DOI: 10.24996/ ijs.2017.58.4C.19

ISSN: 0067-2904

Evaluation of Two Thresholds Two Divisor Chunking Algorithm Using Rabin Finger print, Adler, and SHA1 Hashing Algorithms

Hala Abdulsalam*, Assmaa A. Fahad
Department of Computer Science, College of Science, University of Baghdad, Baghdad, Iraq.

Abstract

Data deduplication is a data reduction technology that is worked by detecting and eliminating data redundancy and keep only one copy of these data, and is often used to reduce the storage space and network bandwidth. While our main motivation has been low band-width synchronization applications such as Low Bandwidth Network File System (LBNFS), deduplication is also useful in archival file systems. A number of researchers have advocated a scheme for archival. Data deduplication now is one of the hottest research topics in the backup storage area. In this paper, A survey on different chunking algorithms of data deduplication are discussed, and studying the most popular used chunking algorithm Two Threshold Two Divisor (TTTD), and evaluated this algorithm using three different hashing functions that can be used with it (Rabin Finger print, Adler, and SHA1) implemented each one as a fingerprinting and hashing algorithm and then compared the execution time and deduplication elimination ratio which was the first time this comparison performed and the result is shown below.

Keywords: Big Data, Deduplication, Rabin Finger print, Adler, SHA1.

تقييم خوارزمية (TTTD) بأستخدام ثلاث خوارزميات هاش مختلفة

هلا عبد السلام جاسم*، أسماء عبد الله فهد
قسم علوم الحاسبات، كلية العلوم، جامعة بغداد، بغداد، العراق.

الخلاصة

الغاء البيانات المكررة هي تقنية تقليل حجم البيانات عن طريق استكشاف و حذف البيانات المتكررة والاحتفاظ بنسخة واحده فقط من هذه البيانات ، وغالبا ما تستخدم للتقليل من مساحة التخزين وكمية البيانات المنقولة عبر الانترنيت. في حين كان الدافع الرئيسي لدينا هو الحد من كمية البيانات المنقولة عبر الشبكات بأستخدام تطبيقات مثل (LBNFS)، تقنية حذف البيانات المكررة تستخدم ايضا في أنظمة ارشفة البيانات. اوصى عدد من الباحثين باستخدم الخوارزميات لأرشفه البيانات التي تكون مبنية على اساس تقنية حذف البيانات المكررة . تعتبر تقنية حذف البيانات المكررة من اهم العناوين في مجال التخزين الاحتياطي. في هذا البحث، تم مناقشة وعرض خوارزميات التقطيع المختلفه في تقنية حذف البيانات المكررة، ودراسة خوارزمية التقطيع الاكثر استخداما (TTTD)، واختبار وعرض نتائج هذة الخوارزمية مع ثلاث خوارزميات هاش بديله يمكن استخدامها مع هذه الخوارزمية (Rabin Finger Print , Adler , SHA1) . اجريت هذة الدراسة لأول مرة واستخدمنا هذة الخوارزميات كخوارزميات تقطيع و خوارزميات هاش بنفس الوقت والنتائج وسوف نعرض النتائج في هذة البيبر.

*Email: hala_aljumaily@yahoo.com

1. Introduction

Over the last several years, we have witnessed an unpredictable growth of the volume of stored digital data universe. A recent study pointed that the amount of digital data that are generated in 2002 was about 5 exabytes, which is nearly double the volume of data created in 1999 [1].And with the heading toward digitization and Internet of Things (IoT) the digital universe is doubling every two years in term of size, and by 2020 it reaches 44 zettabytes, or 44 trillion gigabytes, which is about 50 time by it now [2]. Most of these digital data is redundant, which is formed as Archival or backup data or any other redundant data, One study point that about to 70% of data collected from 1000 computer of an enterprise, and about to 60% of Network outgoing and 30% on incoming traffics is redundant [3] .The archival data is an increasing part of digital universe, which is unchangeable data that keep stored for long time for the sake of legal or archival purposes. As the researchers observed we can take advantage of these data, in order to improve the storage efficiency. Deduplication techniques was proposed for this purpose.

Data Deduplication technique are mainly used in the disk-based backup system because of its cost-effective space utilization, it proposed to eliminate Enter-file redundancy by exploiting the high degree of similarity among archival data in order to improve storage efficiency because traditional data compression technique can only find the intra-file redundancy [4] .Deduplication may be occur either inline or post-process. In in-line the duplication process (hash calculations and lookup) done in the real-time. With post-process or (offline) deduplication, the whole data is stored on the storage device when it arrived and then duplication process at a later time will be performed. Deduplication process take good amount of time causing the degradation of system performance, post process will solve this problem by the way it work. On the other hand storing redundant data on a system that nearly reached its full capacity is not recommended and may cause a problem and this can be consider as an advantage of in-line deduplication over post process deduplication.

The data set used in this work is a different versions of Linux kernel that continent text files with different size and types which is suitable to test the efficiency of deduplication system.

The aim of this paper can be summarized as follows: (i) provide a brief survey on data deduplication algorithm and its development (ii) explain in detail the TTTD algorithm and its performance with different hash function (Rabin Fingerprint, Adler and SHA-1) (iii) discussing the results.

2. Deduplication System:

The full deduplication system is consists of three parts:

1. **Chunker**: the Chunker is the most important part in this system, it splits the data into numbers of chunks and assign each chunk a unique hash value identifier.
2. **Lookup table**: save the file as a key with its hash value. For example: “C:\\File1\\chunk0 Hash0”.
3. **Matching**: This part compare the new file with the chunks of file that has the same file name and type which is already stored in the database of the system. If the identifier of the chunk is found, the chunk will be deleted and a logical reference is added to the matched one. Otherwise it will be considered as a new chunk, and added to the system database and the lookup table. The key objective of good deduplication system is number of chunks in it , which effect of the performance of the system .

3. Chunking Algorithms

The main challenges of deduplication system is chunking, there are several chunking strategies such as the File level, fixed-sized, content-defined and content aware. The essential idea of these chunking strategies is to break a file into small chunks and then find out the redundancy by fingerprint comparison.

Fingerprint is a hash value that is specialized for the specific chunk, since the hash functions are collision resistant, having the same fingerprint means (with a very high probability) the two data blocks are identical. So to avoid wasting resources on storage of duplicate data only one of them will be stored. If the data block is different from another their fingerprint would be different, so both of them will be store· [5] .

In general we can classify it in two categories:

3.1- Single Instant Store (File Level Deduplication): For SIS entire files are given a hash signature using hash function such as MD5 or SHA-1. This method used in Windows 2000 [6]. It avoids maximum metadata lookup overhead and CPU usage. Also, it reduces the index lookup process. The problem of the SIS is its low deduplication Ratio because it fails when a small portion of the file is changed, the whole file will be consider as new file.

3.2- Chunk level deduplication: breaks the file into number of chunks according to the algorithm that use, there are three kind of it:

3.2.1- Fixed-size chunking: splits files into equally sized chunks. The chunk boundaries are based on offsets like 4, 8, 16 kB, etc. This method used in Venti and Oceanstore [6], it improved Deduplication Ratio significantly compared with SIS. However, the effectiveness of this approach is highly sensitive to the sequence of edits, for example, an insertion of a single byte at the beginning of a file can change the content of all chunks in the file resulting in no sharing with existing chunking.

3.2.2- Content aware Chunking: based on similarity detection between data objects, and then found the deltas between them using delta encoding and store it instead of entire data.

3.2.3- Content Defined Chunking (CDC): employs hash function to choose partitioning points in the object. CDC restrict the effect of inserting or deleting characters to the regions where changes have been made that cause one or two chunk only to consider new but the rest of chunks may remain as it was. CDC was first applied in LBFS, lots of chunking algorithms are developed based on it to improve Deduplication Ratio. CDC algorithm uses more CPU resources. Based on the characteristics of the file such as content, size, image, color, etc. but it the most used because it has the best Deduplication Rate, below the most famous kind of CDC [7]:

❖ **Basic Sliding Window (BSW):** Fingerprint computed by a hash function such as Rabin then a predefined condition is tested if the value of fingerprint satisfy that condition it consider as a breakpoint (chunk boundary), two main factors used to define that condition window size and a divisor D if (Hash [window size string] % D = D-1) then it a breakpoint, If not then slide the window size one byte until a breakpoint found.

❖ **The Two Threshold Two Divisor (TTTD):** an improvement of BSW, it define three more factors, two of them work as threshold (Tmin and Tmax) and the third is a second divisor (Ddash). TTTD guarantees that no chunk smaller than (Tmin) and no chunk larger than maximum size (Tmax). Ddash variable is used to avoid any large abnormal block size [5].

❖ **Two Thresholds and Two Divisors with SwitchP (TTTD-S):** presented as an improvement on the TTTD , it reduce 50% of large chunks size and 7% of the running time, TTTD take an expensive calculation and running time, only the second divisor Ddash take about to 10% of total running time, so the new divisor (SwitchP) appears as the solution, it will switch main divisor D and the second divisor Ddash values to 1/2 of the original when the algorithm reaches a specific point , then return it to original after finding a breakpoint increasing the main divider probability to happen earlier led to skip some calculation and saving time [8].

❖ **Bimodal chunking algorithm**: Bimodal algorithms perform content-defined chunking in a scalable manner, it can dynamically change the expected chunks size. It combine chunk that have different size together. For non-duplicated chunks it divide it into smaller ones in order to find more redundancy [9].

❖ **Multimodal Content Defined Chunking (MCDC) Algorithm:** it consider as improvement to bimodal chunking algorithm.at the beginning they divide the data into fixed size chunks and find the Compression ratio (CR) for each one apart from the others. This method led to shift boundary problem which is solved by dividing the data stream into variable size block using Uni-modal chunking and compute the compression ratio for each one after that second level will start for each chunk, start variable size chunking by using fingerprint by this way system overhead will be reduced and about to 29.1% to 92.4% of chunk numbers will be reduced as well, but the deduplication ratio will sill efficient.

❖ **Leap-based CDC Algorithm:** Leap based chunking algorithm improves the deduplication performance of BSW. The leap based added two parameters M and Pw, these Parameters determine the performance and chunk size of leap based CDC, where M is the number of satisfied window and the Pw is the probability that window satisfied. They found that the optimal value of M=24 and Pw =3/4. By adding a secondary judgment function the computational overhead is reduced and the deduplication ratio is maintained [10].

4. Two Threshold Two Divisor Algorithm:

TTTD was first proposed by HP laboratory in 2005, it take four basic variables that determine its behavior (Min, Max, D and Ddash) parameter values. The optimal values of these parameters are (460, 2800, 540, ad 270) respectively [9]. The TTTD algorithm is consists of the following steps [9]:

- ❖ Read file as one character at time.
- ❖ Skip first Min boundary.
- ❖ When reach Min value start to compute finger print value for last window size.
- ❖ If (finger print) mod D = D-1 then consider it as a chunk boundary and add breakpoint And go to 2
- ❖ Else if (finger print) mod Ddash =Ddash -1 then consider it as backup breakpoint and continue reading next character and update window size by deleting first character and append new one and compute new (finger print).
- ❖ If Max boundary reached, if there is any backup break point use it, else use max as a break point boundary; then go to 2.

Figure-1 shows these steps.

5. Alternative Hash functions Used with TTTD:

Basically TTTD works using a specific kind of hashing algorithm; Rolling Hash, to find if the text matching the pattern or not. Using Hash function may lead to hash collision then a byte to byte comparison operation for the substring and the pattern must be performed, which takes long time.

For a given data block of size S bytes, rolling hash calculates the fingerprints over a sliding window of size W, where (W smaller than or equal to S). The size of the sliding window is variable, and can be within the range of 12 to 64 bytes reading one byte at a time [9]. The first W bytes send to hash then a sliding window rolling one byte removing the first byte of the W size and append the new one and send it again to hash algorithm to update its fingerprint or hash value. Fingerprint is considered as the chunk boundary or breakpoint.

In this paper, three hash algorithms will be implemented and the results will be discussed.

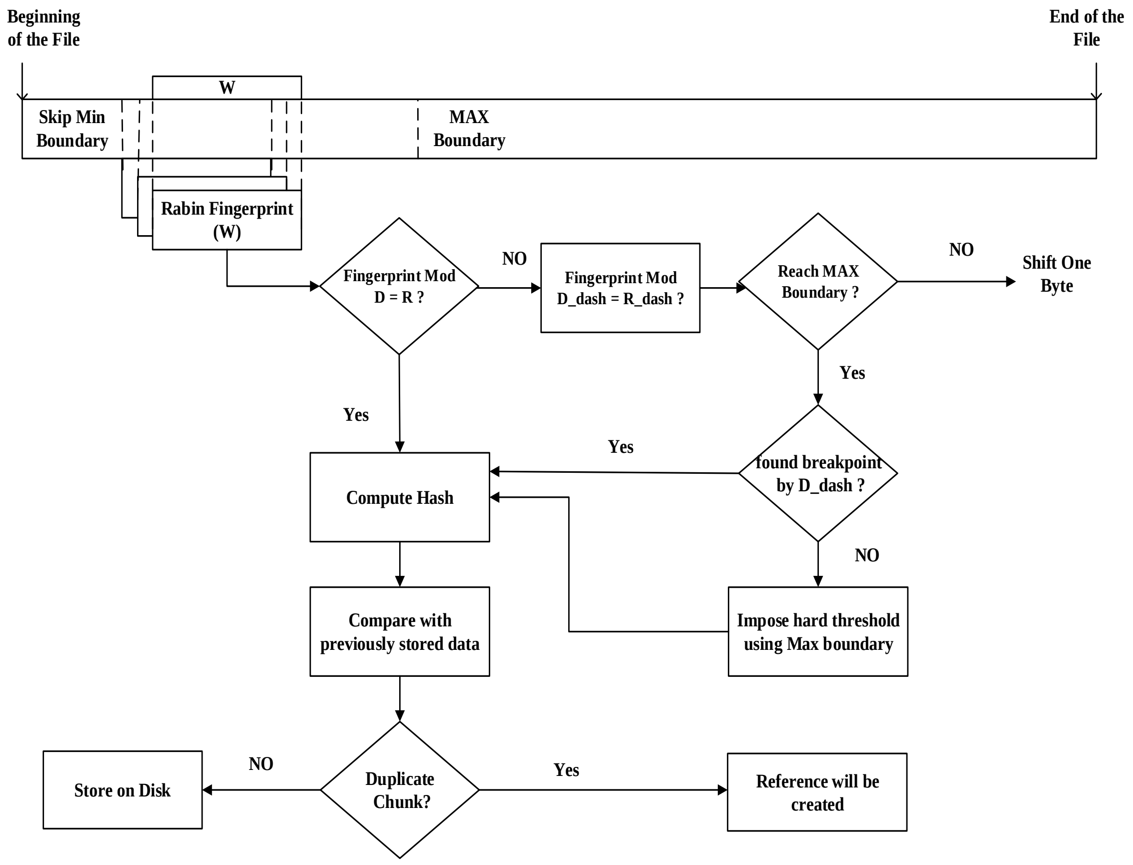

Figure 1- TTTD Deduplication Algorithm

5.1- Rabin Fingerprint: The most used hash with TTTD is Rabin Fingerprint, it is string matching algorithm used in LBFS,[11] it work as follow:

- Calculates a rolling Fingerprint over data. For first substring of length K where (K is Substring size) Fingerprint is computed by using the formula:

Fingerprint (Substring of length k) = (Substring [0] × Prime ^K-1) + (Substring [1] × Prime^k-2) + (Substring [2] × Prime ^ k-3)..... (Substring [0] × Prime ^ 0).

- Calculating a Fingerprint according to a previous one as follows:

❖ Subtract first byte value that has slid out while rolling the sliding window, the value equal to [ASCII Code of character * Prime ^ K] because it rolling about K time before it eliminate.

❖ Multiply the result from first step by the prime number.

❖ Add the value of the new character that has appended to the substring. The value of the new byte equal to [ASCII Code of byte * Prime ^0] as it is the first arrive to it.

5.2- Adler-32 checksum: is part of the widely used zlib compression library. The Adler checksum uses a prime modulus in an attempt to get better mixing of the checksum bits. A is initialized to 1 and each addition is done modulo 65521 it work as follow :

- Calculates a rolling Fingerprint over data. For first substring of length K where (K is Substring size) Fingerprint is computed by using the formula:

```
For (index = 0 to K)
{a = (a + Substring [index]) % MOD_ADLER;
b = (b + a) % MOD_ADLER}
Fingerprint = (b << 16) | a;
```

- Calculating a Fingerprint according to a previous one as follows:

```
a = Old hash value;
b = (a >> 16) & 0xffff;
a &= 0xffff;
a = (a - Substring [0] + ASCII (New Char)) % MOD_ADLER;
b = (b - (K* Substring [0]) + a - 1) % MOD_ADLER;
```

5.3- Secure Hash Algorithms (SHA-1): is a cryptographic hash function designed by the National Security Agency (NSA). SHA-1 hash function is the most widely used of the existing SHA hash functions, it takes a variable length input message and produces a fixed size (160 bit length) output message called the hash or the message digest of the original message. The basic SHA-1 algorithm is presented as follows: [12]

- Initializing the five sub-registers of the first 160-bit register X labeled H_0, H_1, H_2, H_3, and H4 as follows:

H_0=67452301; H_1=EFCDAB89; H_2=98BADCFE; H_3=10325476; H_4=C3D2E1F0;

- Iterates each of 512 message bits blocks (m_0, m_1, m_2 … m_{n-1}). For each one do :

❖ Write m_j as a sequence of sixteen 32-bit words, $m_j = W_0 \parallel W_1 \parallel W_2 \parallel \dots \parallel W_{15}$

❖ Compute the remaining sixty four 2-bit words as follows:

$W_t = (W_{t-3} \text{ xor } W_{t-8} \text{ xor } W_{t-14} \text{ xor } W_{t-16})$

Cyclic shift of W_t by 1 i.e. $S^1(W_t)$

❖ Copy the first 160 bit register into the second register as follows:

A= H_0; B= H_1; C=H_2; D=H_3; E= H_4;

This step involves a sequence of four rounds, corresponding to four intervals 0<=t<=19, 20<=t<=39, 40<=t<=59, 60<=t<=79. Each round takes as input the current value of register X and the blocks W_t for that interval and operates upon them for 20 iterations as follows:

```
For t = 0 to 79 do:
{$T=S^5 (A) + f_t (B, C, D) + E + W_t + K_t$
E=D; D=C; $C= S^{30} (B)$; B=A; A=T}
```

❖ Once all four rounds of operations are completed, the second 160-bit register (A, B, C, D, E) is added to the first 160-bit register (H_0, H_1, H_2, H_3, H_4) as follows:

H0 = H0 + A; H1 = H1 + B; H2 = H2 + C; H3 = H3 + D;
H4 = H4 + E;

- Once the algorithm has processed all of the 512-bit blocks, the final output of X becomes the 160-bit message digest.

6. The Data Sets

The Data Sets used in this paper are:

1. Five versions of GNU (Emacs) files including versions 22.1, 22.2 22.3, 23.1 and 24.1, the total data size was about 580 MB consist of 16,296 Files, 327 Folders continent different type.
2. Eight versions of 3DLDF files of GNU including versions 1.1.3 ,1.1.4 ,1.1.5 ,1.1.5.1 , 2.0 ,2.0.1 , 2.0.2 and 2.0.3.The total data size was about 2.27 GB consist of 5,795 Files and 63 Folders continent different type.

The files within the data sets are with different sizes starting from zero byte to very large file in order to test all cases;

Table *1* shows the characterization of each data set.

Table 1-Characteristics of the used Data Set

Data Set	**On line link**	**No. of Files and folders**	**Total input size**
Versions of Emacs of GNU	**http://www.gnu.org/**	**16,296 Files, 327 Folders**	**580 MB**
Versions of 3DLDF of GNU	**https://www.kernel.org/**	**5,795 Files, 63 Folders**	**2.27 GB**

7. Experimental Result and discussion

In this paper TTTD deduplication algorithm is implemented using three different hashing algorithms: Rabin finger print, Adler, and SHA-1, with two different datasets are used: Versions of Emacs of GNU, and Versions of 3DLDF of GNU. The three hash methods are implemented in an Intel core i7 CPU, 16 GB RAM and 1TeraByte HDD in Windows 10 environment with C++ language in Visual Studio 2017 development tool.

The results in **Table-2** are produced by implementing TTTD deduplication algorithm with Rabin Fingerprint, Adler and SHA-1 hashing algorithms using Versions of Emacs of GNU data set.

Table 2- Result of data Set 1

Hash / **performance**	**Rabin Finger Print**	**Adler Rolling Hash**	**SHA1**
Size of input Data in bytes	**608,528,261**	**608,528,261**	**608,528,261**
Size of output Data in bytes	**342,963,573**	**348,633,873**	**342,094,332**
Deduplication Rate	**1.7743**	**1.745**	**1.778**
TTTD Elapsed Time in second	**2208**	**1743**	**7712**
Whole program time in second	**3571**	**3343**	**9339**
Total Number of chunk	**623922**	**619761**	**622457**
Total Number of chunk Using D	**598801**	**594975**	**597524**
Total Number of chunk Using D Dash	**8510**	**8337**	**8398**
Total Number of chunk Using Max	**315**	**153**	**239**

Table -3 shows the results produced by implementing TTTD deduplication algorithm using Versions of 3DLDF of GNU data set.

Table 3- Result of Data Set 2

Hash / performance	Rabin Finger Print	Adler Rolling Hash	SHA1
Size of input Data in bytes	**2442270245**	**2442270245**	**2442270245**
Size of output Data in bytes	**860227050**	**862132181**	**860373952**
Deduplication Rate	**2.839**	**2.832**	**2.838**
TTTD Elapsed Time in second	**3192**	**1073**	**29428**
Whole program time in second	**9798**	**5783**	**38017**
Total Number of chunk	**2468026**	**2477446**	**2440705**
Total Number of chunk Using D	**2424864**	**2440286**	**2397265**
Total Number of chunk Using D Dash	**35686**	**30843**	**37031**
Total Number of chunk Using Max	**1681**	**522**	**614**

As noticed the same results are obtained approximately for the three hashes. For each data set the experiments repeated 10 times in order to obtain reliable results. Because of the natural of the TTTD algorithm and its dependency on the content of the file, the number of chunks will be considered during the test operation, when the dataset remains unchanged, the algorithm will produce the same results. But for the execution time the average of the resulted time of tests was considered as the final execution time.

From the results shown in Table -2 and Table -3, it is notes that if the main devisor D used to find the breakpoint it will produce minimum number of chunks with larger chunk size than the chunks produced by D Dash; this case will leads to minimum CPU overhead, but at the same time it will reduce the deduplication rate because large chunk size miss to find matches as the small chunk size. In another side maximum number of chunk, with small chunk size produces maximum CPU overhead with high deduplication rate. It important to mention out the three special cases that appeared during this work:

❖ The files with size “Zero“, will be considered as one empty chunk in order to reconstruct this file during the decompression operation.

❖ The size of the file is smaller than the minimum threshold, this file will be considered as one chunk.

❖ The remaining part after the last breakpoint is small than the minimum threshold, will be considered as one small chunk.

8. Conclusion

From the above, if the deduplication system focuses on deduplication rate SHA-1 is the most suitable hashing algorithm that can be used with TTTD algorithm. Otherwise if the system prefers speed over the deduplication rate then Adler rolling hash is the best. Between the two Rabin Fingerprint consider as the middle of the two it produce deduplication rate near to SHA-1 with less time that is almost as much as Adler Rolling hash.

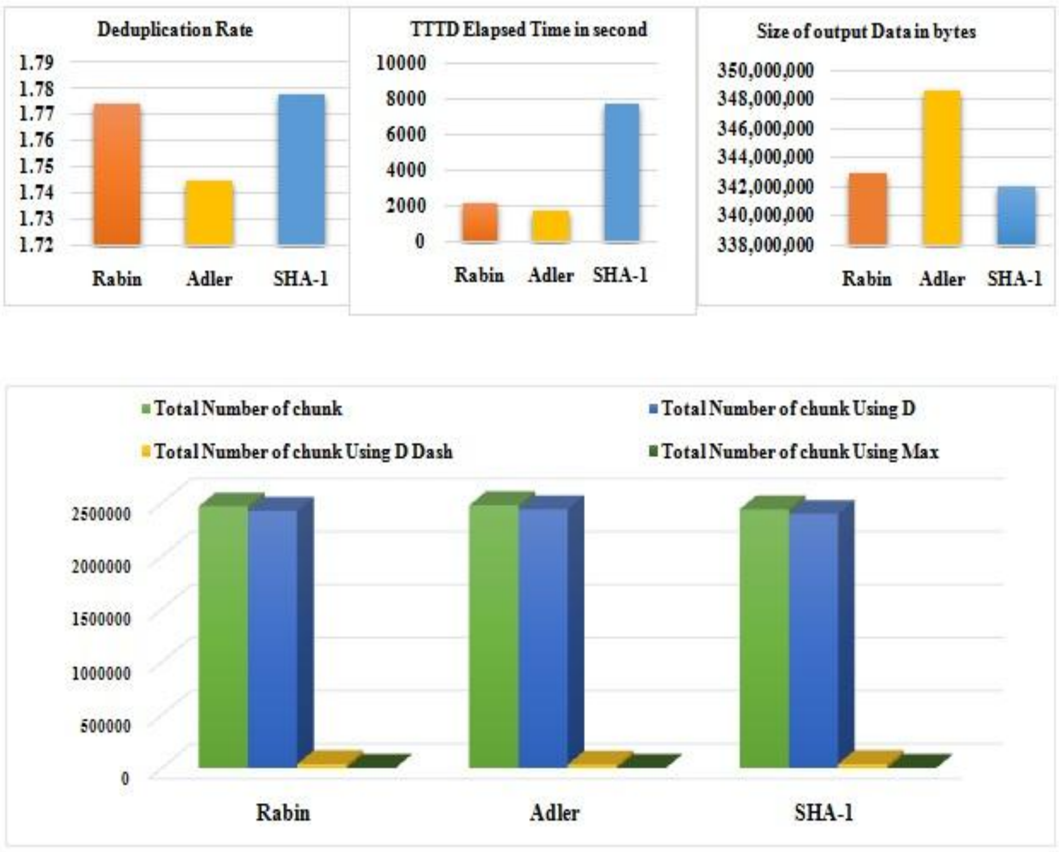

Figure 2- Result of Data set 1

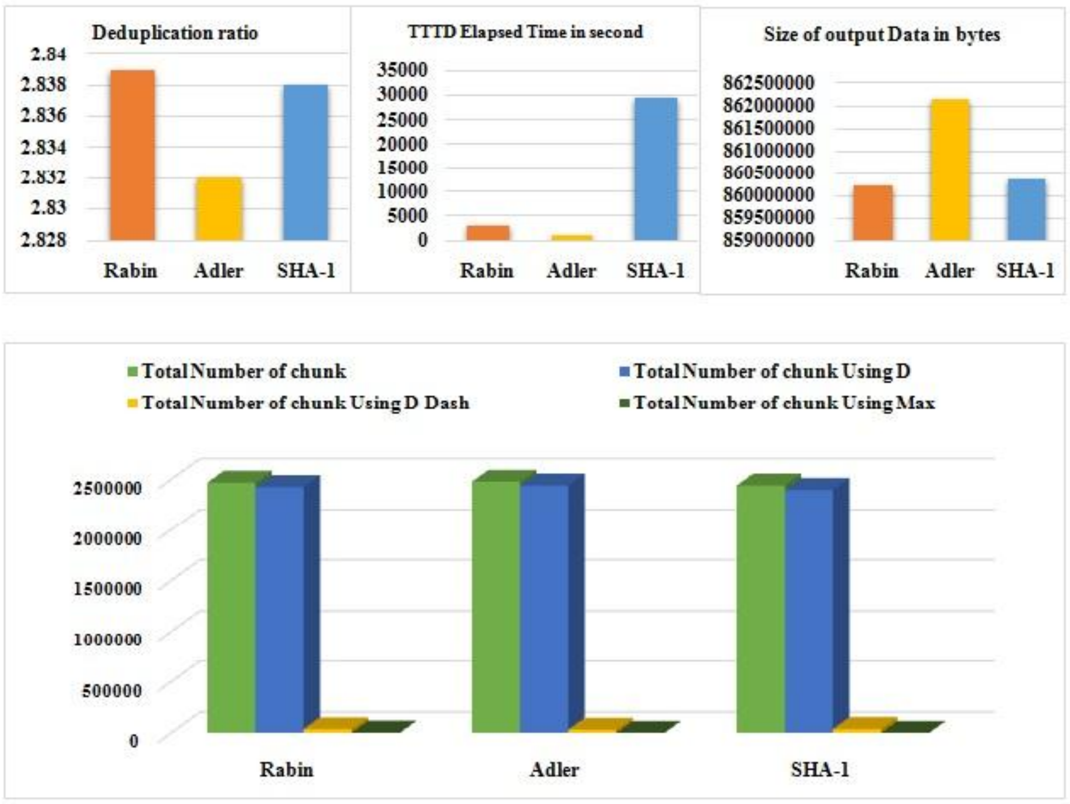

Figure 3-Result of Data set 2.

References

1. Quinlan, S. and Dorward, S. **2002**. Venti: A New Approach to Archival Storage. *FAST*, **2**: 89-101.
2. Gantz, J., & Reinsel, D. **2012**. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007, 1-16.
3. Kim, D., Song, S. and Choi, B.-Y. **2016**. *Data deduplication for data optimization for storage and network systems*. Springer International Publishing Switzerland 2017.
4. You, L. and Karamanolis, C. T. **2004**. Evaluation of Efficient Archival Storage Techniques. *MSST*, pp. 227-232.
5. Kave, E. and Khuern, T. H. **2005**. A framework for analyzing and improving content-based chunking algorithms. *International Enterprise Technologies Laboratory*, HP Laboratories Palo Alto, Tech. Rep.
6. Wang, L., Dong, X., Zhang, X., Guo, F., Wang, Y. and Gong, W. **2016**. A Logistic Based Mathematical Model to Optimize Duplicate Elimination Ratio in Content Defined Chunking Based Big Data Storage System. *Symmetry*, **8**(7): 69.
7. Venish, A. and Sankar, K. S. **2016**. Study of Chunking Algorithm in Data Deduplication. Proceedings of the International Conference on Soft Computing Systems, 13-20.
8. Moh, T.-S. and Chang, B. **2010**. A running time improvement for the two thresholds two divisors algorithm. Proceedings of the 48th Annual Southeast Regional Conference, (p. 69).
9. Kruus, E., Ungureanu, C. and Dubnicki, C. **2010**. Bimodal Content Defined Chunking for Backup Streams. *Fast*, pp. 239-252.
10. Yu, C., Zhang, C., Mao, Y. and Li, F. **2015**. Leap-based content defined chunking—theory and implementation. Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on, (pp. 1-12).
11. Dang, Q. **2013**. Changes in federal information processing standard (FIPS) 180-4, secure hash standard*.* *Cryptologia*, **37**: 69-73.
12. Eshghi, K. and Tang, H. K. **2005**. A framework for analyzing and improving content-based chunking algorithms. Hewlett-Packard Labs Technical Report TR, 30.