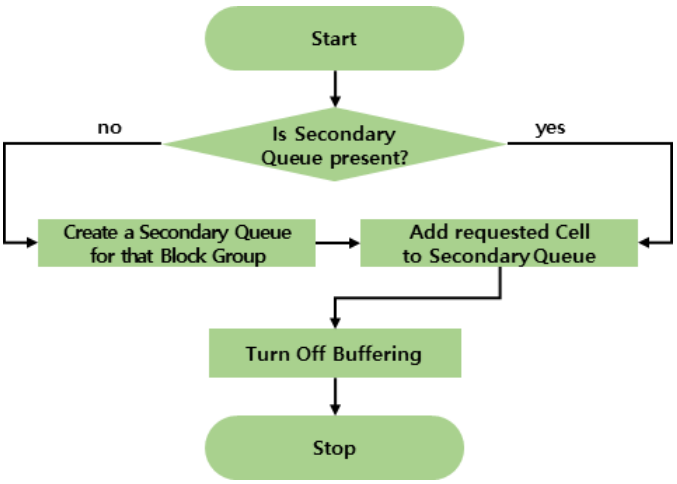

Fig 7. Buffering Steps

The Buffering process first verifies that secondary queue exists. If not present, create a secondary queue and add the standby Cell to the secondary queue. If Secondary Queue exists, add the requested Cell to the generated Secondary Queue. When the add-on is complete, finally turn off the Secondary Queue setting. Buffering can minimize the waiting Cell to Primary Queue, thus preventing system performance degradation. Figure 8 shows the processing of the Buffering step.

The Buffering step allows other workers to process the same file without waiting for Cell to be requested during the first Cell of the file. This means that Master1 registers to perform Cell 1 for File, and while Master2 is performing this, Master2 is called to import and process Cell 2 for File. At this time, register that Master2 is also buffering Cell for the File. Master2 checks the file information being processed and confirms that Master1 handles Cell 1 for File, so it creates a secondary queue associated with the file and registers write2.

The interruption in Figure 8 shows Master2 putting Cell 2 in the secondary queue and taking and processing the next Cell 3 in the primary queue. At this time, Cell 2 is waiting in Secondary Queue. The bottom of Figure 8 shows the next write Cell 4 and Cell 5, taken from Primary Queue and processed. In Secondary Queue, you can see that Cell 2, Cell 3, Cell 4 are stacked for File.

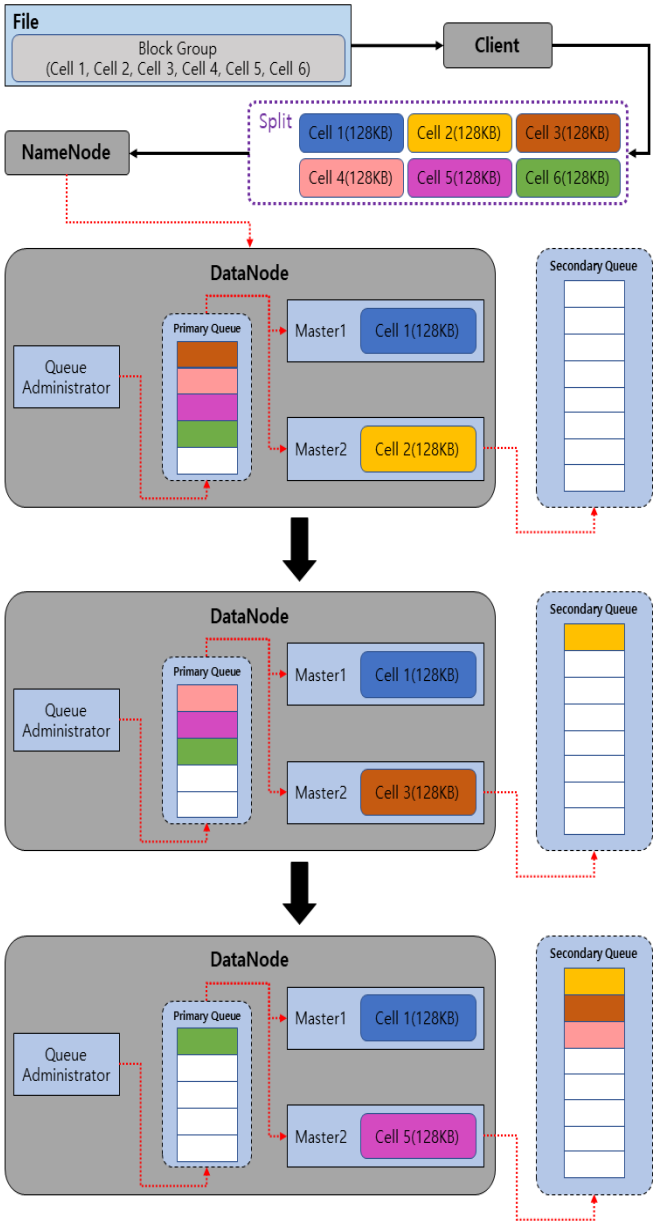

Fig 8. Buffering Step Processing Appearance

3.2 Combining

The Combining step is not treated as a single write unit when processing in the Worker. This refers to the step of merging Cells accumulated in the secondary queue through Buffering and processing it as one Cell. Figure 9 shows the processing procedure of the Combining step.

First of all, check if there are more than two Cell in the Primary Queue. If only one Cell exists after verification, the primary queue acquires the standby Cell, performs one Cell processing, and returns the processing result. After the return, the step is taken to verify that the data is buffered.

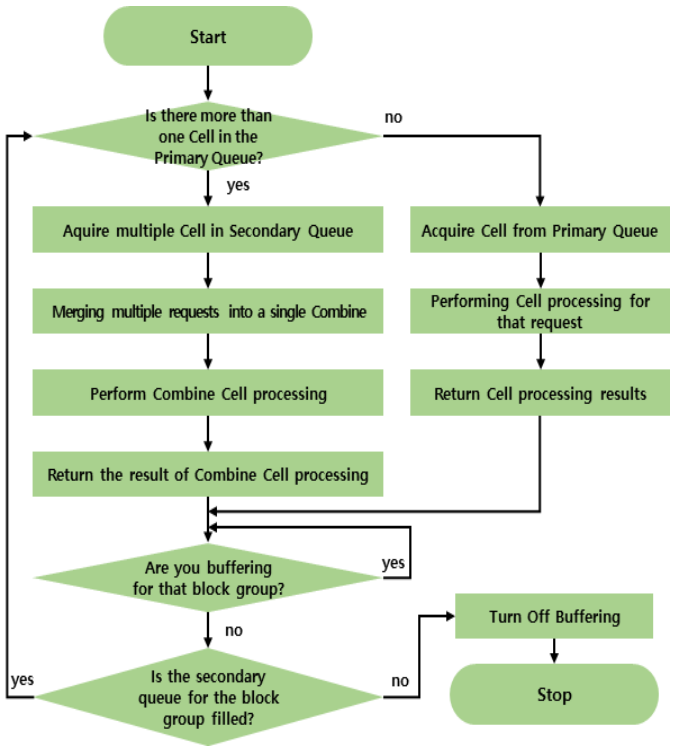

Fig 9. Combining Steps

If there is more than one Cell in Primary Queue, multiple Cells are obtained from Secondary Queue and merged into one using the Combine technique. The Combine is generated and the Combine processing is performed to return the processing result. Verify that buffering is in progress for the next corresponding block group and wait until the Buffering process is over. If it is not buffering, make sure that the secondary queue is filled, and if it is filled, perform Combine again. If the Secondary Queue is not populated, un-set that the block group is processing and exit processing.

Figure 10 shows the process of Buffering and Combining after Master2 completes the processing of Cell 1 owned by Master1, returning the results.

Figure 10 1) confirms that Master1 is Buffering for the corresponding block group after returning the processing results for Cell 1, so Master1 waits for Buffering to complete. Figure 10 2) shows Master1 checking secondary queue after Master2's Buffering ends, and if multiple Cells are stacked in secondary queue, Combine them with a specified unit and treat them as one Cell.

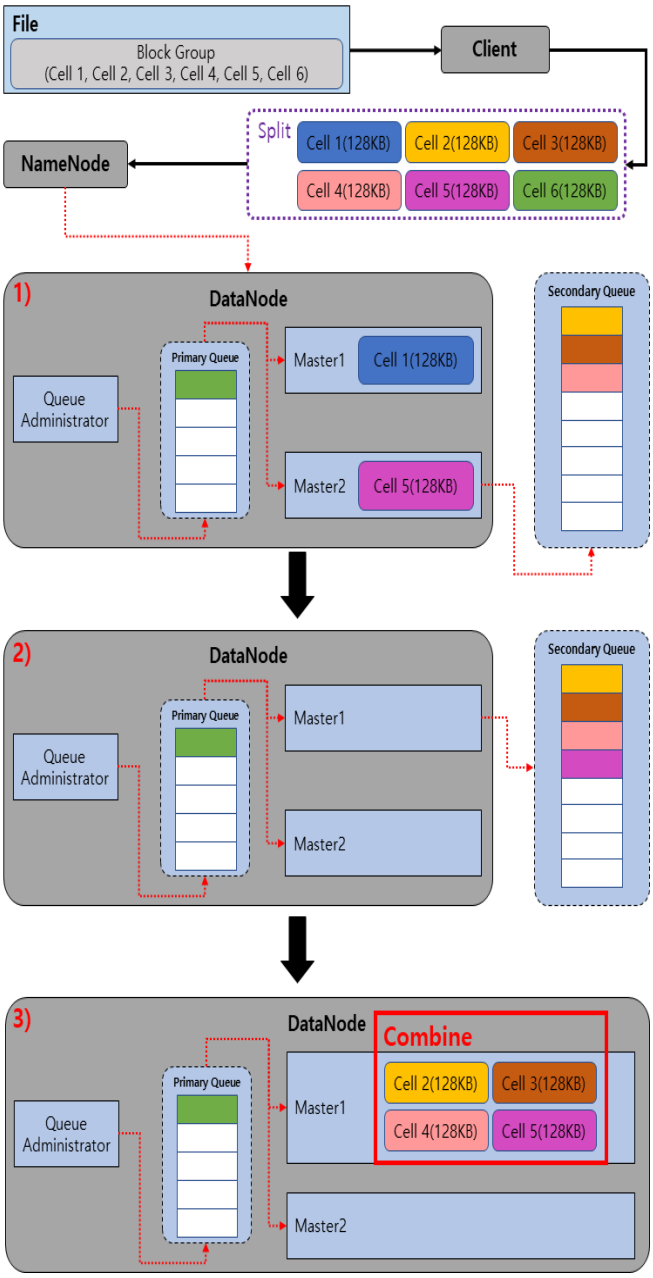

Fig 10. Combining Step Processing Appearance

Figure 10 3) shows that four Cells are taken from the Secondary Queue and processed by integrating them into one Combine Cell. Therefore, Cell Combining allows multiple Cells to be treated as one, increasing Cell size, and decreasing the number of Cells, significantly improving write efficiency by reducing overall network load and contention. Figure 11 shows the core part of the code applied with Buffering and Combining.

```
BEGIN

 Metadata METADATA ←get split
BlockGroup FILE_LAYOUT
 Master#1← from Request Cell by
PRIMARY_QUEUE

 IF (PRIMARY_QUEUE > 2 more Request
Cell)
  Call MASTER#2
  MASTER#2 ← Create
SECONDARY_QUEUE (Buffering START)
    SECONDARY_QUEUE[Used] ← 1
    add Request Cell ← Delay Request Cell in
PRIMARY_QUEUE
    Buffering FINISH & Combine Processing
    Result RETURN
 ELSE
  add Request Cell ← MASTER#1
  Single Processing
  Result RETURN
 END IF

 IF (Check Buffering to BlockGroup)
  Go To 6) Line
 END IF

 IF (SECONDARY_QUEUE == EMPTY)
  SECONDARY_QUEUE[Used] ← 0
  Buffering CANCEL
 END IF

END
```

Fig 11. Combining Core Code

(1–4) When a client stores a file to store, it divides the file into groups of blocks of a certain size. Through the file layout process, metadata is recorded for each block group and passed to the Master of the data node.

(6–12) The Master checks the Queue Administrator for at least two requested Cells in the Primary Queue. If there is more than one, the other Master is called within the data node, and the called Master generates the Secondary Queue, recording "1") that the Secondary Queue is in use. In addition, the called Master proceeds with the Buffering process, which adds standby Cell to the Primary Queue. When the addition is completed, the Cell Combining is converted to one cell, and the result is returned.

(13–17) If there are fewer than two requested Cells (one Cell), the first corresponding Master takes Cells, processes them, and returns the results.

(19–28) After the Combining process, if Buffering is in progress for the block group, go to 8) and perform it again. If there is no Cell in the secondary queue, write ("0") that the secondary queue is not in use, and the Buffering is terminated.

Comparison Analysis

In this chapter, performance comparisons are presented under the name EC HDFS-BC when basic EC-based HDFS Buffering technique proposed in this paper, and Combining technique are applied. It was created using the time command to measure the operation time of the command in Ubuntu and the dd command to generate sample data.

Experiments are performed on the name node using the command time ddif=/dev/zero of=/test/50G bs=128k count=409600. A 128k block was performed 409,600 times (128k*819,200 = 104,857,600kb (50G)) to measure the throughput required to store 50 gigabyte files under the 50G name in the /test/ path of the EC HDFS-BC system.

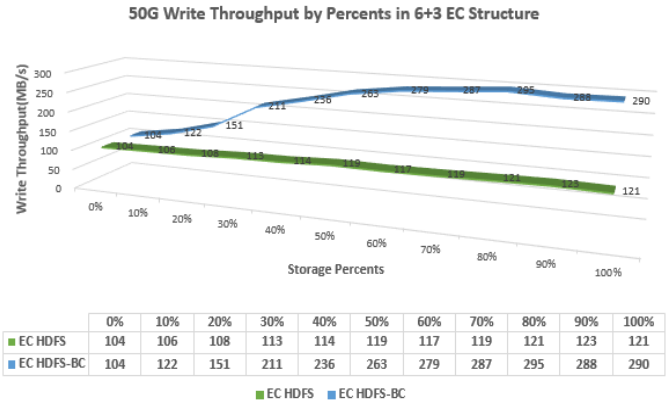

Fig 12. Comparison of EC-based HDFS with EC-HDFS BS data throughput

Figure 12 shows the throughput of the disk when stored in EC-based HDFS and EC HDFS-BC by randomly generating 50 gigabyte of sample data for EC-based HDFS and EC HDFS-BC systems. In other words, the process of storing 50 gigabytes of files was classified into 10 units of percentage and repeatedly measured disk throughput over time when stored from 0% to 10% or from 10% to 20%.

Basic EC-based HDFS shows slight performance gains, but little change and EC HDFS-BC systems show higher write processing as file storage approaches completion, with approximately 2.5 times more disk processing performance than EC-based HDFS.

The following is done 409,600 times (128k*409,600 = 52,428,800kb (50G)) with the command time ddif=/dev/zero of=/test/50G bs=128k count=409600. We measured the time it would take to store a 10 gigabytes file under the 50G name in the /test/ directory.

In addition, a 100 gigabytes file was created in the same way as above, and the time taken to store 50 GB and 100 GB files was compared.

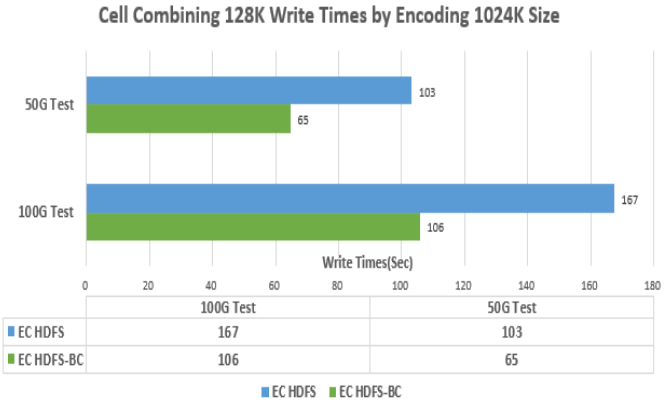

Fig 13. Storage Time Comparison by File Size

Figure 13 shows a performance comparison when storing sample data in different sizes of EC HDFS and EC HDFS-BC systems. EC-based HDFS allows encoding to be resized to the desired size. By default, we store data cells on a 1024K basis (RS-Default 1024K) when using Reed-Solomon. Thus, the encoding size was set to 1024K and the Cell combining size to 128K.

When storing data of 50 gigabytes of sample data, EC-based HDFS was 103 seconds and EC HDFS-BC was 65 seconds, showing a performance improvement of about 1.5 times. When storing 100 gigabytes of data, EC-based HDFS is 167 seconds and EC HDBS-BC is 106 seconds, which is about 1.6 times faster.

Thus, we can see that although the storage speed is faster when storing data with the same encoding size and different data sizes, there is no change in EC HDFS-BC that increases the storage speed as the data grows.

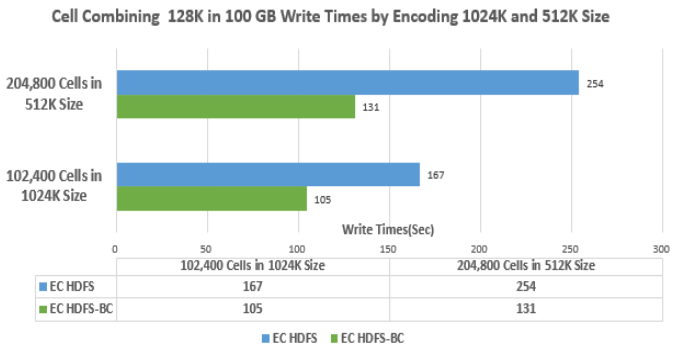

Fig 14. Storage Time Comparison by Encoding Size

Figure 14 shows a performance comparison with encoding size when storing sample data. This experiment shows performance comparison when encoding units are set to 1024K, and 512K when storing 100 gigabytes of data, and the Combine size of EC HDFS–BC is set to 128K.

When 100 gigabytes of sample data were stored when encoding in 1024K size, EC-based HDFS was 167 seconds and EC HDFS-BC was 105 seconds, which was about 1.6 times faster. When encoding with a smaller 512K size, EC-based HDFS is 221 seconds and EC HDFS-BC is 131 seconds, which is about twice as fast.

Although it is the same sample data size, the smaller the encoding, the greater the performance improvement. This is because EC HDFS-BC processes once through the Combination process when processing small and large numbers of Cells.

Conclusion

In this paper, we introduce problems arising from Hadoop, a storage solution, when storing data, and propose ways to improve them. Among the various storage techniques, EC-based distributed file systems have higher spatial efficiency compared to replication techniques because they are generated and stored as parity cells through encoding. However, the disk throughput load that occurs when saving files in an EC-based distributed file system and the ability to access many disks when recovering files significantly reduces performance. Therefore, we propose an efficient file storage and recovery method by selecting HDFS, one of the EC-based distributed file systems. File storage leverages Buffering and Combining techniques, resulting in approximately 1.6x performance improvement over conventional HDFS.

However, memory usage problems exist while generating Secondary Queue on other masters to perform Buffering. In other words, there is a situation where there is a lack of memory space for other applications while maintaining memory for a long time, which has the disadvantage of increasing the overall memory capacity. Therefore, it is planning to further study ways to solve memory-related problems of Buffering in the future.

References

[1] B.G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M.F. Kaashoek, J. Kubiatowicz, R. Morris (2006) Efficient Replica Maintenance for Distributed Storage Systems. NSDI '06 Proceedings of the 3rd conference on Networked Systems Design & Implementation 6:45-58

[2] J. Li, B. Li (2013) Erasure coding for cloud storage systems: A survey. Tsinghua Science and Technology 18(3):259-272

[3] J.D Cook, R. Primmer, A. de Kwant (2014) Compare Cost and Performance of Replication and Erasure Coding. Hitachi Review 63:304-310

[4] D.O. Kim, H.Y. Kim, Y.K. Kim, J.J Kim (2019) Cost analysis of erasure coding for exa-scale storage. The Journal of Super Computing 75(8):4638-4656

[5] D. Sun, Y. Xu, Y. Li, S. Wu, C. Tian (2016) Efficient Parity Update for Scaling RAID-like Storage Systems Networking. Architecture and Storage (NAS), 2016's IEEE International Conference 1-10

[6] L.J. Mohan, R.L. Harold, P.I.S. Caneleo, U. Parampalli, A. Harwood (2015) Benchmarking the Performance of Hadoop Triple Replication and Erasure Coding on A Nation-Wide Distributed Cloud. Network Coding (NetCod), 2015's International Symposium 61-65

[7] Dong-Jin Shin, Kwang-Jin Kwak, Seung-Yeon Hwang, Jeong-Min Park, Jeong-Joon Kim (2018) Efficient Storage Structure Research in Hadoop Distributed Storage System. 2018's IPACT Conference 56-57

[8] Dong-Jin Shin, Seung-Yeon Hwang, Kwang-Jin Kwak, Kyoung-Won Park, Jeong-Min Park, Jeong-Joon Kim (2019) A Study on the Recovery Techniques of Distributed File System in a Big Data Environment. 2019's IIBC Conference 83-86

[9] K. Nansai, X. Chen, S. Chen, J. Zang (2019) HDFS Erasure Coding in Production. CLOUDERA Blog