Efficient Data Deduplication System Considering File Modification Pattern

Ho Min Jung*, Sang Yong Park, Jeong Gun Lee, Young Woong Ko

Department of Computer Engineering, Hallym University, Chuncheon, Korea
{chorogyi,sypark,jeonggun.lee,yuko}@hallym.ac.kr

Abstract

In a data deduplication system, the performance of data deduplication algorithms are varying on the condition of file contents. For example, if a file is modified at the end of file region then Fixed-length Chunking algorithm superior to Variable-length Chunking in terms of computation time with similar space reduction result. Therefore, it is important to predict in which location of a file is modified in a deduplication system. In this paper, we discuss a new approach to one of the key methods that is invariably applied to data deduplication. The essential idea is to exploit an efficient file pattern checking scheme that can be used for data deduplication. The contribution of this paper is to find in which region of a file is modified using file similarity information. The file modification pattern can be used for elaborating data deduplication system for selecting deduplication algorithm. Experiment result shows that the proposed system can predict file modification region with high probability.

Keywords: Deduplication, File modification, File pattern, Fixed-length Chunking, Variable-length Chunking

1 Introduction

Data deduplication is a specialized data compression technique for eliminating redundant data by eliminating redundant information and leaves only one copy of the data to be stored, along with references to the unique copy of data. With a help of data deduplication mechanism, the required storage capacity can be reduced and the deduplication scheme is adapted to various storage systems. There are lots of data processing system[1][2], In Content-defined Chunking[3], each block size is partitioned by anchoring based on their data patterns. This scheme can prevent the data shifting problem of the Static Chunking approach. One of the well-known Content-defined Chunking algorithms is LBFS[4], a network file system designed for low bandwidth networks. Delta encoding stores data in the form of differences between sequential data[5]. REBL system use delta encoding approach to implement deduplication service[6]. REBL can efficiently remove the duplicated data with a combination of block suppression, delta encoding and compression.

* This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2009-0076520) and by the Ministry of Education, Science Technology (MEST) and National Research Foundation of Korea(NRF) through the Human Resource Training Project for Regional Innovation

The primary goal of this work is to develop a scheme that, eventually, allows for efficient data deduplication by utilizing data file pattern. For example, in a backup system or versioning file system, most files has similarity with previous version and the file modification is commonly happened at the end of file section. In this case, we can easily eliminate duplicated information using Fixed-length Chunking algorithm by simply dividing data blocks with fixed size chunk. However, if the front part of the file is modified then Fixed-length Chunking will show worst performance result. Therefore, to predict file modification pattern is very important in a deduplication system.

To address these problems, we propose a deduplication system that addresses these issues by adapting dynamic policy changing algorithm considering file modification pattern. The key idea is to exploit file similarity information for predicting file modification pattern. In the proposed system, each file has a file similarity information that composed of representative hash keys and location offset in a file. By utilizing the file similarity information, we can exactly predict in which section of file is modified.

2 System overview

The proposed system exploits file similarity information for efficiently checking duplicated files on the server. In this paper, we expand file similarity-based deduplication system to adapt file modification pattern for more efficient data deduplication. First, the client checks the level of file similarity by sending hash key set to the server. The server check whether the server has similar files or not, by comparing hash keys with files on the server. If the file has similarity value over minimum similarity values then the system starts deduplication processing. In this work, we calculate the SHA1 hash key and send it to the deduplication server. If there is an identical hash key in the hash index, we have a duplicated file on the server. Here, we add the finding file pattern module on the server that calculates in which section of the file is modified compared with the file on the client. In this work, if the file is modified in the front section or end section then we apply the fixed-length chunking algorithm. Otherwise, we apply variable-length chunking algorithm that requires more significant CPU resource for deduplication than fixed-length chunking.

2.1 File similarity search

In this paper, we apply file similarity concept to the deduplication system. File similarity search concept is widely used in data processing system area. The main idia is to extract hash keys from a file. Usually, Rabin hash function calculates hash key from a file and stores the hash key in a queue. By shifting one byte step by step, Rabin hash function repeatedly generates hash key and insert the hash key to the queue. The queue contains only several number of hash keys in ascending order or in descending order by configuration of the system. If Rabin hashing is finished, there remains several hash keys whose value is maximum or minimum. These key value is used for file similarity search. When A file and B file have duplicated hash keys, this means that the files have duplicated region of data. In our system, we extract 1 hash key for every 1 MByte block, therefore if a file has 10 MByte size then the number of hash keys are 10.

2.2 File pattern search algorithm

The main purpose of file pattern search algorithm is to predict the relationship between two files using file similarity information; (1) no duplicated region (2) non-duplicated data is located in the front of a file (3) non-duplicated data is located at the end of a file (4) non-duplicated data is located in the middle section of a file or several sections have non-duplicated data.

Algorithm 1: Find Pattern Search Algorithm

```
Input: Array1, Array2
begin
    isEqualChange ← init
    for $i \leftarrow 0$ to $Array1.Length.$ do
        isEqual ← false
        for $j \leftarrow 0$ to $Array2.Length$ do
            if $Array1[i].hash = Array2[j].hash)$ then
                isEqual ← true
                shift ← Array1[i].offset - Array2[j].offset; break
        if $isEqual = true$ then
            if $isEqualChange = true$ then
                flip ← true
            else if $isEqualChange = false$ then
                flip ← false; flipcnt++
                if $flipcnt == 2$ then
                    cnt++; flipcnt ← 0
            isEqualChange = true
        else
            if $isEqualChange = true$ then
                flip ← false; flipcnt++
                if $flipcnt == 2$ then
                    cnt++; flipcnt ← 0
            else if $isEqualChange = false$ then
                flip ← true
            isEqualChange = false
    if $shift! = 0$ and $cnt == 0$ then
        HeadSection()
    else if $cnt > 0$ then
        EndSection(); HeadSection()
    else
        EndSection()
end
```

In algorithm 1, elements in each array(*Array1, Array2*) contains two tuples: file hash key(*Array.hash*) and file offset(*Array.offset*) value that is a block start location in a file. Algorithm compares hash key elements in each array. *cnt* means the number of block region which contains non-duplicated data. *shift* value contains the difference between two location. First, if two hash keys extracted from *Array1* and *Array2* are equal, *IsEqual* is set to TRUE and stores location difference into *shift* value. if *isEqualChange* is turn to FALSE from TRUE then increase *cnt* value. If there is no change on *shift* and *cnt* is 0 then we assume that two file is identical or have slight difference at the end section. In this case, *headSection()* function is operated for fixed-length chunking. If *cnt* value is over 1 and the modification point is in the middle of the file then we apply variable-length chunking by calling *HeadSection()* and *EndSection()* function. Finally, if shift value is over 0 and *cnt* is 0 then this means that the file is modified in the front of file section. Here, *EndSection()* is processed for fixed-length deduplication.

Algorithm 2: Deduplication algorithm considering file pattern

```
HeadSection()
begin
    offset ← seek(FileStream, Array[ne].offset, seek_set)
    while offset < fdlength do
        byte ← readbyte(FileStream)
        fingerprint ← rabinfingerprint(byte)
        if lookup(fingerprint) = true then
            block ← read(FileStream, offset-blocksize, blocksize)
            hash ← digest(block) if lookup( hash ) = true then
                hashlist ← hashlist ∪ (hash, offset, true)
                break
            else
                hashlist ← hashlist ∪ (hash, offset, false)
    while offset < fdlength do
        block ← read(FileStream, blocklength)
        hash ← digest(block)
        if lookup(hash) = true then
            hashlist ← hashlist ∪ (hash, offset, true)
        else
            hashlist ← hashlist ∪ (hash, offset, false)
end
EndSection()
begin
    while offset < fdlength do
        block ← read(FileStream, blocklength)
        hash ← digest(block)
        if lookup(hash) = true then
            hashlist ← hashlist ∪ (hash, offset, true)
        else
            hashlist ← hashlist ∪ (hash, offset, false)
end
```

In algorithm 2, we explain how *HeadSection()* and *EndSection()* function are work. *Head-*

Section() is a deduplication algorithm for a file that has modified section in front of the file. We use *Array[ne].offset* value as starting point of deduplication where Rabin hash calculates hash key for checking duplicated blocks. Byte shift hash comparison work is perfromed until duplicated hash key is found on certain point where SHA1 hash examines if this block is identical. Here, the proposed system applyes fixed-length chunking to find duplicated blocks. In this paper, we assume that a file has large duplicated area. Therefore, fixed-length chunking can find lots of duplicated blocks. *EndSection()* function is processed in case that a file has modified section at the end of the file. In this case, almost all section of the front region is identical therefore the proposed system begins deduplication process imediately until non-duplicated region is found.

3 Performance evaluation

To perform comprehensive analysis on the proposed algorithm(Predicting file modification pattern), we implemented the client and the server on the platform that consists of 3GHz Pentium 4 Processor, WD-1600JS hard disk and 100Mbps network.

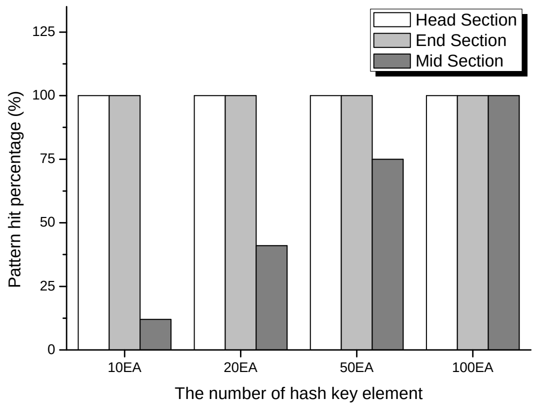

Figure 1: Evaluation result for pattern hit while varying the number of hash key elements

Figure 1 demonstrates the impact of the hash key elements in the file similarity information. The Head Section means that the file is modified in front of the file section. End Section and Mid Section mean that the file is modified at the end section and middle section, respectively. We evaluated file modification pattern hit percentage varying the number of hash key elements from 10 up to 100. We found that file modifications in the Head Section and End Section are perfectly examined with small number of hash key elements. However, modifications in the Mid Section needs many hash key elements for raising hit percentages. The evaluation results explain that the proposed scheme efficiently performs Fixed-length Chunking for Head Section and End Section with less overhead.

Figure 2 shows the impact of the patch size for predicting file pattern. In this experiment, the patch means data block that is used for modifying a file in a random manner. We modified a file using lseek function in Linux system using random value for file offset and applied a patch to make test data file. If the patch size is small then it is difficult to find a location where the file is modified. Thus, the proposed system shows better performance while the patch size is big. For Head Section and End Section, pattern hit percentage is always 100% Regardless of patch size.

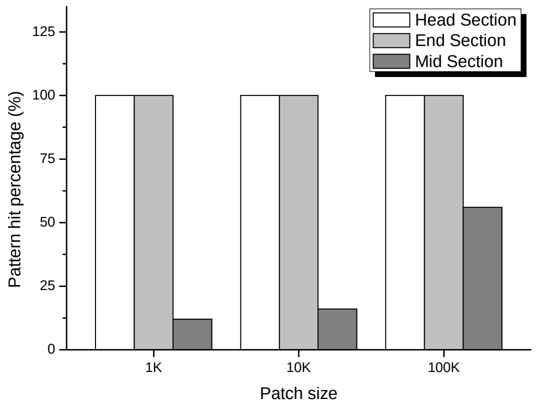

Figure 2: Evaluation result for pattern hit while varying patch size

4 Conclusion

This paper presents an enhanced storage system utilizing file modification pattern to achieve high performance deduplication throughput and capacity. the proposed deduplication system provides dynamic policy changing algorithm considering file modification pattern using file similarity information. We have found that using file modification pattern is superior to traditional data deduplication. In experiment result, with few hash keys from file similarity information, we can predict in which section is modified between files. Futhermore, the proposed system perfectly predicts regardless of patch size for file modification in front of a file or at the end section of a file.

References

[1] JY. Oh and HJ. Kouh, A study on aes extension for large-scale data. The Journal of IWIT. 9, 6 pp. 63–68 (2009)

[2] KY. Lee, MJ. Lim, JJ. Kim, KH. Kim and JL. Kim, Design and implementation of a data management system for mobile spatio-temporal query. The Journal of IWIT. 11, 1 pp. 109–113 (2011)

[3] K. Eshghi and H. Tang, A framework for analyzing and improving content-based chunking algorithms. Hewlett-Packard Labs Technical Report TR. 30 (2005)

[4] A. Muthitacharoen, B. Chen and D. Mazieres, A low-bandwidth network file system. ACM SIGOPS Operating Systems Review. 35, 5 pp. 174–187 (2001)

[5] F. Douglis and A. Iyengar, Application-specific delta-encoding via resemblance detection. In: Proceedings of the USENIX Annual Technical Conference. (2003)

[6] P. Kulkarni, F. Douglis, J. LaVoie and J. Tracey, Redundancy elimination within large collections of files. In: Proceedings of the annual conference on USENIX Annual Technical Conference, USENIX Association (2004)