

Received December 22, 2019, accepted December 30, 2019, date of publication January 3, 2020, date of current version January 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2963625

Function of Content Defined Chunking Algorithms in Incremental Synchronization

CHANGJIAN ZHANG^{ID}, DEYU QI^{ID}, WENLIN LI^{ID}, AND JING GUO^{ID}

Department of Computer Science and Engineering, South China University of Technology, Guangzhou 510000, China

Corresponding author: Deyu Qi (csa@scut.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61070015, in part by the Guangdong Provincial Frontier and Key Technology Innovation Special Funds Major Science under Grant 201802020035, and in part by the Technology Project under Grant 2014B010110004.

ABSTRACT Data chunking algorithms divide data into several small data chunks in a certain way, thus transforming the operation of data into the one of multiple small data chunks. Data chunking algorithms have been widely used in duplicate data detection, parallel computing and other fields, but it is seldom used in data incremental synchronization. Aiming at the characteristics of incremental data synchronization, this paper proposes a novel data chunking algorithm. By dividing two data that need synchronization into small data chunks, comparing the contents of these small data chunks, different ones are the incremental data that need to be found. The new algorithm determines to set a cut-point based on the number of 1 contained in the binary format of all bytes in an interval. Thus it improves the resistance against the byte shifting problem at the expense of the chunk size stability, which makes it more suitable for the incremental data synchronization. Comparing this algorithm with several known classical or state of art algorithms, experiments show that the incremental data found by this algorithm can be reduced by 32%~57% compared to the others with same changes between two data. The experimental results based on real-world datasets show that PCI improves the calculation speed of classic Rsync algorithm up to 70%, however, with a drawback of increasing the transmission compression rate up to 11.8%.

INDEX TERMS Data synchronization, chunking algorithm, data backup, increment.

I. INTRODUCTION

Data chunking algorithm reads the data as a byte stream. In the process of reading, a single byte or multiple bytes are selected as a boundary of chunk based on certain conditions. The data between two ends of adjacent boundaries ends is called a chunk. Chunking algorithms have been widely used in many fields, such as network transmission [1]–[3], data storage system [4]–[6], data synchronization system [7]–[9], cache system [10]–[12], text recognition [13]–[15] and so on. For example, in the field of text recognition, when parsing natural language texts, a sentence needs to be chunked to extract the subject, predicate, object and other key phrases in it, and then get the true meaning of the sentence by grammar analysis.

A. CLASSIFICATION OF DATA CHUNKING ALGORITHMS

According to the condition of finding boundaries, algorithms can be classified into fixed-size chunking and content-defined chunking (CDC).

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

Fixed-size chunking, as its name implies, is to chunk data based on a fixed length. The sizes of chunks in the results of the algorithm are the same. This kind of chunking is conditional on the subscripts of the read bytes when searching for the boundary. When the subscript is equal to an integer multiple of a preset value, it is set as a boundary here. Fixed-size chunking algorithm is simple, easy to understand and fast. However, because the boundaries are set according to the subscripts of the read bytes, there is a problem of byte shifting. For example, when a byte is inserted at the beginning of an data stream, the chunks will all be changed, which will reduce the efficiency of chunk-based application.

CDC algorithm, also known as variable-size chunking, is based on the content of the read bytes to determine whether it acts as a boundary. In this kind of chunking, the data is read as a byte stream, and a data window is set up. For each read byte, the data window moves one byte forward as well. We can decide to set the boundary here when the data in the window satisfies certain conditions. CDC needs a more complex calculation because it is necessary to calculate the data in the window to determine whether the preset



FIGURE 1. The algorithm for random bytes insertion.

chunking conditions are satisfied. Since this kind of algorithm is content-based, when byte shifting occurs, the data in the window, which meets the preset conditions, will still satisfy the ones and be set as boundaries. In CDC, byte shifting will affect nothing but the current data chunk.

The difference between fixed-size chunking and CDC can be seen roughly in Fig. 1. In fixed-size chunking algorithms, a cut-point is set based on its index, which is equal to or as many times as a fixed integer. On the contrary, in CDC, a cut-point is set based on its content instead of index.

B. INCREMENTAL SYNCHRONIZATION

In data backup system and distributed service system, data synchronization technology is used to keep the data in two places consistent [16]. According to synchronization methods, it can be classified into full synchronization and incremental synchronization. Full synchronization is to replace the whole original data with the target data, so as to achieve the purpose of synchronization. Incremental synchronization, on the other hand, only applies the changed part of the data to the original data, which saves the cost of data transmission and others.

Incremental synchronization includes the following steps: firstly, compare the target data and the original data to find the changed parts of the data; secondly, send these parts to the original data server, and finally process the changed parts at the original data to achieve data consistency. In first step, in order to make the next two steps more smoothly and cheaply, it is particularly important to find the changed parts of the data.

There are many traditional ways to find the changed parts. Trigger mode is creating a trigger for every table in the database [17]. When a table changes, it can be captured by the trigger, and then increments can be found. However, this mode will affect the performance of the database and easily shut down the whole system because of some minor omissions. Programming mode is to record the operation of the database in the code [18]. Every insert, delete, update and new operation are recorded. These operations are equal to the increment of data.

Rsync mode [19], which is used in Linux system, is based on a algorithm with the same name. Rsync can realize the data synchronization between files, but it is very complex and time-consuming during synchronization. Rsync algorithm is a fixed-size chunking algorithm, although it avoids the problem of byte shifting problem, but in exchange for a large amount of computation. Rsync algorithm contains three steps. First, the target file, which is to be synchronized, is divided into fixed length chunks on the PC it belongs, and a checksum with strong and weak check values of all chunks is sent to the server. Second, the original file on the server is chunked by a sliding window and two-stage check value comparison is implemented with the checksum to get the diff-chunks and send them back to target PC. Third, the target PC will generate a new file as same as the original file based on the diff-chunks. Because sliding check is implemented at the second step, Rsync uses weak check to reduce the times of strong check. However, in the case of huge changes in the original file, weak check still consumes lots of computation. In the experimental phase, we will compare the performance of PCI and Rsync based on real data.

This paper searches the changed parts between two data by chunking the target data and the original data into chunks based on the data chunking algorithm, finding the different chunks between the original and target data, and regarding these chunks as changed data.

Our contributions are as follows.

- 1) In Section 2, we give a detail description of CDC algorithms. Point out their ideas and shortcomings. Update the requirements of CDC when used in data incremental synchronization.
- 2) In Section 3, we develop a novel CDC algorithm PCI. Describe its process and discuss its performance on the requirements of CDC mentioned in Section 2. We show analytically that PCI is more suitable in data incremental synchronization.
- 3) In Section 4, we discuss the time and space complexity of all the algorithms mention in our paper.
- 4) In Section 5, we experimentally compare PCI with five of state of art CDC algorithms for test items including chunking speed, chunk size distribution, and incremental data discovery.

II. BACKGROUND AND MOTIVATION

This section discusses the background of chunking algorithms, their limitations, and motivations of our work.

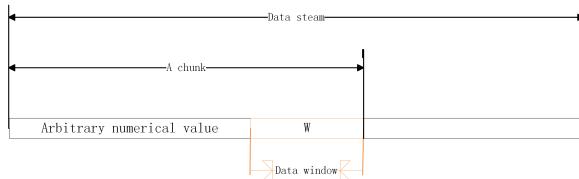
A. BACKGROUND

To solve the problem of byte shifting in fixed-length algorithm, a content-defined variable-length chunking algorithm [20] is proposed, which reads files as a data stream and generates chunks according to the Rabin fingerprint of a window data. To solve the issue that it is difficult to find the cut-off point in Rabin algorithm, it is proposed to adopt two divisors instead of one [21], one is easy to achieve and

Algorithm 5: Algorithm for Rabin chunking

Input: input file,file; default value,Value;length of sliding window,W;
Output: cut point,I;

```
function RabinChunking(file,Value,W)
    i=1
    index=0
    while(byte=readByte(file))
        array[index%W+1]=byte
        if array.length>=W then
            if hashValue(array,index,W)==Value then
                return i
            end if
        else
            continue
        end if
        i=i+1
    end while
end function
```

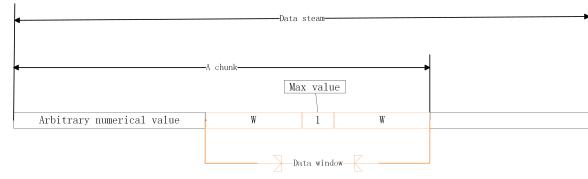
FIGURE 2. The pseudo code of the Rabin algorithm.**FIGURE 3.** The algorithm process of Rabin.

the other is opposite. At the beginning of finding a cut-off point, the difficult divisor should be adopted. If the data is not satisfied in a long data interval, it will be replaced by the easy one, so that large chunks can be avoided. In addition, Rabin fingerprint have a problem called size variance of chunks. To dual with this situation, LMC(Local Maximum Chunking) algorithm is proposed [22]. Instead of calculating Rabin fingerprint, the algorithm decides to set a cut-off point when the maximum value of a window data is in the middle of the window, which saves the time of calculating Rabin fingerprint. At the same time, because the window size can be set, the size of the chunks can be limited, and the distribution of the chunk size is relatively stable. In order to speed up the validation of the window data, AE [23] and RAM [24] algorithms are proposed. By changing the validation method, which will be described in detail later, of window data, the speed of chunking is accelerated. In addition, to achieve a faster data chunking, the idea of parallel computing is applied to data chunking algorithms. Won et al. developed a multi-thread variable size chunking method, which exploits the multi-core architecture of the modern microprocessors [25]. Another multi-thread content based file chunking system is given by Tang et al. in CPU-GPUPU heterogeneous architecture [26]. To improve the parallel performance, a two-stage parallel CDC is proposed by Ni et al. to spit up the parallel chunking process into two stages [27]. Beside, a multi-chunk deduplication scheme is proposed by Niesen et al. to provide an information-theoretic analysis of data deduplication for a designed source model [28] and To address the file reliability

Algorithm 4: Algorithm for LMC chunking

Input: input file,file; size of fixed window,W;
Output: cut point,I;

```
function LMCChunking(file,W)
    i=1
    start=1
    while(byte=readByte(file))
        if byte<=max.value then
            if i==max.position+w and max.position>=start+w then
                start=max.position+1
                return max.position
            end if
        else
            max.value=byte
            max.position=i
        end if
        i=i+1
    end while
end function
```

FIGURE 4. The pseudo code of the LMC algorithm.**FIGURE 5.** The algorithm process of LMC.

issue, Wu et al. proposes a Per-File Parity (short for PFP) scheme to improve the reliability of deduplication-based storage systems [29].

Next, the process and pseudo-code of Rabin algorithm, LMC, AE and RAM algorithms are introduced separately.

1) RABIN CHUNKING ALGORITHM

During reading the data stream, a data window is set to calculate the Rabin fingerprint of the data in the window. If the result of calculation is equal to the preset value, the cut-off point is set at the end of the window. Otherwise, move the data window one byte forwards and repeat the above process until all the cut-off points and chunks in the data stream are found. The pseudo code and chunking process of Rabin algorithm are shown in Fig. 2 and Fig. 3.

2) LMC CHUNKING ALGORITHM

In the process of reading data stream, a data window is set. If the maximum value of bytes in the data window is right in the middle of the window, the cut-off point is set at the end of the window. Otherwise, move the data window one byte forwards and repeat the above process until all the cut-off points and chunks in the data stream are found [22]. The pseudo code and chunking process of LMC algorithm are shown in Fig. 4 and Fig. 5.

3) AE CHUNKING ALGORITHM

During reading data stream, a data window is set. If the maximum value of bytes in the data window is located at the

Algorithm 3: Algorithm for AE chunking
Input: input file,file; size of fixed window,W;
Output: cut point,I;

```
function AEChunking(file,W)
    i=1
    while(byte=readByte(file))
        if byte<=max.value then
            if i==max.position+w then
                return i
            end if
        else
            max.value=byte
            max.position=i
        end if
        i=i+1
    end while
end function
```

FIGURE 6. The pseudo code of the AE algorithm.

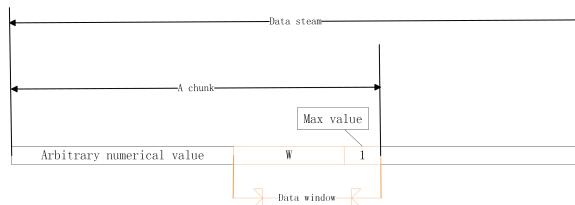


FIGURE 7. The algorithm process of AE.

end of the window, the cut-off point is set at the end of the window. Otherwise, move the data window one byte forwards and repeat the above process until all the cut-off points and chunks in the data stream are found [23]. The pseudo code and chunking process of AE algorithm are shown in Fig. 6 and Fig. 7.

4) RAM CHUNKING ALGORITHM

In the process of reading data stream, a data window is set at the starting point or after the last cut-off point. If a byte value with no less than all byte values in the window is read out of the window, a cut-off point is set at this byte. Otherwise, continue to read the byte values and repeat the above process until all the cut-off points and chunks in the data stream are found [24]. The pseudo code and chunking process of RAM algorithm are shown in Fig. 8 and Fig. 9.

B. MOTIVATION

CDC algorithm is mostly used in the field of duplicate data deletion. To achieve a better deletion, there are several requirements for CDC algorithm proposed by Zhang et al. in [23].

- Content dependence. The condition of chunking must be based on data content. Only in this way can we resist byte shifting and find more duplicate data between similar files [30].

Algorithm 2: Algorithm for RAM chunking
Input: input file,file; size of fixed window,W;
Output: cut point,I;

```
function RAMChunking(file,W)
    i=1
    while(byte=readByte(file))
        if byte>=max.value then
            if i>w then
                return i
            end if
            max.value=byte
            max.position=i
        end if
        i=i+1
    end while
end function
```

FIGURE 8. The pseudo code of the RAM algorithm.

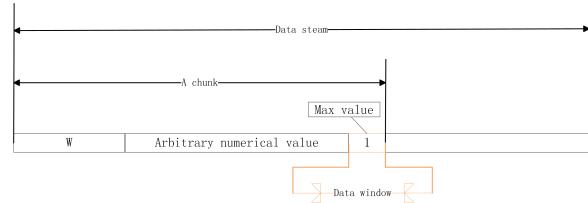


FIGURE 9. The algorithm process of RAM.

- Low chunk size variance. Since the output chunks of the chunking algorithm are supposed to be stored in disk during duplicate data deletion, the high size variance will affect the efficiency of storage and the performance of de-duplication [31].
- Ability to eliminate low entropy strings. Low entropy strings are strings which consist of repetitive bytes or patterns. This challenge means it is preferable for the algorithm to be able to eliminate the redundancy within this kind of string [32].
- High throughput [33]. It means fast running speed, which must be considered in any algorithm.

However, when CDC algorithm is applied to incremental synchronization, even the same algorithm will have different focus due to the change of purpose, and these requirements need to be revised.

- Content dependence. This does not need to be changed, because in incremental synchronization, finding data increments must also be resistant against the byte shifting, because if byte insertion causes all subsequent chunks to change, then a file will look like a new file, even if only a small part of it has been changed.
- Low chunk size variance. This should be changed, because in the field of duplicate data deletion, the stability of data length is guaranteed in order not to waste disk space when storing chunks. However, in

Algorithm 1: Algorithm for MII chunking

Input: input file,file; length of incremental interval,W;
Output: cut point,I;

```

function MIIChunking(file,w)
    i=1
    increment=0
    while(byte=readByte(file))
        if byte>previous.value then
            increment=increment+1
            if increment==W then
                return i
            end if
        else
            increment=0
        end if
        previous.value=byte
        i=i+1
    end while
end function
```

FIGURE 10. The pseudo code of the MII algorithm.

incremental synchronization algorithm, the goal of data chunking is to find different parts of data, not to store them. Therefore, in incremental synchronization systems, chunk size instability can be tolerated.

- 3) Ability to eliminate low entropy strings. This is still important to ensure that a chunk remains one when it has not changed.
- 4) High throughput. This is still important, because the real-time requirements of current applications require all processes should as faster as they can with the same remaining conditions.
- 5) Performance. When we use a CDC algorithm, we always have a purpose. This challenge is to make sure we can accomplish the target of our purpose as well as possible. An algorithm is used to accomplish a task, and the quality of completion is the most important index to judge whether the algorithm is good or bad.

The reasons why we ignore the importance of “the performance of de-duplication” are: (1) the previous papers (AE, RAM, etc) use CDC to reduce hard disk usage and the resulting chunks are stored on disk directly and perhaps permanently, thus the resulting chunks should be equal to or as many times as disk sector to reduce fragmentation on the hard disk. (2) We use CDC only to find different parts between two similar files. During incremental synchronization, it is necessary to find out the difference between two files stored in different places by means of network communication. The generated chunks will only be stored temporarily, and will be deleted after finding the difference data between files.

In the previous research of our team, MII algorithm was proposed to achieve better ability of resistance against the

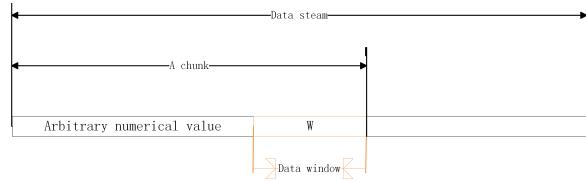


FIGURE 11. The algorithm process of MII.

Algorithm 10: Algorithm for PCI chunking

Input: input file,file; size of fixed window,W; Preset threshold, tsValue;

Output: cut point,I;

```

function PCIChunking(file,W)
    i=1
    index=0
    array[]={-1}
    while(byte=readByte(file))
        array[index%W+1]=byte
        if Content(array)>=W then
            if ParityCheck(array,index,W)>=tsValue then
                return i
            else
                continue
            end if
        end if
        i=i+1
    end while
end function
```

FIGURE 12. The pseudo code of the PCI algorithm.

byte shifting by sacrificing the stability of chunk size [32]. The pseudo code and chunking process of MII algorithm are shown in Fig. 10 and Fig. 11.

However, the effect of the algorithm is not good enough. Our previous work is difficult to adjust the average chunk size. When we need to compress the transmission cost as much as possible, we need to reduce the data contains all the difference between two files, which is difficult for MII to do. To improve the effect of finding incremental data, we propose a more flexible CDC algorithm. By using a more flexible cut-point search method, we can better locate the changed part of the data and reduce the amount of data needed to be synchronized. The experiments show the number of incremental data found by our new algorithm can be reduced by 32%~57%.

III. PARITY CHECK OF INTERVAL

We propose a novel algorithm called Parity Check of Interval(PCI) to locate changed data more accurately by adopting a flexible cut-point search strategy.

A. ALGORITHM PROCESS

Suppose there is a file that needs to be chunked. First step, read the file as a data stream and set up a data window with length of W . The starting point of the data window is the first byte of the file. Second step, read one byte at a time until the file has finished reading. Third step, make the following

Algorithm 10_1: Algorithm for Content function

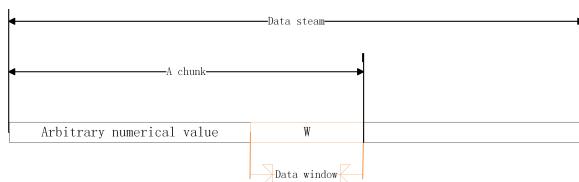
Input: input array,array;
Output: number,num;

```
function Content(array)
    i=0
    w=array.length
    num=0
    while(i<w)
        if array[i] != -1 then
            num=num+1
        end if
        i=i+1
    end while
    return num
end function
```

FIGURE 13. The pseudo code of the content.**Algorithm 10_2: Algorithm for ParityCheck function**

Input: input array,array; start index, index; length of window,W;
Output: number,num;

```
function ParityCheck(array,index,W)
    i=index
    num=0
    parityOfByte[]={0,1,1,⋯,8} // size:256
    while(i!=index-1)
        num=num+parityOfByte[array[i]]
        i=(i+1)%W
    end while
    return num
end function
```

FIGURE 14. The pseudo code of the Parity Check.**FIGURE 15.** The algorithm process of PCI.

decision: if the data window is filled with bytes and the number of 1s in binary form(NO1BF) of all bytes is not less than the preset value, set cut-off point at the end of the data window, move the starting point of the data window to the next byte of the cut-off point and return to the second step; If the data window is not filled with bytes, return to the second step; If the data window is filled with bytes but NO1BF of all bytes is less than the preset value, move the data window forward by one byte and return to the second step. The pseudo code and chunking process of PCI algorithm are shown in Fig. 12, Fig. 13, Fig. 14 and Fig. 15.

B. HOW TO PRESET THE THRESHOLD

In the PCI algorithm, there are two values to be preset, the threshold of total NO1BF in data window and the length of data window. A byte value is between 0 and 255. NO1BF of a byte value is between 0 and 8. If a byte is random, then its

NO1BF obeys the discrete uniform distribution with values ranging from 0 to 8. If the sum of N bytes is limited, only some N bytes can be guaranteed to meet the conditions. Based on this, PCI algorithm calculates the NO1BF in a fixed data window, and then compares this value with the preset value to determine whether to set a cut-off point. Assuming that the size of the data window is W and the threshold value of total NO1BF is V , the probability of a random data window meeting the condition can be calculated by a given formula [34]. Since this paper do not discuss this mathematical problem, here just set the probability to p as we can see in the follow formula.

$$f_n(V) = P \{X_1 + X_2 + \dots + X_w \geq V\} = p$$

In this formula, X_i , $\{i = 1, 2, \dots, w\}$ stands for NO1BF.

By setting the values of W and V , we can adjust the difficulty of finding the cut-off point, and then roughly control the size of the chunks. The larger W , the smaller V , the easier it is to find the cut-off point, and vice versa.

In this paper, W is set to 5 and V is set to 34. The purpose of this setting is to ensure that there is approximately the same number of chunks compared with other algorithms in the experiment section, which means that the bandwidth cost of the first data transmission in the synchronization are almost same among algorithms. In practical application, two preset values can be adjusted freely according to the actual situation and demand, which shows the algorithm is much flexible.

C. CHUNKING SPEED

In terms of chunking speed, the algorithm only needs to traverse the file once for a file, and only needs assignment, addition and value-taking during searching the cut-off point with less computation. When calculating the same file, the algorithm will not take the disadvantage of chunking speed from other algorithms. Specific comparisons of running time will be made in the section of time complexity, and the comparisons of the chunking speed of the same file will be given in the following experimental section.

D. CHUNK SIZE VARIANCE

In introducing the principle of the algorithm, it is pointed out that for a random data window, it can calculate a certain value of the probability when it satisfies the cut-off points condition. So finding cut-off points in the algorithm can be regarded as a Bernoulli experiment. Then, starting from the previous cut point, the distance N from reading to the next cut-off point roughly follows a geometric distribution. The distance N is actually the size of the chunks. Consequently, the size of the chunks is not stable enough. The smaller the length of the chunks, the more the ones, and vice versa. However, when the algorithm is used to discover incremental data, the diversity of chunk size will not cause too much impact, because the chunks are not for storage, but to find different parts. In practical applications, it is necessary to merge the different chunks found together and send them to

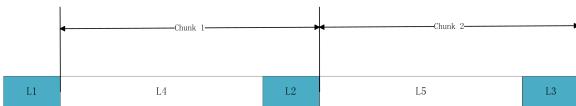


FIGURE 16. Resistance to byte shifting (Step1).

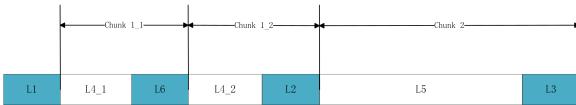


FIGURE 17. Resistance to byte shifting (Step2).

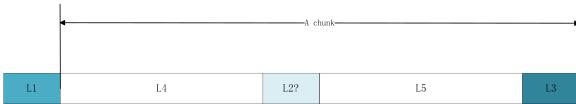


FIGURE 18. Resistance to byte shifting (Step3).

the server that needs data synchronization. Therefore, chunk size does not have much importance.

E. BYTE SHIFTING RESISTANCE

The main purpose of this algorithm is to find the same parts of two similar files, so the ability to resist byte shifting is the most important index. To ensure that the same content in two files falls into the same chunk as much as possible, it is necessary to ensure that the same data has the same boundary. This also means that the same boundaries need to withstand the impact of changes in nearby data. Byte shifting resistance refers to whether changes in bytes nearby will affect the original chunk. The ability of the algorithm to resist byte shifting is demonstrated by deduction as follow.

Assuming that there is a piece of data as shown in Fig. 16, where L_1 , L_2 and L_3 are three boundaries satisfying the presupposition of the algorithm, $Chunk1(L_4 + L_2)$ and $Chunk2(L_5 + L_3)$ are two chunks. When byte shifting occurs in the middle of two boundaries, assume that it occurs in L_4 . If no new boundaries are generated because of the byte shifting, only $Chunk1$ changes. If byte shifting results in new boundaries, as shown in Fig. 17, $Chunk1$ becomes two chunks, $Chunk_{1,1}$ and $Chunk_{1,2}$. However, $Chunk2$ and chunks before L_1 are unaffected. If byte shifting occurs right on a boundary, assuming that it occurs on L_2 , as shown in Fig. 18, we regard L_4 , L_2 and L_5 as a large chunk and do not care whether L_2 will be changed into a data window that does not meet the preset conditions, then the chunks before L_1 and after L_3 will not be affected. In summary, when byte shifting occurs, only the chunk in which it is and the one after it are affected, and the rest are not affected.

F. ABILITY TO ELIMINATE LOW ENTROPY STRINGS

For low-entropy strings, the principle of the algorithm is that the NO1BF in a data window is greater than or equal to the preset value. So even if a string is roughly the same

TABLE 1. Time complexity contrast among algorithms.

Algorithm	Time complexity	each loop(operation)
Rabin	$O(n)$	$1ADD + 2M + 2ASS + 1C + 1H$
LMC	$O(n)$	$4ADD + 2ASS + 3C$
AE	$O(n)$	$2ADD + 1ASS + 2C$
RAM	$O(n)$	$1ADD + 3ASS + 2C$
MII	$O(n)$	$2ADD + 3ASS + 2C$
PCI	$O(n)$	$1ADD + 2ASS + 2C + 1FUNC$

value, the original data window that meets the preset conditions will still be the data window. If the low-entropy string does not change, it will be included in a chunk by the original data window. When looking for duplicate data, it will be considered as the same data without affecting subsequent chunks. For example, assume that there is a string $10001101'cut-point1'000000000000'cut-point2'1011100$ in the source file. In the destination file, the string ' $cut-point1'000000000000'cut-point2'$ will not be in the different parts unless this string is changed in the destination file.

IV. TIME AND SPACE COMPLEXITY

In this section, we discuss the time and space complexity of five algorithms which are the newest or state of art and our algorithm. The algorithms covered in this section include Rabin, LMC, AE, RAM MII, and PCI.

A. TIME COMPLEXITY

The processes and pseudo-codes of these algorithms have been introduced in the previous section. It can be seen from the pseudo-codes that only one loop is contained in these algorithms, that is, the time complexity of them is $O(n)$. But the number of operations of each algorithm is different in a loop, so the actual operands of each algorithm are given here, as shown in Table 1. ADD means addition, M means modular reduction, ASS means assignment, C means comparison, H means hash operation and $FUNC$ means function.

Although these algorithms mainly use comparisons to find a cut-point, there are some other operations during the chunking process and these operations cost non-ignorable time. For example, LMC needs an array to store the information of the bytes in the two windows and updates it in every loop. We only counted the total operations in the pseudocodes of these algorithms whether or not they will actually be implemented in the "each loop(operation)" part. This is unsuitable and the true operations of each algorithm can be found in its pseudocode. And the running time can be optimized by programming.

B. SPACE COMPLEXITY

In Rabin algorithm shown in Fig. 2, it is necessary to record all the byte values of the sliding window because they will be used next time when the current matching of Rabin fingerprint fails. Therefore, the space complexity of the algorithm is $O(1)$, specifically an array of length W (W is the length of the sliding window).

TABLE 2. Space complexity contrast among algorithms.

Algorithm	Time complexity	exact space used
Rabin	$O(1)$	$Array[W]$, W is the length of the sliding window
LMC	$O(1)$	$Array[2W+1]$, W is the length of the fixed window
AE	$O(1)$	one integer variable
RAM	$O(1)$	one integer variable
MII	$O(1)$	two integer variables
PCI	$O(1)$	$Array[W+1]$, W is the length of the data window

In LMC algorithm shown in Fig. 4, it uses a circular queue to store the data in the sliding window. All the bytes in it are used to exchange space for time. Therefore, the space complexity of the algorithm is $O(1)$, specifically an array of length $2W + 1$ (W is the length of the fixed window).

In AE algorithm shown in Fig. 6, it just need to remember the maximum byte value in the sliding window, and then compare the read byte to this maximum byte. Therefore, the space complexity of the algorithm is $O(1)$, specifically one integer variable.

In RAM algorithm, the space complexity is $O(1)$, specifically one integer variable, because only the maximum value is needed during current and next comparisons as we can tell from Fig. 8.

In MII algorithm shown in Fig. 10, there is an integer variable to store the current byte value during comparison with the next byte value. Besides, another integer variable is needed to store the length of current incremental interval. Therefore, the space complexity of the algorithm is $O(1)$, specifically two integer variables.

In PCI algorithm shown in Fig. 12, when NO1BF of the data window misses the target, we need to slide one byte forward to calculate NO1BF of the new window, which means it is necessary to record all the byte values of the data window. Therefore, the space complexity of the algorithm is $O(1)$, specifically an array of length $W + 1$ (W is the length of the data window, 1 is used to realize a circular queue).

The Contrast between space complexities of algorithms is shown in Table 2.

V. EXPERIMENTS OF CHUNKING ALGORITHMS

This section discusses the experiments of the article.

A. OBJECTIVE

Compare the items including running time, chunk size distribution and incremental data discovery among the Rabin, LMC, AE, RAM, MII and PCI algorithms.

B. DATASETS

There are two problems in using the real documents when choosing the experimental data. One is that individual algorithms may have special effects on some data, which will lead to inconsistent conclusions in the comparison of algorithms; the other is that because of the variety and quantity of actual

TABLE 3. The datasets used for experiments of chunking algorithms.

name	size(bytes)	generating algorithm		
data1	$2 * 10^9$	Mersenne Twister	Pseudo-Random	
data2	$2 * 10^9$	Mersenne Twister	Pseudo-Random	
data3	$2 * 10^9$	Mersenne Twister	Pseudo-Random	
data4	$1.5 * 10^9$	Mersenne Twister	Pseudo-Random	
data5	$1.5 * 10^9$	Mersenne Twister	Pseudo-Random	
data6	$1.5 * 10^9$	Mersenne Twister	Pseudo-Random	
data7	$1 * 10^9$	Mersenne Twister	Pseudo-Random	
data8	$1 * 10^9$	Mersenne Twister	Pseudo-Random	
data9	$1 * 10^9$	Mersenne Twister	Pseudo-Random	

documents, only a few ones can be selected in the experiment section of the article, which can not represent all the actual data. It is inappropriate to judge which algorithm is better only based on a few actual documents.

In this paper, when choosing the datasets, we adopt the method of random file generation, which can maximize the non-particularity of the file to ensure that the experimental data are random and common. We choose the Mersenne Twister Pseudo-Random Number Generator [35], which has quite good pseudo-random characteristics, to generate datasets. The experimental data consist of nine files, including three files about $2G$ in size, three files about $1.5G$ in size and three files about $1G$ in size, as shown in Table 3. The reason why the number of experimental data is nine is that the data are all randomly generated and there will be no kind of sampling. Although more experimental data will be more convincing, nine are enough.

C. CHUNKING SPEED

In this section, the chunking speed is compared among these six algorithms through the experimental results. As to an experimental file, the running time of each algorithm, i. e. chunking time, can be obtained by implementing the six algorithms separately. In the case of the same file, the shorter the chunking time, the faster the chunking speed will be. The experimental results of nine experimental files are shown in Fig. 19.

The experimental results show that PCI algorithm is not the fastest one among these algorithms, but in an order of magnitude, which is acceptable in the processing of massive data. Among these algorithms, RAM algorithm has the fastest chunking speed, followed by AE, MII, RABIN and LMC from fast to slow.

D. CHUNK SIZE DISTRIBUTION

In this section, the size distribution of chunking results among six algorithms is discussed with the help of experimental

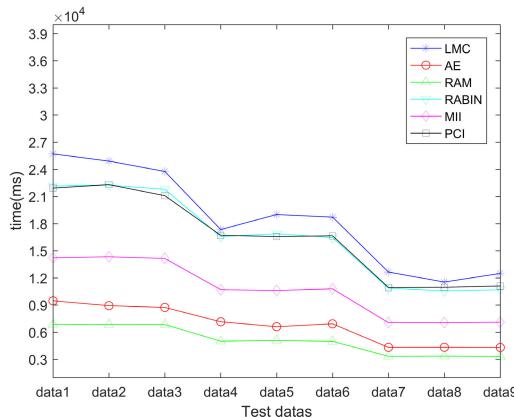


FIGURE 19. The running time of chunking algorithms in different datasets.

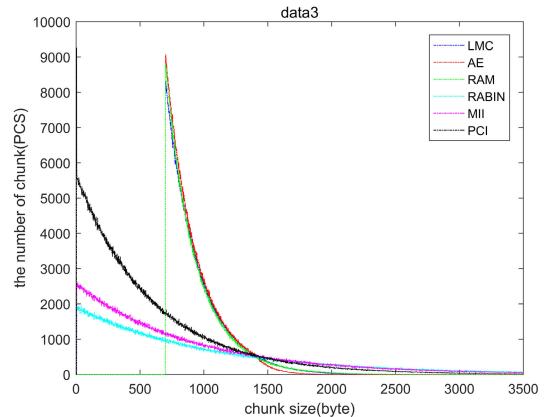


FIGURE 22. The chunk size distributions of chunking algorithms in data3.

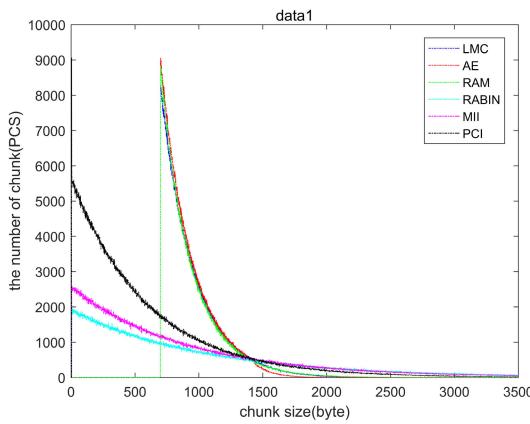


FIGURE 20. The chunk size distributions of chunking algorithms in data1.

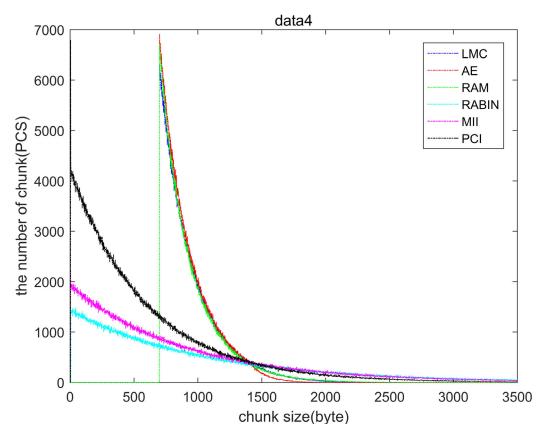


FIGURE 23. The chunk size distributions of chunking algorithms in data4.

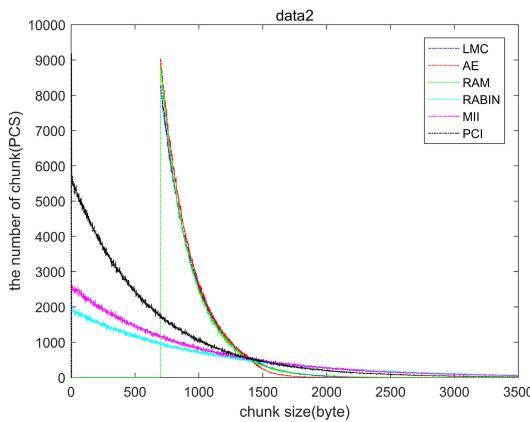


FIGURE 21. The chunk size distributions of chunking algorithms in data2.

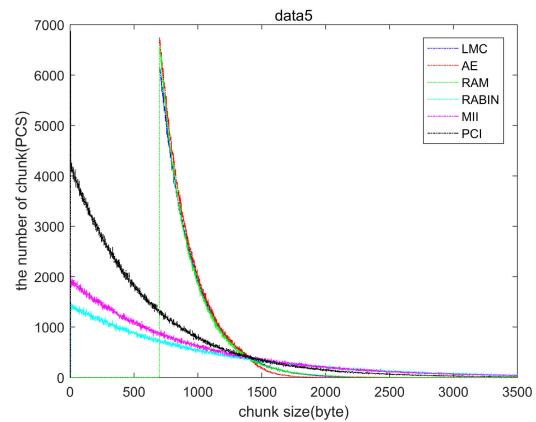
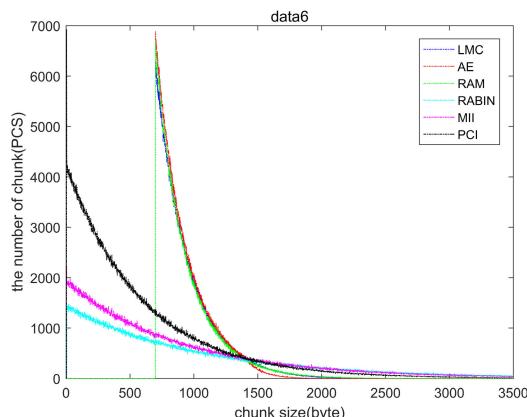
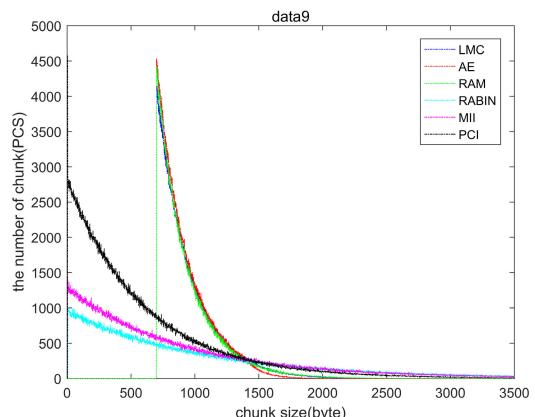
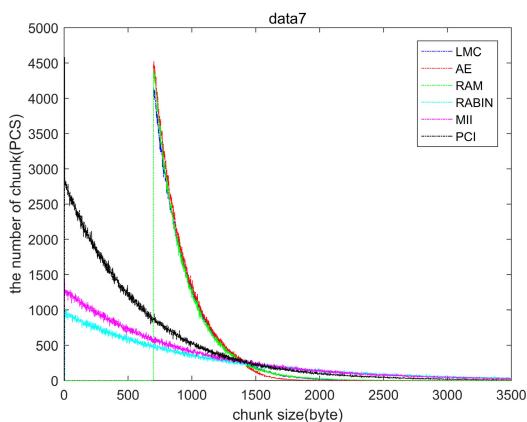
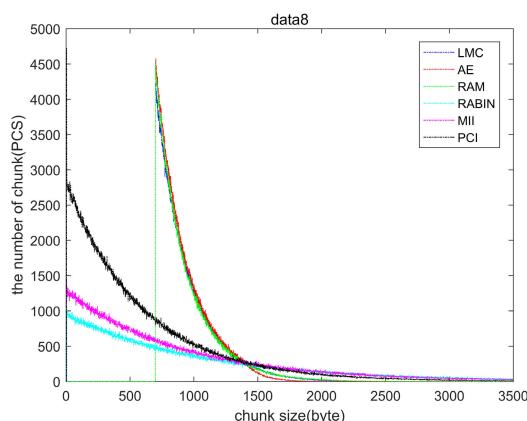


FIGURE 24. The chunk size distributions of chunking algorithms in data5.

results. As to an experimental file, six chunking algorithms are used to process it separately, and the results, including all the chunks, of each chunking algorithm can be obtained. By accumulating all the chunks with the same length, size distribution of the chunks can be obtained. The experimental results of nine experimental files are shown in Fig. 20, Fig. 21, Fig. 22, Fig. 23, Fig. 24, Fig. 25, Fig. 26, Fig. 27 and Fig. 28.

As can be seen in these figures, the chunk size of LMC, AE and RAM algorithms has a minimum value and the fluctuation of size is small, which is related to the sliding window set up. The existence of sliding window makes the size of chunks not less than the length of sliding window. In the experiment of this section, the sliding window is set to 700, the purpose of which is also to make the total number of chunks

**FIGURE 25.** The chunk size distributions of chunking algorithms in data6.**FIGURE 28.** The chunk size distributions of chunking algorithms in data9.**FIGURE 26.** The chunk size distributions of chunking algorithms in data7.**FIGURE 27.** The chunk size distributions of chunking algorithms in data8.

among these six algorithms is roughly the same. If the total number of chunks of all the algorithms are designed almost the same during the experiments. Since we use MD5 as the fingerprint, the data transferred(fingerprints) are almost the same for all the algorithms. The chunk size distributions of RABIN, MII and PCI are basically in conformity with geometric distributions, and the size fluctuations are large,

which have been discussed in the previous section. However, to find incremental data in data synchronization, chunks are not used for storage. The process of data synchronization is as follow: fingerprints of the chunks in source file are transferred to destination file location, different fingerprints between source file and destination file are transferred back to source file location after comparison, and then the original chunks, aka different parts between two files, of the different fingerprints are transferred to destination file location, finally, a same version of source file is composed in destination file location. All the chunks generated during this process are deleted after the same version of source file is composed. So the diversity of chunk size will not affect the ultimate purpose of chunking, which is positioning different parts between two files and make these two the same.

E. INCREMENTAL DATA DISCOVERY

In this section, we discuss how much incremental data can be found in the chunking results of each algorithm with the help of experimental results. The steps are as follows. As to an experimental file *data1*, firstly, new files *data1_{add}*, *data1_{delete}* and *data1_{insert}* are obtained by means of addition, random deletion and random insertion algorithms. The addition algorithm is shown in Fig. 29, the random deletion algorithm is shown in Fig. 30, and the random insertion algorithm is shown in Fig. 31. Secondly, one chunking algorithm is used to process *data1* and *data1_{add}* respectively to find out the incremental data *data1_{add;nc}* in the case of addition. Then the incremental data *data1_{delete;nc}*, in the case of random deletion, and *data1_{insert;nc}*, in the case of random insertion, are obtained by the same method. Then repeat the above steps with the other five chunking algorithms, and the incremental data found by six algorithms in three cases, which are addition, random deletion and random insertion, can be obtained. Finally, by repeating the above steps for the remaining eight experimental files, we can get the incremental data found by each algorithm in three cases under different experimental files. The results in three cases are shown in Fig. 34, Fig. 33 and Fig. 32.

Algorithm 9: Algorithm for Adding**Input:** Input file,fileIn;**Output:** Output file,fileOut;**function** Add(fileIn)

i=0

j=0

while(byte=readByte(fileIn))

fileOut.write(byte)

end while **for** j 0 to 20000 by 1 **do**

fileOut.write(Random(0,255))

end for**end function****FIGURE 29.** The algorithm for random bytes addition.**Algorithm 8:** Algorithm for Deleting**Input:** Input file,fileIn;**Output:** Output file,fileOut;**function** Delete(fileIn)

i=0

j=0

while(byte=readByte(fileIn))

i=i+1

if j==0 **then**

fileOut.write(byte)

else

j=j-1

end if **if** i%10000==0 **then**

j=100

end if **end while****end function****FIGURE 30.** The algorithm for random bytes deletion.

As can be seen from the figures, in the case of addition, the incremental data found by the six algorithms are roughly the same, because the addition is made at the end of the file, so the original part remains unchanged, and the data found by all the algorithms are basically the last additional data. In the case of random deletion and insertion, the incremental data found by PCI is significantly lower than the other five algorithms, because in these two cases, the changes of new data will change the original data, which may affect the data chunk. If the resistance against byte shifting of the algorithm is not strong enough, the original chunk boundary is easy to

Algorithm 7: Algorithm for Inserting**Input:** Input file,fileIn;**Output:** Output file,fileOut;**function** Insert(fileIn)

i=0

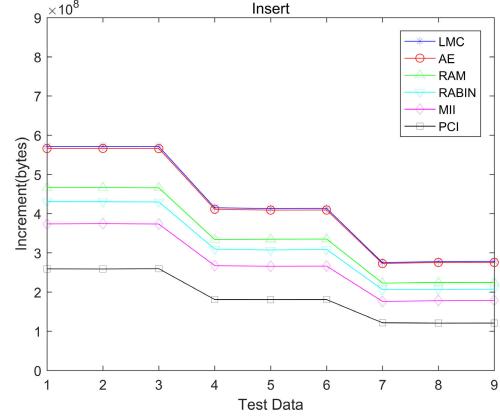
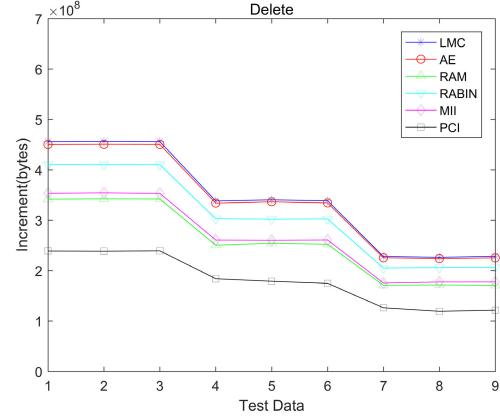
while(byte=readByte(fileIn))

i=i+1

fileOut.write(byte)

if i%10000==0 **then** **for** j 0 to 100 by 1 **do**

fileOut.write(Random(0,255))

end for **end if** **end while****end function****FIGURE 31.** The algorithm for random bytes insertion.**FIGURE 32.** The incremental data discovery of the chunking algorithms in random bytes insertion case.**FIGURE 33.** The incremental data discovery of the chunking algorithms in random bytes deletion case.

be changed because of the data, which leads to the change of a whole chunk. The most important feature of PCI algorithm

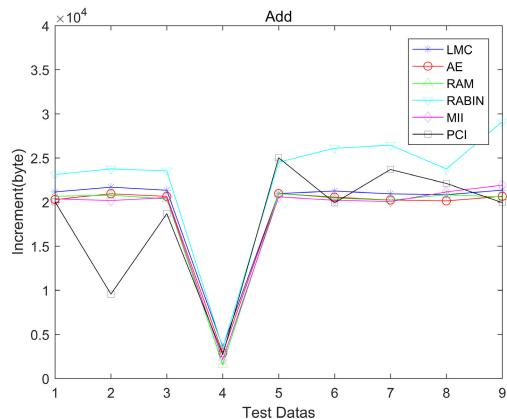


FIGURE 34. The incremental data discovery of the chunking algorithms in random bytes addition case.

is its strong ability of resistance against byte shifting, which makes PCI algorithm keep chunks from changing when the data nearby changes, so it can better identify the duplicate data when looking for incremental data, thus reducing the incremental data found. As can be seen from the figures, PCI algorithm can delineate the location of the incremental data in a smaller range and less is better, which is reduced by about 32%~57% compared with other algorithms. In incremental synchronization system, most of the changes are random deletion and insertion, thus PCI algorithm can locate the changed data more accurately in incremental synchronization system and reduce the bandwidth cost of data transmission.

VI. EXPERIMENTS BETWEEN PCI AND CLASSIC RSYNC

A. OBJECTIVE

Compare the items including proceeding time, actual amount of data transferred and ideal amount of data transferred between the Rsync and PCI algorithms.

B. DATASETS

Our datasets adopt the operating system snapshots mentioned in the literature [23], [27], [36], which can be downloaded to the website <http://tracer.filesystems.org/>. These snapshots were collected on a Mac OS X Snow Leopard server running in an academic computer lab. The server runs the following services:

- 1) LDAP: OpenDirectory (user/group management).
- 2) SMTP: Postfix.
- 3) MySQL for Bugzilla.
- 4) HTTP: Apache.
- 5) FTP.
- 6) Calendar server (CalDAV).
- 7) Wiki server.
- 8) Contacts server (CardDAV).

There are over 250 users in the system, many are current and ex-students, some guests, and collaborators. At any given time, between 20-30 users are actually active. We modify the snapshots according to the way mentioned in the

TABLE 4. The datasets used for experiments between PCI and classic Rsync.

Name	Size(Gb)	Files(pieces)	Source
data1	13.70	7	http://tracer.filesystems.org/
data2	13.11	7	http://tracer.filesystems.org/
data3	12.82	7	http://tracer.filesystems.org/
data4	11.55	7	http://tracer.filesystems.org/

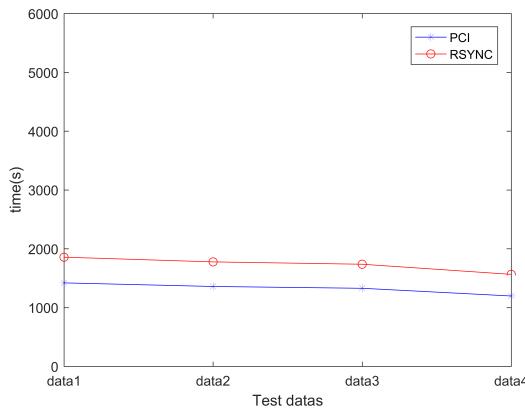
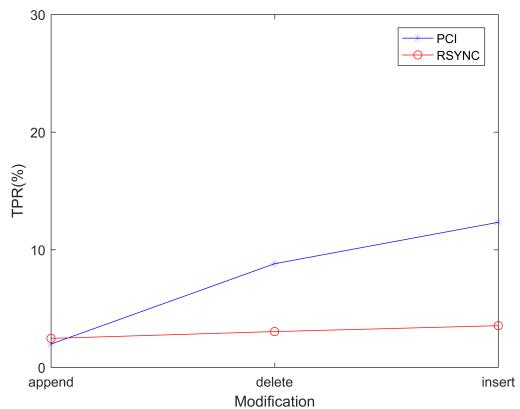
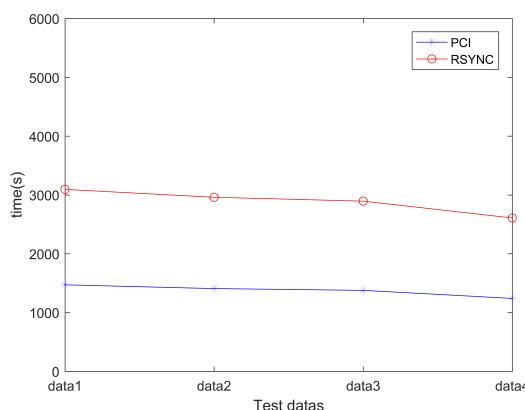
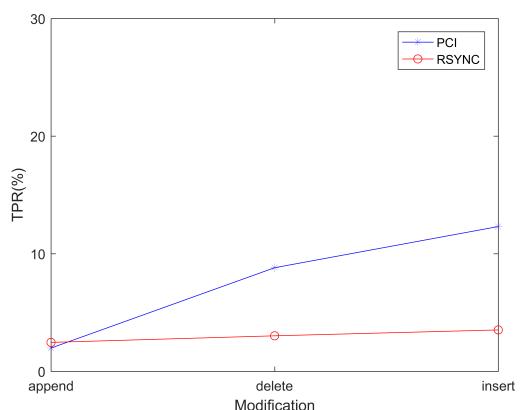
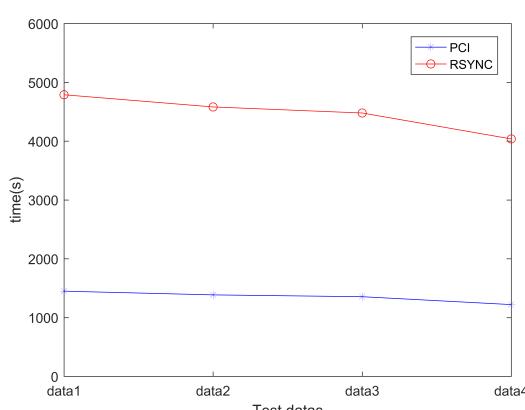
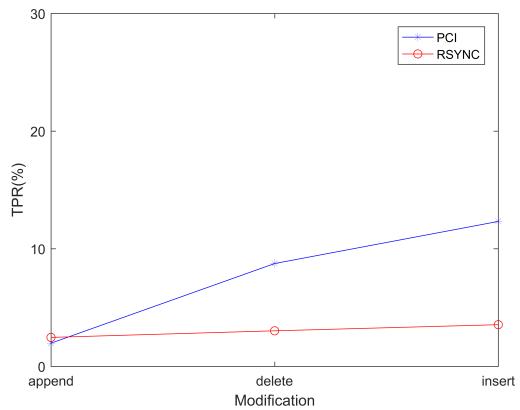
paper [23], [37] to simulate the changes of the file in practice. Our modification includes three ways: append, random delete and random insert. The experimental data set is shown in Table 4.

We use the latest version 3.1.3 of Rsync algorithm, which can be downloaded at the website <https://rsync.samba.org/>. Since we only force on the incremental synchronization performance at a chunk level, we do not consider the incremental synchronization at the file level. During the experiments, we use two virtual machines to simulate the two servers in the real environment. The communication between virtual machines is realized based on Netty. Due to the communication between virtual machines, there will be almost no network congestion, so we compare the actual amount of transferred data to judge the performance of network transmission. The processing time in the experiments includes the chunking time on the target file, the time of producing the checksum, the chunking time on the original file and the time of finding the difference chunks, without the time of network transmission. The actual amount of transferred data contains all the data transferred in two transmissions. The ideal amount of data transferred equals the changed data. We adjust the parameters to ensure that the number of chunks generated by the two algorithms is almost the same. Our project can be found at the website <https://github.com/zhang03091354/Sync>.

C. PROCESSING TIME

In this section, we discuss the proceeding time of PCI and Rsync. The experiment results based on appending, random deletion and insertion modifications are shown in Fig. 35, Fig. 36 and Fig. 37 respectively.

As we can see, PCI has less processing time than Rsync. In the case of appending, although PCI as a CDC algorithm is slightly slower than fixed length chunking algorithm, PCI only needs to calculate one strong check, while Rsync needs to calculate weak check more. It can be seen from the Fig. 35 that the processing time of PCI is about 24% less than Rsync algorithm. In the case of deletion, because Rsync adopts the sliding check in step 2, and in the case of random deletion, it results in byte by byte sliding check, which consumes a lot of computation. From the Fig. 36, it can be seen that the processing time of PCI is about 52% less than that of Rsync. In the case of insertion, there will be more sliding checks, which will increase the time-consuming of Rsync. It can be seen from the Fig. 37 that the processing time of PCI is about 70% less than that of Rsync algorithm.

**FIGURE 35.** The processing time on appending modification.**FIGURE 38.** The TCR on data1.**FIGURE 36.** The processing time on deletion modification.**FIGURE 39.** The TCR on data2.**FIGURE 37.** The processing time on appending modification.**FIGURE 40.** The TCR on data3.

D. ACTUAL AMOUNT OF DATA TRANSFERRED

In this section, we discuss the proceeding time of PCI and Rsync. First, let's introduce a concept: TCR(Transmission Compression Rate), which is the compression rate when transferring data for Incremental synchronization. TCR can be calculated with the following formula.

$$CTR = \frac{D1}{D2}$$

where D1 denotes the actual amount of data transferred and D2 denotes the file size to synchronize. The experiment results based on data1, data2, data3 and data4 are shown in Fig. 38, Fig. 39, Fig. 40 and Fig. 41 respectively.

As we can see, PCI basically transmits more data than Rsync. However, in the case of appending, PCI's TPR is about 0.5% less than Rsync, which is because PCI only needs to transmit a strong check in the first transmission. Although the amount of difference data found is slightly more than Rsync,

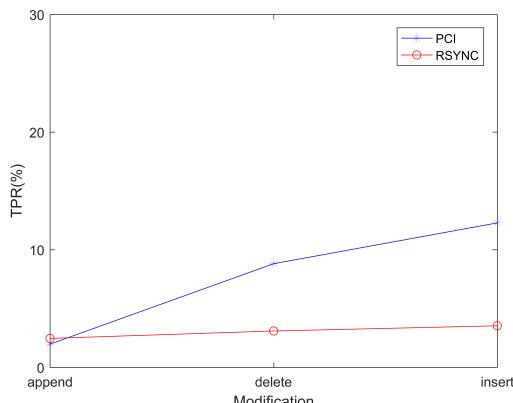


FIGURE 41. The TCR on data4.

the total amount of data transmitted is still slightly less than Rsync. In the case of deletion, PCI has about 5.8 percentage points more TPR than Rsync. In the case of insertion, PCI has about 11.8 percentage points more TPR than Rsync.

To sum up, PCI increases the processing speed by 24%~70% when it loses part of the transmission compression rate compared to Rsync.

VII. CONCLUSION

In this paper, a novel data chunking algorithm is proposed to find incremental data between two similar files. In the field of data incremental synchronization, chunks are used to search for changed data instead of being stored, so the instability of chunk size does not cause much impact. The ability to resist byte shifting can better maintain the status quo of unchanged chunks, thus greatly reducing the amount of changed data found. This algorithm is designed to get a better performance of finding incremental data by improving the byte shifting resistance at the expense of the chunk size stability in the algorithm results. Experiments show that the algorithm can delineate the location of the incremental data in a smaller range, which is reduced by about 32%~57% compared with other algorithms. During comparison with Rsync algorithm based on real-world datasets, PCI has a better performance on calculation speed and a drawback of increasing the Transmission compression rate.

ACKNOWLEDGMENT

(Changjian Zhang and Deyu Qi contributed equally to this work.)

REFERENCES

- [1] Q. N. Nguyen, M. Arifuzzaman, K. Yu, and T. Sato, “A context-aware green information-centric networking model for future wireless communications,” *IEEE Access*, vol. 6, pp. 22804–22816, 2018.
- [2] S. Sanyal and P. Zhang, “Improving quality of data: IoT data aggregation using device to device communications,” *IEEE Access*, vol. 6, pp. 67830–67840, 2018.
- [3] J. Li, J. Wu, and L. Chen, “Block-secure: Blockchain based scheme for secure P2P cloud storage,” *Inf. Sci.*, vol. 465, pp. 219–231, Oct. 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0020025518305012>
- [4] T.-Y. Youn, K.-Y. Chang, K.-H. Rhee, and S. U. Shin, “Efficient client-side deduplication of encrypted data with public auditing in cloud storage,” *IEEE Access*, vol. 6, pp. 26578–26587, 2018.
- [5] Y. Zhou, Y. Deng, L. T. Yang, R. Yang, and L. Si, “LDFS: A low latency in-line data deduplication file system,” *IEEE Access*, vol. 6, pp. 15743–15753, 2018.
- [6] Y. Zhou, D. Feng, Y. Hua, W. Xia, M. Fu, F. Huang, and Y. Zhang, “A similarity-aware encrypted deduplication scheme with flexible access control in the cloud,” *Future Gener. Comput. Syst.*, vol. 84, pp. 177–189, Jul. 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167739X17309238>
- [7] D. Rasch and R. Burns, “In-place rsync: File synchronization for mobile and wireless devices,” in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATEC)*, 2003, p. 15.
- [8] M. Constantinou, “Tuning of rsync algorithm for optimum cloud storage performance,” Dept. Comput. Sci., Univ. Bath, Bath, U.K., Tech. Rep. CSBU-2013-10, 2013.
- [9] J. Ma, C. Bi, Y. Bai, and L. Zhang, “UCDC: Unlimited content-defined chunking, a file-differing method apply to file-synchronization among multiple hosts,” in *Proc. 12th Int. Conf. Semantics, Knowl. Grids (SKG)*, Aug. 2016, pp. 76–82.
- [10] K. Thar, N. H. Tran, S. Ullah, T. Z. Oo, and C. S. Hong, “Online caching and cooperative forwarding in information centric networking,” *IEEE Access*, vol. 6, pp. 59679–59694, 2018.
- [11] H. Noh and H. Song, “Progressive caching system for video streaming services over content centric network,” *IEEE Access*, vol. 7, pp. 47079–47089, 2019.
- [12] X. Zhang, N. Wang, V. G. Vassilakis, and M. P. Howarth, “A distributed in-network caching scheme for P2P-like content chunk delivery,” *Comput. Netw.*, vol. 91, pp. 577–592, Nov. 2015. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1389128615002959>
- [13] X. Ren, Y. Zhou, Z. Huang, J. Sun, X. Yang, and K. Chen, “A novel text structure feature extractor for chinese scene text detection and recognition,” *IEEE Access*, vol. 5, pp. 3193–3204, 2017.
- [14] W. Lu, H. Sun, J. Chu, X. Huang, and J. Yu, “A novel approach for video text detection and recognition based on a corner response feature map and transferred deep convolutional neural network,” *IEEE Access*, vol. 6, pp. 40198–40211, 2018.
- [15] M. Huang and R. M. Haralick, “A method for discovering knowledge in texts,” *Pattern Recognit. Lett.*, vol. 124, pp. 21–30, Jun. 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167865518300801>
- [16] F. Xhafa, “Data replication and synchronization in P2P collaborative systems,” in *Proc. IEEE 26th Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2012, p. 7.
- [17] E. Bertino, G. Guerrini, and I. Merlo, “Trigger inheritance and overriding in an active object database system,” *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 4, pp. 588–608, Jul. 2000.
- [18] J. Lee, K. Kim, and S. Cha, “Differential logging: A commutative and associative logging scheme for highly parallel main memory database,” in *Proc. 17th Int. Conf. Data Eng.*, Nov. 2002, pp. 173–182.
- [19] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*. Accessed: Feb. 1999. [Online]. Available: https://www.samba.org/~tridge/phd_thesis.pdf
- [20] M. Rabin, “Fingerprinting by random polynomials,” Center Res. Comput. Techn., Aiken Comput. Lab., Harvard Univ., Cambridge, MA, USA, Tech. Rep. TR-15-81, 1981.
- [21] R. Raju, M. Moh, and T.-S. Moh, “Compression of wearable body sensor network data using improved two-threshold-two-divisor data chunking algorithms,” in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2018, pp. 949–956.
- [22] N. Björner, A. Blass, and Y. Gurevich, “Content-dependent chunking for differential compression, the local maximum approach,” *J. Comput. Syst. Sci.*, vol. 76, nos. 3–4, pp. 154–203, May 2010, doi: [10.1016/j.jcss.2009.06.004](https://doi.org/10.1016/j.jcss.2009.06.004).
- [23] Y. Zhang, D. Feng, H. Jiang, W. Xia, M. Fu, F. Huang, and Y. Zhou, “A fast asymmetric extremum content defined chunking algorithm for data deduplication in backup storage systems,” *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 199–211, Feb. 2017.
- [24] R. N. Widodo, H. Lim, and M. Atiquzzaman, “A new content-defined chunking algorithm for data deduplication in cloud storage,” *Future Gener. Comput. Syst.*, vol. 71, pp. 145–156, Jun. 2017.

- [25] Y. Won, K. Lim, and J. Min, "MUCH: Multithreaded content-based file chunking," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1375–1388, May 2015.
- [26] Z. Tang and Y. Won, "Multithread content based file chunking system in CPU-GPGPU heterogeneous architecture," in *Proc. 1st Int. Conf. Data Compress., Commun. Process.*, Jun. 2011, pp. 58–64.
- [27] F. Ni, X. Lin, and S. Jiang, "SS-CDC: A two-stage parallel content-defined chunking for deduplicating backup storage," in *Proc. 12th ACM Int. Conf. Syst. Storage (SYSTOR)*, 2019, pp. 86–96, doi: [10.1145/3319647.3325834](https://doi.org/10.1145/3319647.3325834).
- [28] U. Niesen, "An information-theoretic analysis of deduplication," *IEEE Trans. Inf. Theory*, vol. 65, no. 9, pp. 5688–5704, Sep. 2019.
- [29] S. Wu, B. Mao, H. Jiang, H. Luan, and J. Zhou, "PFP: Improving the reliability of deduplication-based storage systems with per-file parity," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2117–2129, Sep. 2019.
- [30] Y. Tan and Z. Yan, "Multi-objective metrics to evaluate deduplication approaches," *IEEE Access*, vol. 5, pp. 5366–5377, 2017.
- [31] W. Tian, R. Li, Z. Xu, and W. Xiao, "Does the content defined chunking really solve the local boundary shift problem?" in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.
- [32] C. Zhang, D. Qi, Z. Cai, W. Huang, X. Wang, W. Li, and J. Guo, "MII: A novel content defined chunking algorithm for finding incremental data in data synchronization," *IEEE Access*, vol. 7, pp. 86932–86945, 2019.
- [33] B. Chapuis, B. Garbinato, and P. Andritsos, "Throughput: A key performance measure of content-defined chunking algorithms," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Workshops (ICDCSW)*, Jun. 2016, pp. 7–12.
- [34] D. M. Bradley and R. C. Gupta, "On the distribution of the sum of n non-identically distributed uniform random variables," *Ann. Inst. Stat. Math.*, vol. 54, no. 3, pp. 689–700, Sep. 2002, doi: [10.1023/A:1022483715767](https://doi.org/10.1023/A:1022483715767).
- [35] J. Draghi and G. P. Wagner, "Evolution of evolvability in a developmental model," *Evolution*, vol. 62, no. 2, pp. 301–315, Feb. 2008.
- [36] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 183–198. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2591272.2591292>
- [37] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuennen, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2012, p. 24. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2342821.2342845>

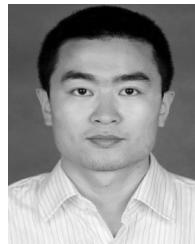


DEYU QI received the M.S. degree from the National University of Defense Technology and the Ph.D. degree from the South University of Technology. He is currently a Full Professor and a Doctoral Supervisor with the School of Computer Science and Engineering, South China University of Technology, an Academic Team Leader of the Advanced Computing Architecture, and the Director of the Research Institute of Computer Systems, South China University of Technology.

His research interests include software developing method and architecture, software developing environment and tools, distributed computing systems, new generation computer architecture, and computer system security. He has published more than 200 journal articles, one monograph, and two educational materials. He also holds many patents for invention and software copyright. He has also held the 863 Project and NSFC Project. He proposed the VLSI dynamic analysis method Fanalysis, object-oriented LOODS abstract model, the large granularity distributed application system interoperation model XIOM, and multidatabase middle-ware DoD.



WENLIN LI received the B.S. and M.S. degrees from Northeast Forestry University, in 2013 and 2016, respectively. She is currently pursuing the Ph.D. degree in computer science and technology with the South China University of Technology.



JING GUO received the B.Eng. degree from Sun Yat-sen University and the M.S. degree from Soochow University. He is currently pursuing the Ph.D. degree (by Research) with the Research Institute of Computer Systems, South China University of Technology. He has authored or coauthored some articles in the areas of numerical solution of differential equations. His research interests are mainly on numerical analysis, data analysis, machine learning, and software engineering.



CHANGJIAN ZHANG received the B.S. degree from the Department of Computer Science and Technology, Xi'an Electronic and Engineering University, in 2013. He is currently pursuing the Ph.D. degree in computer science and technology from South China University of Technology.

Since September 2015, he has been studying on data synchronization for big data at South China University of Technology. His research interests include data synchronization, data compression, data storage, and their applications in cloud computing and big data.