International Journal of Software Engineering and Its Applications
Vol.8, No.1 (2014), pp.33-42
http://dx.doi.org/10.14257/ijseia.2014.8.1.03

Design and Implementation of Storage System Using Byte-index Chunking Scheme

Ider Lkhagvasuren, Jung Min So, Jeong Gun Lee, Jin Kim and Young Woong Ko[1]

[1]*Dept. of Computer Engineering, Hallym University, Chuncheon, Korea*
{Ider555, jso, jeonggun.lee, jinkim, yuko}@hallym.ac.kr

Abstract

In this paper, we present an enhanced storage system that supports Byte-index chunking algorithm. The storage system aims to provide efficient data deduplication with high performance and to be performed in rapid time. We describe the overall procedure of Byte-index chunking based storage system including read/write procedure and how the system works. The key idea of Byte-index chunking is to adapt fixed-size block chunk scheme which are distributed to "Index-table" by chunk's both side boundary values. We have found that Byte-index chunking in storage system provides high performance compared with other chunking schemes. Experiments result shows that the storage system with Byte-index chunking compresses overall data with high deduplication capability and reduce the speed of file processing.

Keywords: *Storage System, Deduplication, Byte-index, Chunking*

1. Introduction

The amount of digital data is growing very explosive, so efficient data management scheme is more and more important than before. To overcome this challenge, there are many approaches to enhance the performance of storage system. The data deduplication technique is regarded as an enabling technology. The data deduplication is a specialized data compression scheme that eliminates duplicated information to improve storage utilization and reduce network bandwidth. In the deduplication process, duplicate data is eliminated by using hash comparison scheme, leaving only one copy of the data to be stored, along with references to the unique copy of data. For example, input data is divided into fixed or variable size chunk and transformed into a hash value for each chunk. If the hash value is same with a hash value in a server, we regard that the chunk is identical one. Therefore, we can eliminate duplicate data blocks to be stored in a server. Furthermore, it saves network bandwidth between a client and a server. So, the data deduplication technique can be used efficiently handling data files in a storage system.

Generally, the functionalities of deduplication based storage system are composed of hashing, comparing, input/output data blocks, searching and network transferring. The architecture of data deduplication system varies based on composing mechanism of the functionalities. To effectively reduce hash calculation time, we propose a novel storage server using Byte index chunking approach. The primary key benefit of this work is minimizing performance time and achieving high rate redundancy in storage system. The proposed scheme can diminish disk I/O and hashing time, additionally it is also designed for elimination of redundant data blocks using Byte-index deduplication. The key idea of the

[1] Corresponding author : yuko@hallym.ac.kr

ISSN: 1738-9984 IJSEIA

Byte-index chunking based approach is using Index-table to find very high-probable duplicated data. Index-table is 256*256 sized table structure, contains original file chunk-indexes and used as reference of original file in the proposed system. Chunk-index of the chunk is stored into Index-table where left side boundary byte value of the chunk is express row number and right side boundary value of the chunk express column number in the Index-table. The proposed system only takes two boundary bytes at the each offset and using these boundary bytes, accesses the cell of Index-table and checks if cell is storing value while system is scanning through modified file. If there is value in the corresponding cell of Index-table, scanning offset position is shifted by length of chunk-size. Other case, scanning offset position is shifted by only one byte. Therefore proposed system could save time in detecting duplicated data region in a file.

This paper is organized as follows. In Section 2, we describe related works about deduplication policy and systems. In Section 3, we explain the design principle and implementation details. In Section 4, we show performance evaluation result of the proposed system and we conclude and discuss future research plan.

2. Related Works

There are several different data deduplication algorithms including static chunking [1], content defined chunking [2], whole-file chunking, delta encoding and file pattern-based approach [4]. Static chunking [1] is the fastest algorithm among the others for detecting duplicated blocks but the performance is not acceptable. On the other hand, byte shifting can detect all of the duplicated blocks with high overhead. The main limitation of the static chunking is "boundary shift problem". For example when adding new data to a file, all subsequent blocks in the file will be rewritten and are likely to be considered as different from those in original file. Therefore, it's difficult to find duplicated blocks in the file, which makes deduplication performance degrade. One of the well-known static chunking schemes is Venti [1]. Venti is network storage system using static chunking, where 160-bit SHA-1 hash key is used as the address of the data. This enforces write-once policy since no other data block can be found with the same address. The addresses of multiple writes of the same data are identical, so duplicate data is easily identified and the data block is stored only once. Content-defined chunking [5] data deduplication provides a user with opportunity of best density of storage. However, content-defined data deduplication approach can achieve high deduplication ratio, but spends too much time to perform than other data deduplication approaches. In content defined chunking 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. LBFS [2] is a network file system designed for low bandwidth networks. LBFS exploits similarities between files or versions of the same file to save bandwidth. It avoids sending data over the network when the same data can already be found in the server's file system or the client's cache. Using this technique, LBFS achieves up to two orders of magnitude reduction in bandwidth utilization on common workloads, compared to traditional network file systems. Delta encoding stores data in the form of differences between sequential data. Lots of backup system adopts this scheme in order to give their users previous versions of the same file from previous backups. This reduces associated costs in the amount of data that has to be stored as differing versions. DEDE is a decentralized deduplication system designed for SAN clustered file system that supports a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host queries and updates a shared index for the hashes in its own write-log to identify and reclaim storage for duplicate blocks. Unlike inline deduplication

systems, the deduplication process is done out-of-band so as to minimize its impact on file system performance. In [4], they propose a data deduplication system using file modification pattern. This approach can detect how file is modified and what types of deduplication is best for the data deduplication.

3. System Architecture of Byte-index based Storage System

We have implemented a storage system that adapts Byte-index chunking. The proposed system is designed to support deduplication performance comparable to content-defined chunking, also provide rapid processing time that be able to match fixed-length chunking approach. The key idea of the system is to adapt Byte-index chunking. In this chapter, we describe byte-index chunking algorithm very briefly.

3.1 Byte-index Chunking Algorithm Concept

The purpose of Byte-index chunking is to find the chunks that are expected to be duplicated (highly probable duplicate chunk) by their byte values. For improving search results to be more accurate, the proposed system not only searches a single chunk, but also aims to seek adjacent double chunks of the target file. If the chunk in target file not only has the same length with the any adjacent chunks in the server, but also both these chunks store same bytes of values at the position where boundaries of each chunk, then we call this chunk as duplicate chunk with high probability.

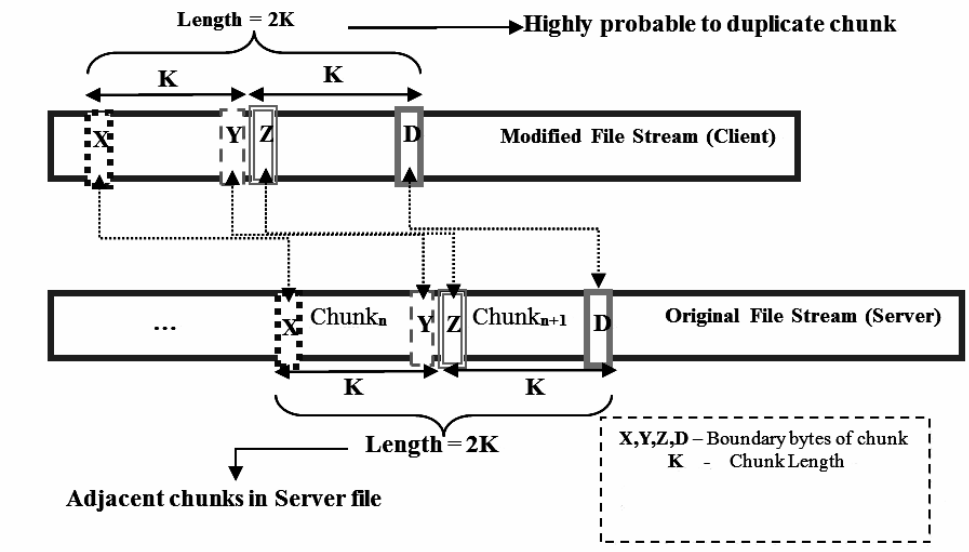


Figure 1. Duplicated Chunk Look up Process Overview

As we can see in Figure 1, original file stream is divided into chunks and its boundary bytes are used as index values. For example, the hash value of $Chunk_n$ is stored into (X,Y) position of index table. Also of $Chunk_{n+1}$ is stored into (Z,D) position of index table. When there is a modified file stream, the proposed system reads a hash value in (x,y) position of index table which are boundary values with K length apart. If the hash value is same with the value in index table then we assume the chunk is same one with original file. In this figure, (X,Y) value of modified file stream will be match with (X,Y) value of original file. Furthermore if we find one Byte-index (X,Y) in a modified file stream then we will find

adjacent Byte-index (Z,D) because duplicated blocks tend to have locality. In our approach, probability of duplication is the only one from 2564 (4,294,967,296) occasions and those two chunks have the same bytes in their specific four positions. Moreover, we can get each data block hash by applying hash functions such as SHA1, MD5 and SHA256.

3.2. Byte-index Chunking Procedure

Suppose we have two general purpose computers α and β. Computer α has access to a file A and β has access to file B, where A and B are "similar"[6].

The Byte-Index chunking algorithm consists of the following steps:

1. β Splits the file B into a series of fixed-sized blocks of size S bytes. The last block may be shorter than S bytes.
2. For each of these blocks β calculates SHA1 hash and numbering
3. After calculating hashes and numbering process, β takes left and right side edge bytes (boundary bytes) of each chunks, then sets chunk numbers to the Index-table by using their boundary values. Left boundary byte value of chunk implies vertical column number of Index-table and right boundary byte value of chunk implies horizontal column number of Index-table.
4. β sends filled Index-table to α.
5. α starts to scan through the A, read only 4 specific bytes at the each reading offset. Those specific 4 bytes imply boundary bytes of the adjacent chunks. I.E. suppose α is scanning A at i-th offset. Therefore locations of our specific 4 bytes are i-th, (i+S-1)-th, (i+S)-th and (i+2S-1)-th offset. After read 4 specific bytes, α access cells of Index-table where row number equals to i-th byte value, column number equals to (i+S-1)-th byte value and row number equals to (i+S)-th byte value, column number equals to (i+2S-1)-th byte value to check if these two cells contain adjacent chunk indexes. If check result is negative, scanning process continues to shifting single byte and checks next byte to read 4 specific bytes. If there are adjacent chunk-indexes stored in the these cells of Index-table, α calculates hashes of chunks which are i-th to (i+S-1)-th offset (i+S)-th to (i+2S-1)-th offset of the A, consider those chunks to be high-probable to duplicated chunk, put it to the result hash-list and continue to scanning process with shifting by 2S. Hereby α fills the result hash-list.
6. α sends result hash-list to β.
7. β collects hashes from B and α, comparing them to be matched one by one. Matched chunks are mean to be duplicated.
8. β sends to α chunk number-list of matched and unmatched.
9. For unmatched chunks from β, α repeat step 5-8, searching process is performed in only region of unmatched chunk's region. Repeat process is taken action until there's no region for scanning.
10. α sends non duplicates from A to the β

3.3. Byte-index Chunking Procedure

Our goal is to transfer only non-overlapping chunks of files and circumvents sensitivity. The proposed system server has to perform a few processes in idle mode that will already have done before the arrival of the file deduplication request from client. In other words, the processes of indexing the server file, calculates hashes and move them to the server database are have to be done in the server at first in the system idle mode.

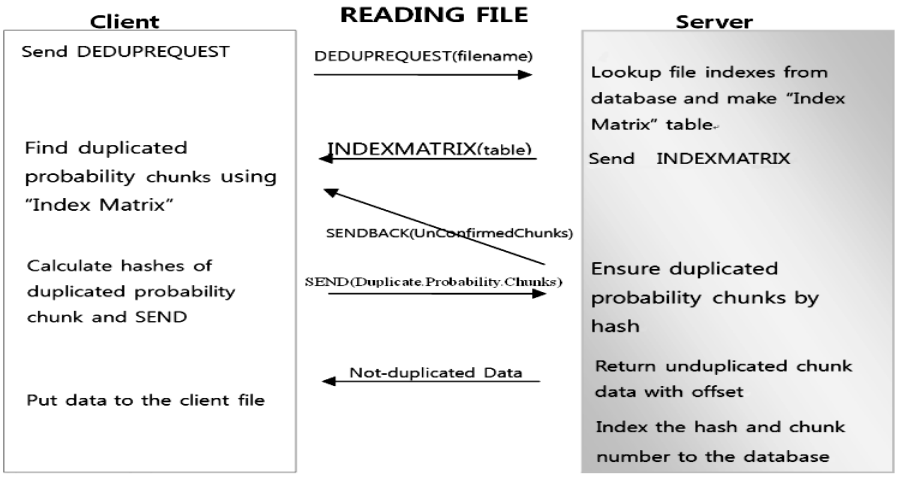


Figure 2. Reading a File using Byte Index based Deduplication System

As shown in the Figure 2, when the server receives a deduplication request from the client, the server examines every chunk index of through the requesting file, take out the chunk number (chunk index) of them and put them into "Index-table". Particularly, this means distribute the all the chunk numbers of requested file from database to the 256*256 table (Index-table) mentioned in the previous sections. After that, the server sends "Index-table" to the client. The client receives "Index-table" and examines the client file (modified file) bytes to look for a high probable of duplicate chunks using received "Index-table". It estimates hash values in the lookup results of duplicated probability chunks.

The client sends these hash values to the server to confirm precisely whether highly probable chunks are duplicated by their hash values or not. If any of chunks is not proven to be duplicated, then the non-duplicated chunk will be sent back to the client for re-examination to find another duplicated probable chunk in the range of its non-duplicated chunk only.

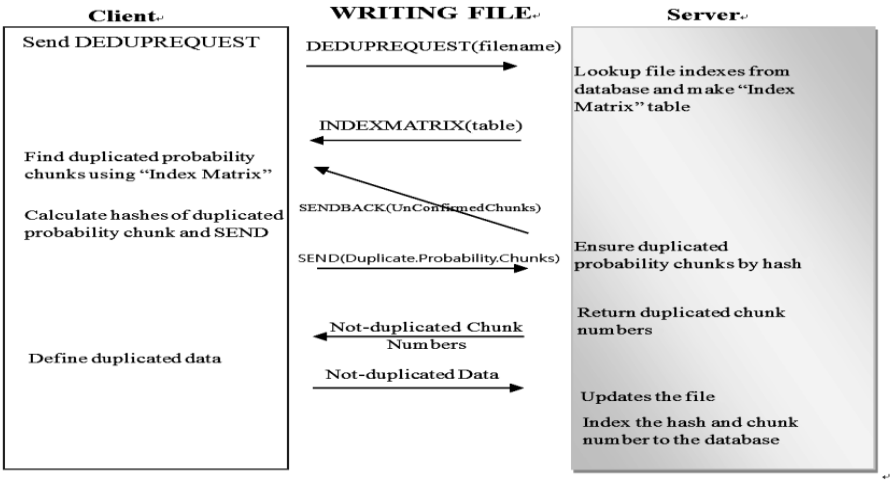


Figure 3. Writing a File using Byte Index based Deduplication System

After all these processes, we can define duplicated chunks in the server. Finally, depending on what we do - processing on reading file or writing a file - to the server, next processes become different. To read file from the server (Figure 2), the server sends directly non-

duplicated data to the client. To write file to the server (Figure 3), the server sends to the client duplicated chunk numbers from the client. Then the client receives the duplicated chunk numbers, and defines non-duplicated chunks and finally sends non-duplicated chunks to the server. The server updates again the file in the database as indexing and hashing in the offline mode.

4. Performance Evaluation

This section evaluates the performance of a storage system with Byte-index chunking with several experiments. First, we examine the behavior of proposed system's deduplication ratio result with comparing content-defined chunking, and fixed-size chunking approach. Next, we measure the performance time consumption utilization of system under several common workloads and compare it to the content-defined and fixed-size chunking approaches with deduplication result and with measured the performance time. In this work, we developed a deduplication storage system and evaluate the performance of the proposed algorithm. The server and the client platform consist of 3 GHz Pentium 4 Processor, WD-1600JS hard disk, 100 Mbps network. The software is implemented on Linux kernel version 2.6.18 Fedora Core 9. To perform comprehensive analysis on deduplication algorithm, we implemented several deduplication algorithms for comparison purpose including fixed-length chunking, and content-defined chunking.

We have 1110MB sized two (.rar) files including 80% duplicates between each other. We made file deduplication experiments using these files as inputs of the experiment when block sizes are 16KB, 32KB, 64KB, 128KB, 256KB, 512KB and 1MB respectively.

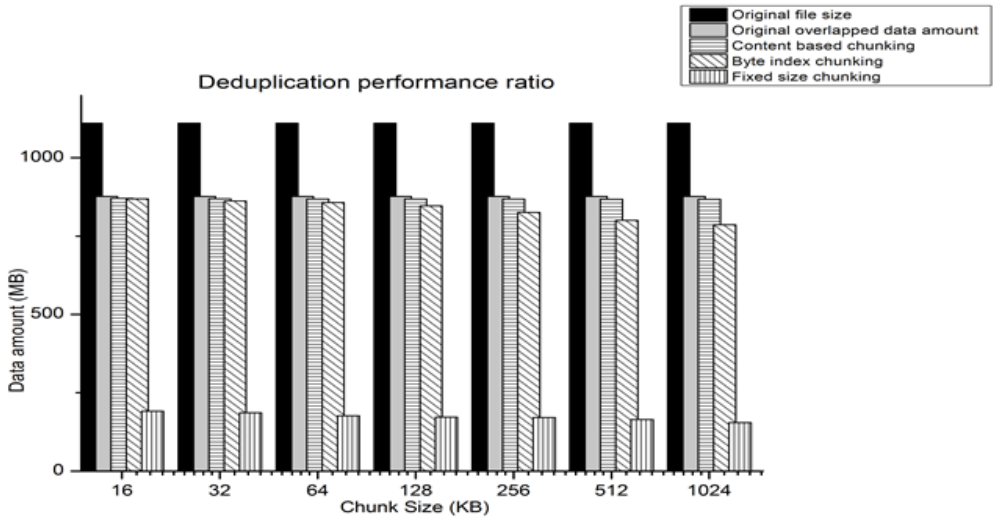


Figure 4. Deduplication Ratio result of Chunking based Approaches (By Chunk Size)

Figure 4 shows each duplicate redundancy capability of content-defined chunking, fixed-size chunking approach and Byte-index chunking approach. From this experiment graph, content-defined chunking approach shows the best performance among several algorithms for detecting actual duplicates. There is almost no difference between overlapping amount of original file and the file using content-defined chunking. Fixed-size chunking approach shows the lowest deduplication ratio result because of vulnerable to byte shifts problem inside the

data stream. The proposed system shows high-rate redundancy result as content-defined chunking does.

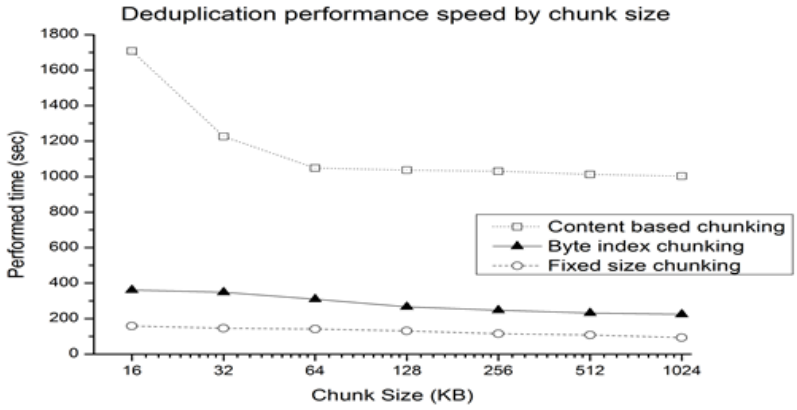


Figure 5. Performance Result of Byte-index Chunking

Figure 5 presents the deduplication speed performance for several chunking algorithms. All of these approaches tend to be slow as chunk size is getting increased. From the graph, we can find the content-defined chunking approach consumes longest time to perform and fixed-sized chunking approach is performed in the fastest time. The proposed Byte-index chunking approach shows much faster performance than content based chunking and shows slightly slower than fixed-length chunking approach. Therefore, we can conclude that the proposed Byte-index chunking approach is optimal approach with fast and high-rate deduplication performance among other chunking approaches.

5. Conclusion

This paper presents an enhanced storage system that supports byte-index chunking algorithm which can be efficiently used with high performance deduplication throughput and capacity in rapid time. In this paper, we described the overall procedure of byte-index chunking based storage system including read/write procedure and how the system works. We have found that using Byte-index chunking in storage system provides high performance compared with other chunking schemes. Experiments result shows that the storage system with Byte-index chunking compresses overall data with high deduplication capability and reduce the speed of file processing. We believe that the proposed system can be applicable to various storage system such as backup, P2P and cloud system. However, several issues remain open. First, our work has limitations on supporting simple data file which has redundant data blocks with spatial locality; therefore, if the file has several modifications then overall performance will be degrade. For future work, we plan to build a massive storage system with huge number of files. In this case, the file deduplication performance evaluation result will be very realistic.

Acknowledgements

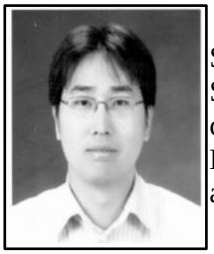
This research was supported by Basic Science Research Program through the NRF funded by the MEST (2012R1A1A2044694).

References

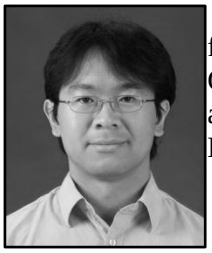
[1] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage", Proceedings of the FAST 2002 Conference on File and Storage Technologies, **(2002)**.
[2] A. Muthitacharoen, B. Chen and D. Mazieres, "A low-bandwidth network file system", ACM SIGOPS Operating Systems Review, vol. 35, no. 5, **(2001)**, pp. 174-187.
[3] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection", Proceedings of the USENIX Annual Technical Conference, **(2003)**, pp. 1-23.
[4] H. M. Jung, S. Y. Park, J. G. Lee and Y. W. Ko, "Efficient Data Deduplication System Considering File Modification Pattern", International Journal of Security and Its Applications, vol. 6, no. 2, **(2012)**.
[5] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms", Hewlett-Packard Labs Technical Report TR., vol. 30, **(2005)**.
[6] A. Tridgell and P. Mackerras, "The Rsync algorithm", Tech. Rep. TR-CS-96-05, The Australian National University, **(1996)** June.

Authors

Ider Lkhagvasuren graduated from Mongolian University of Science and Technology in 2007. He also graduated Department of Computer Engineering, Hallym University with master's degree in 2013.He is currently an working as researcher in Operating System Laboratory, Department of Computer Engineering, Hallym University. His research interests include Data deduplication and Cloud system.



Jungmin So received the B.S. degree in computer engineering from Seoul National University in 2001, and Ph.D. degree in Computer Science from University of Illinois at Urbana-Champaign in 2006. He is currently an assistant professor in Department of Computer Engineering, Hallym University. His research interests include wireless networking and mobile computing.



Jeong-Gun Lee received the B.S. degree in computer engineering from Hallym University in 1996, and M.S. and Ph.D. degree from Gwangju Institute of Science and Technology (GIST), Korea, in 1998 and 2005. He is currently an assistant professor in the Computer Engineering department at Hallym University.

Jin Kim received an MS degree in computer science from the college of Engineering at the Michigan State University in 1990, and in 1996 a PhD degree from the Michigan State University. Since then he has been working as a professor on computer engineering at the Hallym University. His research includes Bioinformatics and data mining.

Young Woong Ko received both a M.S. and Ph.D. in computer science from Korea University, Seoul, Korea, in 1999 and 2003, respectively. He is now a professor in Department of Computer engineering, Hallym University, Korea. His research interests include operating system, embedded system and multimedia system.

