http://dx.doi.org/10.5392/IJoC.2014.10.1.062

Efficient Update Method for Cloud Storage System

Ki-Jeong Khill
Department of Computer Engineering
Korea National University of Transportation 50 Daehak-ro, Chungju-si, 380-702, Korea

Sang-Min Lee, Young-Kyun Kim
Department of Cloud Computing, Storage System laboratory
Electronics and Telecommunications Research Institute (ETRI), 218 Gajeong-ro, Daejeon Korea

Jaeryong Shin
Department of Health Administration
Gwangju Health University, 73, Bukmun-daero 419, Gwangsan-gu, Gwangju, 506-701, Korea

Seokil Song
Department of Computer Engineering
Korea National University of Transportation, 50 Daehak-ro, Chungju-si, 380-702, Korea

ABSTRACT

Usually, cloud storage systems are developed based on DFS (Distributed File System) for scalability and reliability reasons. DFSs are designed to improve throughput than IO response time, and therefore, they are appropriate for batch processing jobs. Recently, cloud storage systems have been used for update intensive applications such as OLTP and so on. However, in DFSs, in-place update operations are not carefully considered. Therefore, when updates are frequent, I/O performance of DFSs are degraded significantly. DFSs with RAID techniques have been proposed to improve their performance and reliability. Their performance degradation caused by frequent update operations can be more significant. In this paper, we propose an in-place update method for DFS RAID exploiting a differential logging technique. The proposed method reduces the I/O costs, network traffic and XOR operation costs for RAID. We demonstrate the efficiency of our proposed in-place update method through various experiments.

Key words: *DFS, RAID, In-place update, Differential logging.*

1. INTRODUCTION

Existing storage systems are limited for cloud computing when storing large amounts of data in a reliable manner and providing a fast I/O speed. A DFS (distributed file system) has been introduced to solve these problems with a low cost. However, DFS replicates the data three times to guarantee reliability, which incurs high storage overheads [1].

RAID based DFSs have been proposed in [5], [6] to address the storage overheads problem but it may cause performance degradation due to the lack of load balancing when multiple clients access data at the same time, as well as the costs of running RAID.

In DFS, a high processing capacity is more important than data access speed improvement because it was designed for batch processing. Therefore, most DFSs do not support in-place updates. Instead of in-place updates, the original data is not updated and a new chunk is recorded in another server while the existing chunk is deleted for processing. This method can degrade the I/O performance significantly in an environment where in-place updates occur frequently.

This can cause further problems in DFS if RAID is applied. A DFS with RAID incurs operational overheads because it requires additional RAIDing when the data is recorded and again when an in-place update occurs. This can be a serious problem in applications where in-place updates occur frequently.

In this paper, a differential logging (*D-Log*) based in-place update technique is proposed for DFS RAID. The proposed method generates a small *D-Log* updated part of a chunk when an in-place update occurs in a DFS where RAID is applied and it updates the parity using *D-Log.* The proposed in-place update technique can reduce the I/O cost and the XOR operation cost, as well as preventing data loss due to system failure.

* *Corresponding author, Email: sisong@chungju.ac.kr*
Manuscript received Mar. 06, 2014; revised Mar. 14, 2014; accepted Mar. 21, 2014

This paper is organized as follows. Chapter 2 describes related work, and Chapter 3 explains our proposed *D-Log* based in-place update technique. Chapter 4 presents an evaluation of the performance of the proposed method based on a simulation. Finally, chapter 5 gives our conclusion and outlines future work.

2. RELATED WORK

HDFS [2] is open source distributed file system in Hadoop and is highly similar to GFS [9]. HDFS supports write-once-read-many semantics on files. Each HDFS cluster consists of a single name node (metadata node) and a usually large number of data nodes. HDFS replicates files three times to handle system failures or network partitions. Files are divided into blocks, and each stored on a data node. Each data node manages all file data stored on its persistent storage.

It handles read and write requests from clients and performs "make a replica" requests from the name node. There is a background process in HDFS that periodically checks for missing blocks and, if found, assigns a data node to replicate the block having too few copies [3].

DiskReduce[1], [3] is a RAID technique for a HDFS. It supports RAID 5+1 (RAID 5 and mirror) and RAID 6. RAID 5+1 uses two copies of data and a single parity so it can be recovered to allow normal operation even when two disk failures occur. It is designed to minimize the change to original HDFS. It takes advantage of following two important features of HDFS. First, files are immutable after they are written to the system and second, all blocks in a file are replicated three times initially.

Its file commit and replication are same to those of HDFS. However, it exploits the background re-replication of HDFS in a different way. While HDFS processes to look for insufficient number of copies in background, DiskReduce looks for blocks with high overhead which are replicated blocks that can be turned into blocks with lower overhead (i.e. RAID encoding).

Two methods can be employed in DiskReduce, depending on how the chunks are grouped. The first approach is the "within a file" method where RAIDing occurs in a large file. The second is an "across-files" method where RAIDing occurs regardless of the file size. The across-files method can reduce the storage overheads better than the within a file method. The method proposed in this paper can use both approaches.

DiskReduce uses the Erasure code as a RAIDing algorithm. The Erasure code was proposed to improve the reliability of computer communication [4]. This high encoding and decoding performance of the Erasure code means it has been applied to RAID and many other proposed coding techniques, such as Reed-Solomon, Liberation, and Liber8tion.

3. PROPOSED IN-PLACE UPDATES TECHNIQUE

The proposed method is based on *D-Log*. *D-Log* is obtained by applying an XOR operation to data prior to an update and to the data after an update. *D-Log* is small because it only uses an updated region in a chunk. The storage overheads or network traffic can be reduced if updates and parity updates are carried out in this manner. We explain the in-place update method based on *D-Log* by dividing it into updates in a normal state and updates in a failure state.

3.1 In-place update method in a normal state

In a normal state, the in-place update generates the *D-Log* for an updated part and sends it to a node where parity is present to update the parity. Therefore, we need to show that the previous parity can be updated to the new parity using the *D-Log*.

Theory 1. Using the updated data D_k and data prior to the update D_k the parity C_i encoded by the Erasure code can be updated to the new parity C_i, where D_k and D_k indicate the κ-th data chunk in a stripe and C_i and C_i indicates κ-th parity.

$$D \cdot Log_k = D_k \oplus D_{k'} \tag{1}$$

∑ With regard to ∑, assuming $\sum_{i=0}^{k-1} D_i = D_0 \oplus D_1 \oplus \cdots D_{K-1}$, D_i indicates the i-th chunk.

$$C_i = \sum_{i=0}^{K-1} X_{i,i} D_i \tag{2}$$

$$C_i = \sum_{i=0}^{K-1} X_{i,i} D_i \oplus X_{ik} D_k \oplus \sum_{i=k+1}^{K-1} X_{i,i} D_i \tag{3}$$

C_i is calculated using Equation (2) and C_i is calculated using Equation (3). C_i and C_i are XORed so they yield the following equation, where X_i is a sub-matrix of the bit matrix F .

$$\begin{aligned} C_i \oplus C_i{}' &= \sum_{i=0}^{K-1} X_{i,i} D_i \oplus \sum_{i=0}^{K-1} X_{i,i} D_i \oplus X_{i,k} D_k{}' \oplus \sum_{i=k+1}^{K-1} X_{i,i} D_i \\ &= X_{ik}(D_k \oplus D_k{}') \\ \therefore C_i{}' &= X_{i,k} D \cdot Log_k \oplus C_i \end{aligned} \tag{4}$$

Thus, $C_i{}'$ can be obtained by XOR of C_i and $D \cdot Log$.

Fig. 1 shows the process of an in-place update in a normal state. A client writes a new piece of data to a server DS1 where data is already present. DS1 performs an XOR operation on two pieces of data of before and after an update to generate $D \cdot Log$ The generated $D \cdot Log$ is sent to servers DS4 and DS5 where the parity is stored, and they updates the parity using $D \cdot Log$.

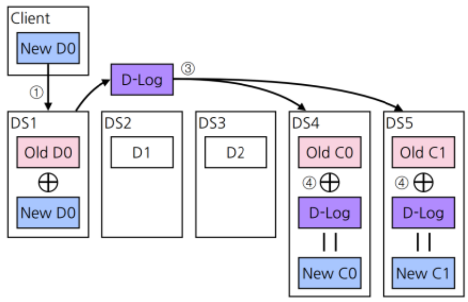


Fig. 1. In-place update in a normal state

Fig. 2 shows the parity update algorithm. In immediate parity update algorithm of Fig. 2 reads data D and updates parity immediately. Algorithm 2 of Fig. 3 shows the delayed parity update operation. In this algorithm, *D-log* is written to storage, and then parity is updated with the stored *D-log* later.

Algorithm 1: Immediate update of parity
for (i = 0; i < n; ++i) {
  Read D_i
  $D - Log_k = D_k \oplus D_k{}'$
  for (j = 0; j < m; ++j) {
    Read C
    $C_i = X_K D - Log_k \oplus C$
    Write C_i
  }
}

Algorithm 2 : Delayed update of parity
for (i = 0; i < n; ++i) {
  Read D_i
  $D - Log_k = D_k \oplus D_k{}'$
  for (j = 0; j < m; ++j) {
    Write $D - Log_k$
  }
}

for (j = 0; j < m; ++j) {
  Read C
  for (i = 0; i < n; ++i) {
    Read $D - Log_k$
    $C_i = X_K D - Log_k \oplus C$
  }
  Write C_i
}

Fig. 2. Parity update algorithms in normal state

In this paper, an in-place update method based on *D-Log* is proposed using the following three methods. The first method is an immediate parity update, which is shown as Algorithm 1 in Fig. 2. The second method is a delayed update after *D-Log* is stored in a local file and fetched later. The final method combines these two methods. When a file is stored in a local disk, the file name specifies a corresponding chunk ID and timestamp so the requisite *D-Log* can be found easily later.

3.2 Write method in a failure state

The proposed in-place updates method uses *D-Log*. However, *D-Log* cannot be generated if the κ-th data chunk cannot be read in the same stripe because of a DS failure or a network partition problem. Therefore, to generate the *D-Log*, the data related to a node that cannot be accessed should be restored and followed by a parity update, as shown in Algorithm 3 in Figure 4.

However, many operations are required to restore data for a node that is not accessible. Without restoring the data, a new parity is created using the new data and other stored data, and a method for generating *D-Log* using the parity is proposed in this paper. Next, we show that parity can be generated using data that excludes inaccessible data and using this *D-Log*.

Theory 2. If the DS storing the κ-th data has a failure, a new parity C_i is generated using the new data $D_k{}'$, which is stored in other DS, $D_{0>} \cdots D_{K-1>} D_{k+1>} \cdots D_{K-1}$, thereby generating *D-Log*$_k$ using $C_i{}'$and the existing parity C_i.

Proof: According to Equations (5) and (6), *D-Log*$_k$ is calculated as follows, so *D-Log*$_k$ can be calculated if C_i and $C_i{}'$ are available.

$$\begin{aligned} D - Log_K &= (X_{i,k})^{-1}(C_i{}' \oplus C_i) \\ &= (X_{i,k})^{-1}(X_{i,k} D_k{}' \oplus X_{i,k} D_K) \qquad (5) \\ &= D_k \oplus D_k{}' \\ \therefore D - Log_K &= (X_{i,k})^{-1}(C_i{}' \oplus C_i) \end{aligned}$$

When a C_1 is encoded or decoded where i is one or more in the Erasure code, the C_0 operation is required, which is more complex than F. This can affect Equation (5). Therefore, we focus on C_0 to calculate the parity when creating the *D-Log* for a write method in a failure state.

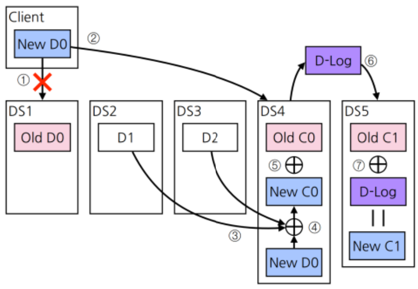


Fig. 3. Write method in a failure state

Fig. 3 shows the process used for a parity update in a failure state. Initially, a client approaches DS1, where the existing data is stored, to send the data. However, DS1, where the existing data is stored, is not available due to a failure. Thus, the client sends updated data to DS4 where the parity F is stored.

Algorithm 3 : Recover data and update parity
Read $D_{b>} \cdots D_{c-1>} D_{c+1>} \cdots D_{c-1} C_k$
Decoding D_i
$D - Log_k = D_k \oplus D_{k'}$
$C_k{}' = X_K D - Log_k \oplus C_k$
Write $C_k{}'$
for (i = 0; i < m; ++i) {
  if i not equal a {
    Read C_i
    Update parity C_i $C_i{}' = X_K D - Log_k \oplus C_k$
    Write $C_i{}'$
  }
}

Algorithm 4 : Update parity in failure state
Read $D_{b>} \cdots D_{c-1>} D_{c+1>} \cdots D_{c-1} C_k$
Encoding $C_k{}'$
$D - Log_K = (X_{i,k})^{-1}(C_k{}' \oplus C_k)$

```
Write C_k'
for (i = 0; i < m; ++i) {
   if i not equal a {
      Read C_i
      Update parity C_i' = X_K D − Log_k ⊕ C_k
      Write C_i'
   }
}
```

Fig. 4. Parity update algorithms in a failure state

DFS then performs an XOR operation on the updated data and other data to create a new parity. It reads the existing parity to perform an XOR operation with the new parity to generate *D-Log*, as well as updating the other parities. Algorithm 4 in Fig. 5 shows the parity update method in a failure state.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Experimental environment

We evaluated the performance of the proposed method using a simulation. This experiment compared the performance of the proposed methods and an existing method in normal and failure states. The simulator was implemented using gcc 4.2.1 in the Mac OS X 10.8.2 environment.

The parameters used in the simulation are shown in Table 1. The chunk size was set as 64 MB, which is generally used in most DFSs. The number of in-place updates was set to 100 times per stripe and the size of the update ranged between 1 MB and 8 MB.

Table 1. Parameters used in the simulation

Parameter	Value
Chunk size	64 MB
Stripe configuration	Data = 4, Parity = 2
Word size	8
Number of update operations	100 times per stripe
Update operation size	1–8 MB
Number of nodes	6

The number of nodes used in the simulation was fixed as six, one stripe comprised data = 4 and parity = 2, and the word size was set as 8. The coding technique used the Liber8Tion code. The simulation did not consider the network traffic or the number of clients. However, I/O was divided into local and remote to allow measurements to be made.

An in-place method has been proposed previously, so the performance evaluation compared the following five methods; a method for performing encoding again (None); our proposed method, which reflected $F(C_0)$ and $C_x(C_1)$ immediately (Immediate P and Q), a method that reflected F only (Immediate P Lazy Q),; a method that reflected C_x only (Lazy P Immediate Q); and a method that reflecting F and C_x at a later time (Lazy P and Q).

4.2 In-place update method in a normal state

Fig. 5 shows a comparison of the result of XOR computing based on the number of in-place updates in a normal state. This figure shows that the proposed algorithm was much more effective than performing encoding again (None = 52.34 GB). The proposed methods had the same number of XOR operations.

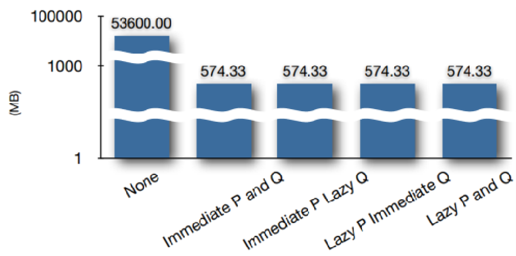


Fig. 5. Comparison of XOR computing using five methods and in-place updates

In the proposed method, C_x was updated using $D \cdot Log$ only, so the minimum number of I/O updates required for in-place updates was 5n. However, the total number of in-place I/O updates in a normal state appears to be larger than 5n in Figure 6. This was because each chunk was divided into a word size to reduce the I/O size and C_x was generated by more than k pieces of data, although F can usually be generated by k pieces of data.

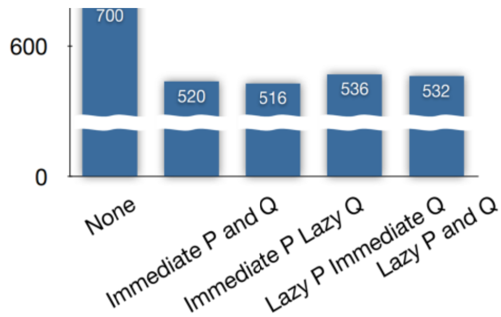


Fig. 6. The number of I/O operations during an in-place update

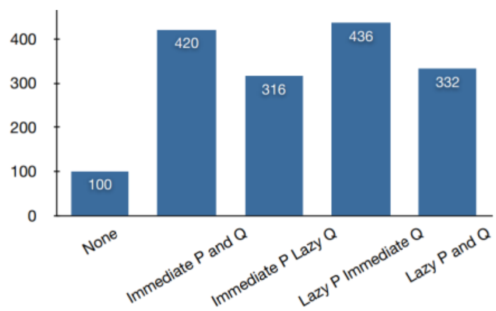


Fig. 7. The number of local I/O operations during an in-place update

Fig. 7 compares the number of local I/O operations, where the method that reflected F immediately had the lowest number of I/O operations. Figure 8 compares the number of remote I/O operations where the method with delayed C_x had a decreasing number of local I/O operations but an increasing number of remote I/O operations.

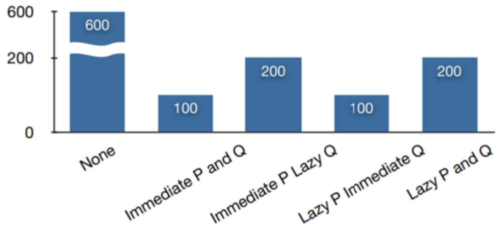


Fig. 8. The number of remote I/O operations during an in-place update

Fig. 9 and 10 shows the IO size of remote operations and location operations. As shown in these figures, lazy update operations reduce the local IO size but as the number of D-Log increases, remote IO size, also, increases.

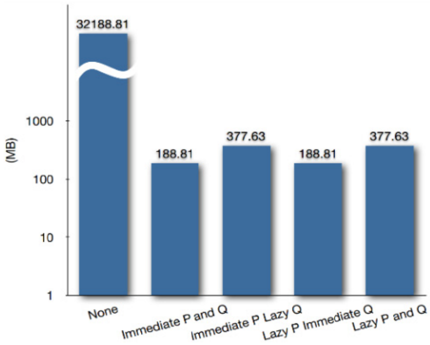


Fig. 9. The remote I/O size during an in-place update

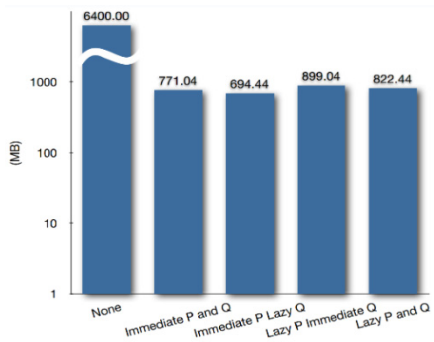


Fig. 10. The local I/O size during an in-place update

Fig. 10 shows the average number of I/O operations as the number of in-place updates increases in a normal state. Using the method that updated F and C_x immediately, there was a low number of I/O operations when there were few in-place updates, while the average number of I/O operations increased with the update frequency. However, the number of I/O operations was close to five with the other methods, which was the minimum number.

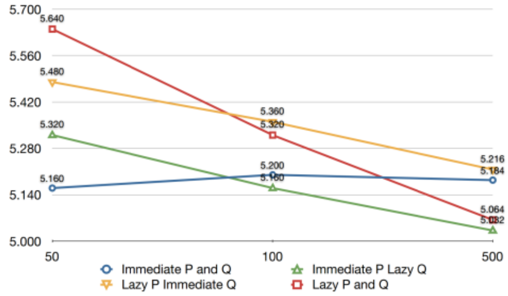


Fig. 11. The mean number of I/O operations as the number of in-place updates increases

5. CONCLUSION

In this paper, we proposed a $D \cdot Log$ based in-place update method for RAID in a DFS. The proposed method improved the in-place update performance of a DFS by reducing the number of I/O and XOR operations. The proposed $D \cdot Log$ based in-place update method updates the parity immediately and it provides a method for storing $D - Log$ on a local disk and updating it later, as well as a method for combining the two methods. These methods can improve the in-place update performance when they are used appropriately, depending on the application. We also proposed a method for updating the parity without decoding when a DS failure occurs or when data is not accessible due to a network partition problem while attempting to update the data. This method did not increase the number of I/O operations and it decreased the number of XOR operations, thereby improving the performance.

We proposed a method for updating parity in a failure state but in practice, the number of I/O operations has a greater effect on the performance in real environments. Therefore, we will develop a method for reducing the number of I/O operations in the future.

ACKNOWLEDGEMENT

This research was jointly supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (NRF-2012R1A1A4A01015615)

REFERENCES

[1] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, “DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing,” Proc. GRID, 2012, pp. 174-183.

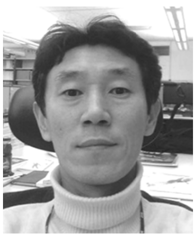
[2] D. Borthakur, “The Hadoop Distributed File System: Architecture and Design,” Hadoop Project Website, 2007.

[3] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, “DiskReduce: RAID for Data-Intensive Scalable Computing,” Proc. PDSW, 2009, pp. 6-10.

[4] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols," ACM SIGCOMM Computer Communication Review, vol. 27, 1997, pp. 24-36.
[5] S. P. James, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems," Proc. Software Practice and Experience, 1997, pp. 995-1012.
[6] S. P. James, "The RAID-6 Liberation codes," Proc. The 6th Usenix Conference on File and Storage Technologies, 2008, pp. 97-110
[7] S. P. James, "The RAID-6 Liber8tion Code," International Journal of High Performance Computing Applications, vol. 23, 2009, pp. 242-251
[8] S. P. James, L. B. Adam, T. Bradley, and V. Zanden, "Minimum Density RAID-6 Codes," ACM Transactions on Storage, vol. 6, 2011.
[9] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," Proc. ACM SIGOPS Operating Systems Review, 2003, pp. 29-43.
[10] Y. Min, H. Kim, and Y. Kim, "Distributed File System for Cloud Computing," Proc. The Korean Institute of Information Scientists and Engineers, 2009, pp. 86-94.
[11] B. Gu, "RAID Technology Introduction," Proc. The Korean Institute of Information Scientists and Engineers, 2009, pp. 61-66.

Ki-jeong Khil
He received the BS and MS degrees in Computer Engineering Department from Korea National University of Korea, Republic of Korea in 2011 and 2013 respectively. He is an researcher of Macroimpact, Republic of Korea. His research interests are database systems, storage systems and so on

Sang-Min Lee
She received the B.S. in computer engineering department from Inha University, Korea in 1991. Since then he has been with the Electronics and Telecommunications Research Institute, Republic of Korea. Her main research interests include system software and distributed system.



Young-Kyun Kim
He received the M.S and Ph.D. in computer science from Chonnam National University, Korea in 1993, 1995 respectively. Since then, he has been with the Electronics and Telecommunications Research Institute, Republic of Korea. His main research interests include file system, distributed system and high performance storage system.

Jaeryong Shin
He received the B.S., M.S, Ph.D. in computer and communication department from Chungbuk National University, Korea in 1996, 1998 and 2002 respectively. From 2003, he has been with, Gwangju Health University, Korea. His main research interests include database, realtime system and multimedia system.

Seokil Song
He received the BS, MS and PhD degrees in Computer and Communication Department from Chungbuk National University of South Korea in 1998, 2000 and 2003, respectively. He is an Associate Professor of the Computer Engineering Department, Korea National University of Transportation, Republic of Korea. His research interests are database systems, index structures, concurrency control, storage systems, sensor network and XML database.