Alternatives for Eliminating Duplicate in Data Storage*

Tianming Yang, Jing Zhang, and Wei Sun
International Collage, Huanghuai University, Henan Province, China
{ytmzqyy & phzj-123}@163.com, swssl@yahoo.com.cn

Abstract –Duplicate Elimination (DE) is a specialized data compression technique for eliminating duplicate copies of repeating data to optimize the use of storage space or bandwidth. The most common form of DE implementation works by dividing files as chunks and comparing chunks of data to detect duplicates. This paper implements a content-based chunking algorithm to improve duplicate elimination over fixed-sized blocking, and evaluates the methods of chunk comparison, that is, compare-by-hash versus compare-by-value. It indicates that compare-by-hash is efficient and feasible even employed in ultra-large-scale storage systems.

Index Terms – Data Storage, Duplicate Elimination, Compare-by-Hash.

1. Introduction

In recent years, we have seen rapid growth in the amount of digital information, which poses big challenges for modern enterprises to protect their valuable data [1]. In order to reduce the amount of storage needed for backups, storage-based DE is now getting popularity [2, 3, 4, 5]. It has been proved very effective in applications where many copies of very similar or even identical data are stored on a single disk or in the case of data backups where most data in a given backup is unchanged from the previous one [6]. Traditional backup systems try to reduce storage requirement by hard linking files that have not changed or storing differences between files. But hard linking does not help with large files that are changed in small ways, such as an email database; differences only find duplicates in adjacent versions of a single file. In contrast, storage-based DE eliminates duplicate in the global system, hence can achieve far more efficient data compression than traditional backup methods. DDFS [2], for example, reported a 38.54:1 cumulative compression rate when backing up a real world data center over a time span of one month.

The most common form of DE implementation works by dividing data stream as chunks and comparing chunks of data to detect duplicates. Dividing data stream as fixed-sized blocks is straightforward and consonant with mainstream file systems in which files are managed as fixed-sized logical blocks, and there indeed exist DE systems such as Venti [7] and DDE [8] that eliminate duplicated copies of fixed size data blocks. However, fixed-sized blocking will result in shifting-offsets-problem that a single byte inserted at the start of a large file would shift all the block boundaries, and thereby thwart any potential storage savings [9]. In order to solve the shifting-offset-problem, other chunking algorithms, such as the semantic-based chunking algorithm ADMAD (Application-Driven Metadata Aware Deduplication) [10], and the content-based chunking algorithms CDC (Content-Defined Chunking) [9], TTTD (two-threshold, two-divisor) [11], Fingerdiff [12] and BiCDC (Bimodal Content Defined Chunking) [13] etc. have been proposed. The most commonly used CDC algorithm uses hashing technique to determine chunk boundaries to divide the data stream into variable-sized data chunks. Compared with fixed-sized blocking, the content-based chunking algorithms may be more time-consuming but more effective in duplicate elimination. This paper implements a content-based chunking algorithm in order to evaluate its CPU overhead and compare it with the simplest fixed-sized blocking method in term of duplicate elimination effectiveness.

There are two methods to determine whether two data chunks are identical: compare-by-value and compare-by-hash. Compare-by-value does a byte by byte comparison of the newly written chunk with the previously stored chunk. However, such a comparison is onerous since it's only possible by first reading the previously stored chunk from disk. This would make it extremely difficult if not impossible to maintain the needed throughput. Compare-by-hash relies on the use of cryptographic hash functions [14, 15, 16, 17] to identify duplicate chunks of data. Two chunks with the same hash value are assumed to be identical. This hash-based comparison is more efficient than otherwise the byte by byte comparison of chunks since a hash value is usually far smaller than the chunk itself and can be stored compactly in a hash table or in-memory cache. However, there exists the probability of hash collision that two different chunks generate the same hash value. Thus, the concern arises that data corruption can occur if a hash collision occurs, and additional bit-for-bit validation of original data for guaranteed data integrity should be offered [18, 19]. This paper analyzes the probability of hash collision when using compare-by-hash in large-scale storage systems. It indicates that if using appropriate cryptographic hash function, the compare-by-hash is feasible even employed in ultra-large-scale storage systems.

2. The Content-Based Chunking Algorithm

The most commonly used content-based chunking algorithm is the CDC algorithm [9], which uses hashing technique to determine chunk boundaries to divide the data stream into variable-sized data chunks. Specifically, it calculates the Robin hash [20] of each overlapping w byte segment of the file. When the low-order k bits of a segment's hash value equals to a constant pre-determined value, the

*This paper was supported by the Key Science-Technology Project of Henan of China under Grant No.112102210446.

segment constitutes a chunk boundary to divide the data stream into variable-sized chunks. The parameter k determines the expected chunk size that is 2^k bytes.

A. Algorithm Implementation

In order to evaluate its performance in terms of CPU overhead and chunking speed, we have implemented the CDC algorithm as following:

```
{input the file F to be divided;
 let anchorMask=00001FFFH; magicValue=61; M=2^32;
 p=17; w=48;
 FILE *f =open (F, 'r'); char b[w], c, c1;
 cleaning the chunk-queue L, integer-queue Q and char-
 queue C;
 if (length (f) <=w) { L.add(F); exit;}
 fgets(b,w,f);
 pos=w−1; RF= (b[0]×p^(w-1)+b[1]×p^(w-2)+…+b[w−1]) mod M;
 if (RF&anchorMask==magicValue)  Q.add(pos);
 for(i=0; i<=w−1;i++)  C.add(b[i]);
 while(c=fgetc(f)≠EOF){
 C.add(c); c1=C.move(); pos++;
 RF= ( p×RF + c − c1×p^w) mod M;
 if (RF&anchorMask==magicValue)  Q.add(pos);}
 firstpos=0;
 while (!empty(Q)){
 pos=Q.move();
 chunk=substring(f, firstpos, pos−firstpos);
 firstpos=pos;
 L.add(chunk);}
 return( L).}
```

In the above algorithm, parameter k is implicit in *anchorMask*, which equals to '00001FFFH', implying that k is 13 and the expected chunk size is 8KB.

B. Algorithm Performance

The algorithm uses a w-bytes-in-size sliding window passing along the file stream to seek out more naturally occurring internal chunk boundaries. This sliding-window approach makes the algorithm very compute-efficient.

Let $a_1, a_2 ... a_w ...$ be the byte stream of the file, the hash function used to identify chunk boundaries is as follow:

$$RF = hash_p(a_1, a_2 ... a_w) = (\sum_{i=1}^{w} a_i p^{w-i}). \quad (1)$$

The calculation takes places in a ring M with multiplication and addition. M and p are constants and p is a prime. This hash function has two important characteristics which can be used to speed up the chunking process.

- The calculation time is linear in w (1 multiplication and 1 addition).

$$hash_p(a_1, a_2 ... a_w) = (\sum_{i=1}^{w} a_i p^{w-i})$$

$$= (\sum_{i=1}^{w-1} a_i p^{w-i-1}) p + a_w$$

$$= hash_p(a_1, a_2 ... a_{w-1}) p + a_w$$

$$= (p((...(p(p \times a_1 + a_2) + a_3)...)) + a_w). \quad (2)$$

So, the time complexity of this hash algorithm is $O(w)$ with w be the number of bytes to be hashed.

- Easy to calculate consecutive w bytes.

$$hash_p(a_2, a_3 ... a_{w+1})$$

$$= p \times hash_p(a_1, a_2 ... a_w) + a_{w+1} - a_1 \times p^{w-1}. \quad (3)$$

The algorithm first computes the hash of the first w bytes using formula (1), and then it just needs to slide forward one byte to compute the hash of next w bytes using formula (3). To further speed up the process, we can compute a table of all possible values of $(a_i \times p^{w-1}) \bmod M$ for all 256 byte values and uses it throughout.

We run the algorithm on a computer with Inter Celeron 2.40GHz CPU, 512MB RAM and a 7200 RPM 80GB Seagate ST3802110A IDE disk. Time spent on content-defined chunking mainly includes detecting the chunk boundary over the sliding-window. The measured chunking speed is about 45.14MB/s with an average CPU utilization of 20.5%. While running the simple fixed-sized blocking algorithm with block size of 8KB on the same computer, we got a blocking speed of 45.27MB/s with an average CPU utilization of 17.3%. The above values are an average of 50 runs of the experiment, using different file sizes from 100 KB to 100 MB. The results indicate that the CPU overhead of the implemented CDC algorithm is reasonably low and the chunking speed is mainly limited by the disk I/O throughput before CPU becomes a bottleneck.

We also compared performance of the CDC algorithm and the fixed-sized blocking algorithm in term of duplicate elimination ratio. The data sets used for experiment include software distribution sources, web content, geographic information system data and research papers and reports in various formats and languages. We cannot present the detailed test figures here due to page limitation; instead, we give the experimental conclusion that the CDC algorithm outperforms the fixed-sized blocking algorithm in term of duplicate elimination ratio by a factor of 1.17 to 3.54 depending on the different data sets. Summary, selecting the CDC algorithm rather than the fixed-sized blocking algorithm is more appropriate for DE storage systems.

3. The Feasibility of Compare-By-Hash

A DE storage system could use the compare-by-value method to determine that two chunks are identical. But this method incurs heavy disk I/O overhead that would inevitably degrade the DE storage performance. To avoid this overhead, many DE storage systems [2, 3, 4, 5] rely on the compare-by-hash method to determine the identity of a chunk. Since there exists the probability of hash collision that may result in data corruption, the feasibility of using compare-by-hash in DE storage systems have caused much controversy in the research

community [18, 19, 21]. This paper argues that, to design a DE storage system with desired sized of capacity, we should select an appropriate collision-resistant hash function and analyze the probability of hash collision. If the probability of hash collision is extremely small, many orders of magnitude smaller than hardware error rate that is currently about 10^{-12}~10^{-15} [22, 23, 24]. When data corruption occurs, it will almost certainly be the result of undetected errors in hardware devices, and not from a collision.

A. Calculating the Probability of Hash Collision

Existing collision-resistant hash functions such as MD4 [14], MD5 [15], SHA-1[16], SHA-256, SHA-384 and SHA-512 [17] have good characteristic of uniformity, that is, the generated hash values are essentially random and independent of each other no matter how similar the inputs are.

Supposing the length of the hash value generated by a collision-resistant hash function is b bits, and Θ is the set of all possible hash values, we have $|\Theta|=2^b$. Let Ω be the set of all unique data chunks, $\forall D\in\Omega$, H(D) be the hash value of D, and Pr(H(D)=h) be the probability that H(D) equals to h, then we have:

$$\forall D\in\Omega,\ h\in\Theta,\ \Pr(\mathrm{H}(D)=h)=1/2^b. \tag{4}$$

Let $M\subset\Omega$ be the set of all the unique data chunks stored in the DE storage system, $|M|=n$, p be the probability that there exist hash collisions in the DE storage system, and then we have:

$$p=\Pr(\exists A, B\in M|((A\neq B)\&(\mathrm{H}(A)=\mathrm{H}(B)))). \tag{5}$$

Without loss of generality, assuming that the n unique data chunks (D_1, D_2…D_n) are stored into the DE storage system sequentially. Let E_k (k=1, 2…n) be the event that H(D_k) is not collided with the previously stored hash values, then we have:

$$\begin{aligned} p &= 1-\Pr(\bigcap_{k=1}^{n} E_k) \\ &= 1-\Pr(E_1)\times\Pr(E_2\mid E_1)\times\Pr(E_3\mid E_2E_1)\times... \\ &\quad\times\Pr(E_n\mid E_1E_2...E_{n-1}). \end{aligned} \tag{6}$$

Combining formulas (4) and (6), having:

$$p=1-(1-2/2^b)\times(1-3/2^b)\times...\times(1-(n-1)/2^b). \tag{7}$$

According to formula (7), when $1\le n\le 2^{(b+1)/2}$, there exists the following inequality:

$$0.316n(n-1)/2^b\le p\le 0.5n(n-1)/2^b. \tag{8}$$

Inequality (8) indicates that when $n<<2^{b/2}$ the probability of hash collision will be very small, while n is getting close to $2^{b/2}$, the probability of hash collision will experience a sharp increase.

It is not convenient to calculate p using formula (7) and inequality (8). For convenience of calculation, when $n<<2^{b/2}$, we can approximately assume that the events of hash collision are independent to each other. According to formula (4), the probability of hash collision by a pair of different data chunks is $1/2^b$, and n different data chunks have a total of $n(n-1)/2$ different pairs, then we have:

$$\begin{aligned} p &= 1-(1-1/2^b)^{n(n-1)/2} = 1-(1-1/2^b)^{-2^b\,n(n-1)/-2^{b+1}} \\ &\approx 1-\exp(-n(n-1)/2^{b+1}). \end{aligned} \tag{9}$$

When $n<<2^{b/2}$, $n(n-1)/2^{b+1}<<1$, then we have:

$$p\approx 1-\exp(-n(n-1)/2^{b+1})\approx n(n-1)/2^{b+1}. \tag{10}$$

B. The Probability of Hash Collision in DE Systems

Let C be the physical capacity of the large-scale DE storage system, s be the expected chunk size, then $n=C/s$. We calculate the probability of hash collision under different design parameters of the DE storage system using formula (10). The calculation results are listed in TABLE I.

TABLE I Hash Collision Probability in Different DE Storage Systems

C (byte)	s (byte)	n	H	b(bit)	p
2^{60}	2^{8}	2^{52}	MD5	128	2^{-25}
2^{60}	2^{13}	2^{47}	MD5	128	2^{-35}
2^{50}	2^{8}	2^{42}	MD5	128	2^{-45}
2^{50}	2^{13}	2^{37}	MD5	128	2^{-55}
2^{60}	2^{8}	2^{52}	SHA-1	160	2^{-57}
2^{60}	2^{13}	2^{47}	SHA-1	160	2^{-67}
2^{50}	2^{8}	2^{42}	SHA-1	160	2^{-77}
2^{50}	2^{13}	2^{37}	SHA-1	160	2^{-87}
2^{50}	2^{8}	2^{42}	SHA-256	256	2^{-173}
2^{50}	2^{13}	2^{37}	SHA-256	256	2^{-183}
2^{72}	2^{8}	2^{64}	SHA-256	256	2^{-129}
2^{72}	2^{13}	2^{59}	SHA-256	256	2^{-139}

The results in TABLE I show that for petabyte-scale DE storage systems (2^{50}bytes≤C<2^{60}bytes) with an expected size of 256 bytes or 8KB, using MD5 hash function with 128 bits hash value, the probability of hash collision is in the range of 2^{-25}~2^{-55}, which is higher than the undetected hardware error rate (currently about 10^{-12}~10^{-15} ,i.e., 2^{-40}~2^{-50}) in most cases. This indicates that using 128 bits length MD5 hash in petabyte-scale DE storage systems is not safe enough according to the reference criterion of hardware error rate. However, when using 160 bits length SHA-1 hash for petabyte-scale DE storage systems with an expected size of 256 bytes or 8KB, the probability of hash collision will be in the range of 2^{-57}~2^{-87}, far smaller than the undetected hardware error rate. While using more strong SHA-256 hash function，even for a ultra-large-scale DE storage system with 2048EB (2^{72} bytes) physical capacity that is larger than the total volume of digital data (about 1800EB) generated by the world in 2011[1], the probability of hash collision is still extremely small, only 2^{-129} and 2^{-139} for an expected chunk size of 256 bytes and 8KB respectively. Considering that the

storage and CPU overheads of the 256 bits SHA-256 hash value is relatively higher than that of the SHA-1 hash value, it is appropriate to use SHA-1 hash function for compare-by-hash in petabyte-scale DE storage systems. Summary, the hash functions that can be used in the DE storage systems include standards such as SHA-1, SHA-256 and others. These provide a far smaller probability of data loss than the risk of an undetected hardware error and can be in the order of 2^{-87} (using SHA-1) or 2^{-183} (using SHA-256) per petabyte (2^{37} data chunks with an expected chunk size of 8KB) of data. Therefore, using compare-by-hash in DE storage system to improve system throughput is reasonably safe and feasible.

4. Conclusions

An ideal Duplicate Elimination storage system should be not only space-efficient (i.e., with high duplicate elimination ratio) but also with high performance. So, designing a DE storage system should take into consideration many possible alternatives. This paper studied two important kinds of alternatives: fixed-sized blocking versus content-based chunking, and compare-by-value versus compare-by-hash.

This paper implemented a content-based chunking algorithm and evaluated its performance in terms of calculation efficiency and duplicate elimination ratio. The experimental results showed that the implemented CDC algorithm incurs reasonably low CPU overhead, and it outperforms the fixed-sized blocking algorithm in term of duplicate elimination ratio by a factor of 1.17 to 3.54 depending on the different data sets. This suggests that selecting the content-based chunking algorithm rather than the fixed-sized blocking algorithm is more appropriate for DE storage systems.

To identify duplicate data chunks, compare-by-hash is more efficient than compare-by-value. But compare-by-hash may result in data corruption due to hash collision, and hence has caused much controversy in the research community. This paper studied the probability of hash collision in petabyte-scale DE storage systems through theoretical analysis. The results indicate that using appropriate cryptographic hash function such as SHA-1 can provide far smaller probability of data corruption than the risk of an undetected hardware error. So, it is reasonably feasible to employ compare-by-hash in large-scale DE storage systems to improve duplicate elimination performance.

References

[1] IDC Predictions 2012: Competing for 2020, http://cdn.idc.com/research/Predictions12/Main/downloads/IDCTOP10Predictions2012.pdf (December 2011).

[2] B. Zhu, H. Li, H. Patterson, Avoiding the disk bottleneck in the data domain deduplication file system, in: Proceedings of the 6th USENIX Conference on File And Storage Technologies, 2008.

[3] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, Y. Wan, DEBAR: a scalable high performance deduplication storage system for backup and archiving, in: In IEEE International Symposium on Parallel and Distributed Processing, 2010.

[4] P. Shilane, M. Huang, G. Wallace, W. Hsu, Wan optimized replication of backup datasets using stream-informed delta compression, in: Proceedings of the 10th USENIX Conference on File And Storage Technologies, 2012.

[5] K. Srinivasan, T. Bisson, G. Goodson, K. Voruganti, idedup: Latency-aware, inline data deduplication for primary storage, in: Proceedings of the 10th USENIX Conference on File And Storage Technologies, 2012.

[6] A. C. amd Vivekenand Vellanki and Z. Kurmas, "Protecting file systems: A survey of backup techniques," in In Proceedings Joint NASA and IEEE Mass Storage Conference, March 1998.

[7] S. Quinlan, S. Dorward., Venti: a new approach to archival storage, in: Proceedings of the USENIX Conference on File And Storage Technologies, 2002.

[8] B. HONG, I. PLANTENBERG, I. I. E. LONG, M. SIVAN-ZIMET, Duplicate data elimination in a san file system, in: Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, 2007.

[9] A. Muthitacharoen, B. Chen, and D. Mazieres. A low bandwidth network file system. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01), Oct. 2001. 174–187.

[10] C. Liu, Y. Lu, C. Shi, G. Lu, Du, D.H.C. and D. Wang. ADMAD: Application-Driven Metadata Aware De-duplication Archival Storage System. In Proceedings of the Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os, 2008.

[11] ESHGHI, K. A framework for analyzing and improving content-based chunking algorithms. Tech. Rep. HPL-2005-30(R.1), Hewlett Packard Laboratories, Palo Alto, 2005. 2-11.

[12] Deepak R. Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. Improving duplicate elimination in storage systems. ACM Transactions on Storage. vol. 2, no. 4, pp. 424-448, 2006.

[13] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In Proceedings of the 8th Conference on File and Storage Technologies, February 2010.

[14] R. Rivest. The MD4 message digest algorithm. Request For Comments (RFC) 1186, IETF, Oct. 1990.

[15] R. Rivest. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, Apr. 1992.

[16] Secure Hash Standard, U.S. Department of Commerce N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[17] NIST FIPS 180-2: Secure Hash Standard, Aug. 2002.

[18] Henson, V. An analysis of compare-by-hash. In HotOS: Hot Topics in Operating Systems, USENIX, 2003. 13–18.

[19] S. Rhea, R. Cox, A. Pesterev, Fast, inexpensive content-addressed storage in foundation, in: Proceedings of the 2008 USENIX Annual Technical Conference, Boston, Massachusetts, 2008, pp. 143–156.

[20] BRODER, A. Some applications of Rabin's fingerprinting method. In Sequences II: Methods in Communications, Security, and Computer Science, R. Capocelli, A. D. Santis, and U. Vaccaro, Eds. Springer-Verlag, 1993, pp. 143–152.

[21] J. Black. Compare-by-hash: A reasoned analysis. In USENIX Annual Tech. Conf., 2006. 85-90.

[22] C. M. Riggle and S. G. McCarthy. Design of error correction systems for disk drives. IEEE Transactions on Magnetics, July 1998. 34(4):2362–2371.

[23] IBM. IBM OEM hard disk drive specification for DPTA-3xxxxx 37.5 GB–13.6 GB 3.5 inch hard disk drive with ATA interface, revision (2.1). (Deskstar 34GXP and 37GP hard disk drives), July 1999.

[24] Hitachi Global Storage Technologies. Hitachi hard disk drive specification: Deskstar 7K400 3.5 inch Ultra ATA/133 and 3.5 inch Serial ATA hard disk drives, version 1.4, Aug. 2004.