Data Deduplication using B+ Tree Indexing

Tin Thein Thwel, Ni Lar Thein
University of Computer Studies, Yangon
tin.thein.thwel@gmail.com, nilarthein@gmail.com

Abstract

As the amount of storage utilization become larger and larger, people have been tried to find out the efficient ways to safe storage space. The single instance storage or data deduplication becomes vague in storage management as it can eliminate duplicated data or segments in those files. In this paper, we proposed Data Deduplication System for sub-file level. This system can perform deduplication with the integrated use of file chunking algorithm; secure hash function and B+ tree indexing. In this system, we will first separate the file into variable_length segments or chunks using Two Thresholds Two Divisors chunking algorithm. ChunkIDs are then obtained by applying hash function to the chunks. The resulted ChunkIDs are used to build as indexing keys in B+ tree like index structure. This system can reduce the indexing time complexity from O (n) to O (log n). The performance of proposed system will be compared with the other systems in terms of performance metrics such as WinZIP, WinRAR, etc.

1. Introduction

At present, there is a vast amount of duplicated data or redundant data in storage systems. Duplicated data exist among variants of the same file as well as it can also occur among different files. Those vast amounts of data duplications result in extra storage spaces to be used and much more power consumptions, greatly lowering the storage utilization. Data de-duplication can eliminate multiple copies of the same file and duplicated segments or chunks of data within those files. Data de-duplication is also called intelligent compression as it can aware the contents of the file. Unlike the simple compression method in which:

(1) lacks awareness of underlying data;
(2) can't recognize two identical files exist in different directories;
(3) can't recognize changed data or capture unique blocks at a sub-file level

In these days, therefore, data de-duplication becomes an interesting field in storage environments especially in persistent data storage for data centers. Many data deduplication mechanisms have been proposed for efficient data deduplication in order to safe storage space. Current issue for data deduplication is to avoid full-chunk indexing to identify the incoming data is new, which is time consuming process. In this paper, we propose an efficient indexing mechanism for this problem using the advantage of B+ tree properties.

The rest of the paper is organized as follow: Section 2 introduces related approaches used to deduplicate the inter-file level redundancies; the architecture of Efficient Indexing Mechanism for Data Deduplication is presented in Section 3; Section 4 depicts the proposed indexing mechanism; Section 5 shows the preliminary experimental setup and results; and Section 6 draws the conclusion.

2. Related Work

To the extent of our best knowledge, B+ tree indexing are only used in relational database system in order to get efficient indexing and not on the inter-file level file storage and indexing with integrated use of hash code (Chunk_ID) as index in data deduplication.

M.Lillibridge et al. described problem statement as chunk-lookup disk bottleneck/full chunks indexing encountered in in-line deduplication and they try to solve using sampling and sparse index and chunk locality [11]. However, they based on assumption that if two segment share one chunk, it is likely to be shared other chunks only limited number of segments are deduplicated and can't do fully deduplication as sometimes can store duplicate chunks.

Sridhar Ramaswamy pointed out the indexing problems in constraint and temporal data [13]. They proposed to uses B+ tree but their work had done on database, solution is optimal for query, but doesn't have good worst-case bounds for updating.

Walter Santos solved the problem of identification of replicas in database with parallel deduplication algorithm using filter-stream model [14] and only considered for databases and not for sub-file level of the file which have their respective secure hash code.

Daniel P.Lopresti described the issues related to the detection of duplicates in document image databases and proposed four algorithm using edit distance [5]. But they can solve only for image database specific.

Jared, D. et al. find out that accurate rule-based deduplication requires significant manual tuning of both rules and thresholds. They proposed learning – based information fusion using SVM but done on database record, specific to set of deduplication rules [9]. If other set of rules use, match accuracy values will decrease.

Z. Benjamin et al. proposed Summary Vector (Bloom Filtering) and Locality Preserved Caching to avoiding the disk bottleneck in the Data Domain Deduplication File System [18]. Their assumption is that if the last time encountered chunk A, it was surrounded by chunks B, C, D, then next time encounter A, it is likely to encounter B, C or D nearby. So, their system can avoid full chunk indexing. Because of this assumption, this system is not fully deduplication and sometimes can store duplicated data.

3. Overview of the proposed system

As the mainly designed for this system is to avoid full chunks indexing, the system tried to use the B+ tree indexing structure for efficient indexing. For checking the identical chunks in the B+ tree structure, one-way hash function is used which can reduce the risk of finding identical information in the file.

The system will first separate the file into variable-length chunks using Two Threshold Two Divisor chunking algorithm (TTTD algorithm). ChunkIDs are then obtained by applying hash function to those variable-length chunks. The resulted chunkIDs are used to build as indexing keys in B+ tree like index structure.

3.1. System Architecture Overview

The architecture of the proposed system is illustrated in Figure. The system includes the major components: File Chunker, Chunk_ID Generator, Duplicate Finder, Metadata and Storage. Input file format may include .PDF, .html, .txt, .doc, etc. Input files are separated into variable length chunks by File Chunker. The Chunk_ID Generator uses the resulted chunks from the File Chunker to generate Chunk_ID by applying SHA1, which is secure hash function, also well known for its hash collision resistance. By using the Chunk_ID, Duplicate Finder checks whether that Chunk_ID is already exist or not in the Metadata, where the proposed B+ tree indexing mechanism is built. The contents of the chunk data are stored in the storage space.

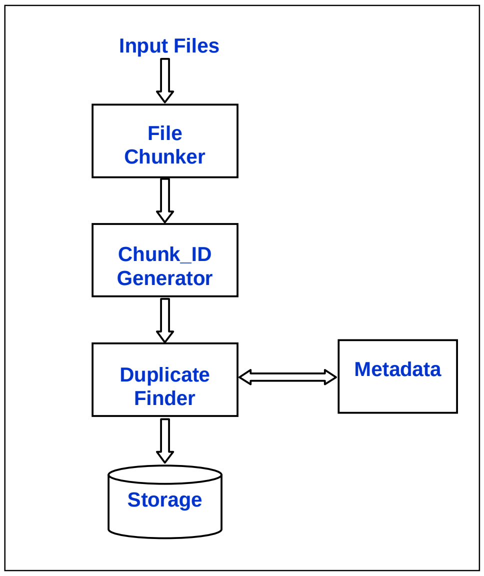

Figure 1. Proposed system architecture

3.1.1. Chunking (File Chunker). Chunking divides a file deterministically into segments or chunks. This technique was initially used in the Low-bandwidth Network File System [7] to conserve bandwidth. Although smaller segments yield better deduplication, they results in a large number of chunks and led to increase in metadata associated with them. On the other hand, large size chunks can reduce the chance of identifying duplicate data. Two Thresholds Two Divisors (TTTD) chunking algorithm can avoid very small chunking and very large chunking. Hence, we use TTTD chunking algorithm in the proposed system.

3.1.2. Hashing (SHA1). To generate the Chunk_ID, the Chunk_ID generator uses SHA1 which produces 160 bits signature for each chunk.

3.1.3. Deduplication (Duplicate Finder). Its finds the duplicate Chunk_ID in the existing B+ tree indexing structure with the incoming Chunk_ID.

The algorithm for the duplicate finder is as follow:

```
DuplicateFinder(chunkID,chunkIDMetadata,Chunk)
Begin
    found ← searchInBPlusTree(chunkID)
    if  found in BPlusTree
    then
            appendBPlusTree(chunkIDMetadata)
    else
     updateBPlusTree(chunkID,chunkIDMetadata)
     store(chunk)
    endif
End
```

3.1.4. Indexing. The metadata maintain the B+ tree structure of the already stored files information including Chunk_ID, file information. The proposed indexing mechanism using B+ tree properties is depicted in Section 4.

4. Proposed B+ Tree indexing mechanism

The resulted Chunk_ID generated from the Chunk_ID Generator are used to construct as B+ tree index structure and maintains as metadata. By using the advantage of B+ tree properties, the optimal search time O(log n) which is more efficient than the full chunk indexing O(n). The proposed indexing mechanism is as shown in Figure 2.

Where,

$Chunk\_ID_n$	– Hash Code
Chunk_ID's Metadata	–file name, file type, chunk number, offset of the chunk in that file, chunk length
$Chunk_n$	–Content of chunk

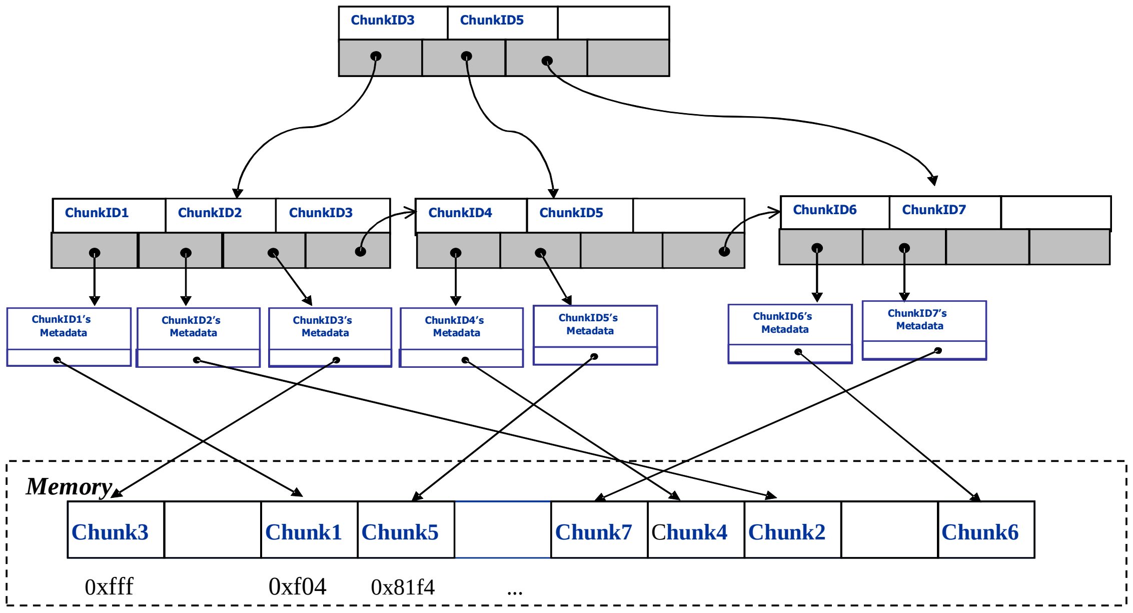

Figure 2. The proposed B+ Tree indexing mechanism

4.1. Time Complexity of B+ Tree

The comparison of B+ tree indexing to full chunk indexing mechanism is shown in Table 1.

Table 1.Comparison of B+ tree indexing to other indexing mechanism

	Creation	Insertion	Searching
Full chunk Indexing	O(1)	O(n)	O(n)
B+ tree Indexing	O(1)	O(log n)	O(log n)

5. Experimental Setup

In our implementation, The ChunkIDs are generated using SHA1 hash algorithm. The data are internally stored as files on the underlying ext3 filesystem. We take an experimental approach to compare it with the compression performance of two other typical traditional methods. The testbed is deployed to consist of one server. The server had dual-core processors at 2.1 GHz, 2 GB RAM, and Hard disk with 320 GB. While by now it is difficult to get the real enterprise level dataset, we collect some workloads which are general enough to represent the characteristics of the real massive data and to prove the concept. The workloads include: 1658.88 MB (1.62 GB) of web documents, text files and PDF files.

A data deduplication ratio over a particular time period is the number of bytes input to a data deduplication process divided by the number of bytes output. Figure 3 depicts the space reduction ratio relevant in most situations which reflects all of the complementary capacity optimization technologies actually used.

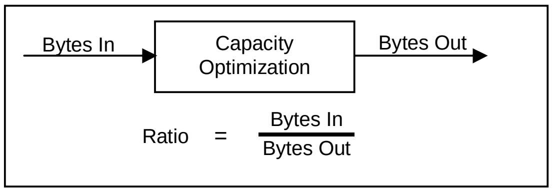

Figure 3. Space Reduction Ratio

The compression results of the proposed system compared with other methods (we use WinZIP and WinRAR here) are described in Table 2.

Table 2.Comparison of compression ratio using different compress methods

	Original Space	Win ZIP	Win RAR	Our System
	1.62 GB	1.29 GB	1.10 GB	1.27 GB
Space Saving	-	338.76 MB (20 %)	532.48 MB (31.5%)	348.16 MB (20.5 %)
Compression ratio(%)	-	1:0.79	1:0.68	1:0.78

6. Conclusion

We designed a framework for Efficient Data Deduplication. It can reduce the storage space as its potential purpose. Because of using effective and efficient Hash Algorithm for creating Chunk_ID to check duplicate data, it can be more effective and reliable for security and hash collision. We proposed an efficient indexing mechanism to speed up the searching facility to identify redundant chunks. By using proposed Chunk_ID based B+ tree searching mechanism, significantly reduce the searching time and comparison space.

7. References

[1] Alberto H.F. Laender, Altigran Soares da Silva," Learning to deduplicate", Proc. 8th ACM/IEEE-CS joint conference on Digital libraries (JCDL), Association for Computing Machinery (ACM) , New York, USA, 2006, pp. 41-50.

[2] Athicha, M., Benjie, C., and David, M., "LBFS: A low-bandwidth network files system". 18th ACM Symposium on Operating Systems Principles (SOSP '01), Canada, 2001, pp. 174–187.

[3] Bayer .R and MC. Creight, "Organization and Maintenance of Large Ordered Indices", Acta Informatica, Volume 1, Springer Berlin/Heidelberg, New York, 1972, pp. 173-189.

[4] Chauanyi, L. et.al, "ADMAD: Application-Driven Metadata Aware De-duplication Archival Storage System", Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI), IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp.29-35.

[5] Daniel P. Lopresti, "Models and Algorithms for Duplicate Document Detection", Preceedings of the Fifth International Conference on Document Analysis and Recognition (ICDAR'99), IEEE Computer Society, Washington, DC, USA, 1999, pp. 297-300.

[6] Dave Reinsel, "Our Expanding Digital World: Can we contain it? Can we manage it?", Intelligent Storage Workshop (ISW), University of Minnesota, MN, May 2008, pp. 13-14.

[7] Eshghi, K., "A Framework for Analyzing and Improving Content-based Chunking Algorithms", Technical Report HPL-2005-30(R.1), Hewlett-Packard Laboratories, Palo Alto, CA, 2005.

[8] Eshghi, E. et.al., "High Performance Scalable Data Deduplication", Storage Systems Research Center: University of California, 2008.

[9] Jared, D. et.al, "Learning-based Fusion for Data Deduplication", Seventh International Conference on Machine Learning and Applications (ICMLA'08), IEEE Computer Society, California, USA, 2008, pp. 66-71.

[10] M. O. Rabin, "Fingerprinting by random polynomials", Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[11] Michael T. Goodrich, Data Structures and Algorithm in C++, Wiley Publishing, 2009, pp. 598.

[12] M.Lillibridge et al., "Sparse Indexing, Large Scale, Inline Deduplication Using Sampling and Locality". 7th USENIX Conference on File and Storage Technologies, USENIX Association, San Francisco, California, 2009, pp. 111-123.

[13] National Security Agency, "Secure Hash Standard", Federal Information Processing Standards Publication 180-1, US government standards agency NIST, 1995.

[14] S., Ramaswamy, "Efficient Indexing for Constraint and Temporal Databases", Preceedings of 6th International Conference on Database Theory, Springer Berlin/Heidelberg Bell, Delphi, Greece, 1997, pp. 419-431.

[15] S. Walter, T.Thiago, M.Carla and Jr. Wagner Meira, "A Scalable Parallel Deduplication Algorithm", 19th International Symposium on Computer Architecture and High Performance Computing, IEEE Computer Society, Brazil, 2007, pp. 79-86.

[16] W.You et al., "PRUN: Eliminating Information Redundancy for Large Scale Data Backup System", International Conference on Computational Sciences and Its Applications (ICCSA 2008), IEEE Computer Society, Italy, 2008, pp. 139-144.

[17] Z. Benjamin et al., "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System", 6th USENIX Conference on File and Storage Technologies (FAST '08), USENIX Association, San Jose, USA, 2008, pp. 269-282.