Leap-based Content Defined Chunking --- Theory and Implementation

Chuanshuai Yu, Chengwei Zhang, Yiping Mao, Fulu Li
Huawei Technologies Co., Ltd.
{yuchuanshuai, zhangchengwei, tony.mao, lifulu}@huawei.com

***Abstract*—Content Defined Chunking (CDC) is an important component in data deduplication, which affects both the deduplication ratio as well as deduplication performance. The sliding-window-based CDC algorithm and its variants have been the most popular CDC algorithms for the last 15 years. However, their performance is limited in certain application scenarios since they have to slide byte by byte. The authors present a leap-based CDC algorithm which provides significant improvement in deduplication performance without compromising the deduplication ratio. Compared to the sliding-window-based CDC algorithm, the new algorithm enables up to two-fold improvement in performance.**

Keywords—deduplication; content defined chunking; judgment function; secondary condition

I. INTRODUCTION

The Content Defined Chunking (CDC) algorithm is an important component in data deduplication which breaks the incoming data stream into chunks when the content window before the breakpoint satisfies a predetermined condition. Since chunk is the basic unit in finding duplications, the CDC algorithm has significant impact on the deduplication ratio. In addition, since the whole data stream needs to be chunked before other deduplication processes, the CDC algorithm also affects deduplication performance.

The sliding-window-based CDC algorithm and its variants [1], [2] have been the dominating CDC algorithms in the field for the last 15 years. However, the sliding-window-based CDC algorithm is not optimal, especially in terms of deduplication performance, since it has to slide the content window byte by byte. At each byte a special *judgment function* is computed to judge whether the content window satisfies a predetermined condition. Although the sliding-window-based CDC algorithm can neglect some regions due to the constraint on the minimal chunk size, it still has to compute the judgment function at each byte for almost half of the whole data stream. Therefore, for a 10PB data stream, the CDC algorithm will compute the judgment function 5 Peta times. The Two Thresholds Two Divisors (TTTD) algorithm [3] introduces a secondary condition when judging whether a content window is satisfied to improve the algorithm. However, while it brings improvement to the deduplication ratio, it does not improve deduplication performance. The authors present a leap-based CDC algorithm with novel leap procedures, resulting in significant improvement to deduplication performance. For a 10PB data stream, the new algorithm only needs to compute the judgment function 1 Peta times while still achieving the same deduplication ratio. However, the new algorithm adopts a more complicated judgment function. Overall, the new algorithm demonstrates up to two-fold improvement in performance compared to the sliding-window-based CDC algorithm.

Generally speaking, there are four steps involved in data deduplication: (1) *chunking* which uses a CDC algorithm to divide the data stream into chunks; (2) *chunk fingerprinting* which computes the fingerprint (for example, SHA-1 value) for each of the chunks; (3) *fingerprint indexing and querying* which stores the fingerprints according to a certain data structure for indexing and queries the index to find chunks of identical fingerprints – these chunks are considered as the duplicated chunks; (4) *data storing and management* which stores new chunks and representative reference pointers for the duplicated chunks. There are two important measurements that evaluate a deduplication system: the deduplication ratio and performance. While other steps also play important roles, the chunking step has significant impact on the deduplication ratio and performance.

In the four steps involved in data deduplication mentioned above, the chunking and the chunk fingerprinting step are CPU intensive, and in contrast the fingerprint indexing and querying step as well as the data storing and management step require a lot of memory and disk resources. While there is ongoing research [8], [9] in improving the performance of the fingerprint indexing and querying and the data storing and management steps to reduce the memory and disk resources consumption, there has been less research in improving the performance of the chunking step. Depending on the use cases, the CPU may or may not be the bottleneck, but decreasing the computational complexity of the chunking step will free up CPU resources for other tasks. It remains an open question of how far the chunking step can be optimized while maintaining the property of creating consistent chunks that deduplicate well.

In addition to deduplication performance, the chunking algorithm also has significant impact on the deduplication ratio. An effective chunking algorithm has to satisfy two properties: chunks must be created in a content-defined manner, and all chunk sizes should have equal probability of being selected. The CDC algorithm only breaks the data stream when the content window before the breakpoint satisfies a predetermined condition to meet the *content defined condition*.

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

In this way, when two similar but different data streams show up, the number of duplicated chunks can be maximized. The *equal probability condition* requires all the content windows to have the equal probability to satisfy the predetermined condition. In this way, the chunking algorithm can accommodate all kinds of data streams. In addition, the average chunk size also affects the deduplication ratio. Mostly, the bigger the chunk size, the harder it is to duplicate the chunk and the lower the deduplication ratio. However, the chunk size cannot be too small, since we have to index the fingerprints of all the chunks. It is of particular importance to control the average chunk size and the distribution of chunk sizes in a deduplication system. When searching breakpoints, the sliding-window-based CDC algorithm slides byte by byte to make sure that no breakpoints are missed. However, with a more delicately designed predetermined condition, the leap-based CDC algorithm is able to leap in the process of searching breakpoints without missing a single breakpoint. The sliding-window-based CDC algorithm adopts the rolling hash as the judgment function to judge whether the content window satisfies the predetermined condition. However, rolling hash is not applicable in the new algorithm. Therefore, we introduce the pseudo-random transformation to replace its role.

Our contributions are summarized as follows:

- We introduce a new chunking algorithm with the leap technique in which the executing times of the judgment function are approximately 1/5 of those in the sliding-window-based CDC algorithm. The computation complexity of the judgment function of the new algorithm is less than 2.5 times that of the sliding-window-based CDC algorithm. Therefore, by combining the gain in leap technology and the loss in the judgment function, our leap-based CDC algorithm reduces the computational complexity by as much as 50%. A secondary condition can also be applied to our algorithm.
- We theoretically analyze the distribution of chunk sizes and the computation overhead, with the former one affecting the deduplication ratio and the latter one affecting deduplication performance. Our analysis agrees well with the experimental results.

The rest of this paper is organized as follows. Section II reviews the background of the sliding-window-based CDC algorithm and the TTTD algorithm. Section III describes our proposed leap-based CDC algorithm and adds a secondary condition to it. We provide a theoretical analysis in Section IV. The algorithms are evaluated in Section V. Related work is given in Section VI, and the conclusion is made in Section VII.

II. BACKGROUND

A. Siding-Window-Based CDC

Many previous papers have used CDC algorithms to form chunks. Examples include LBFS [2], TTTD [3] and Zhu's paper [8]. Both LBFS [2] and Zhu's paper [8] adopt an average chunk size of 8KB, and both LBFS [2] and TTTD [3] adopt the minimum and maximum chunk size. We use the sliding-window-based CDC algorithm in a similar way and define the minimum and maximum chunk size as 4KB and 12KB to get a similar average chunk size. The algorithm slides a fixed size window (e.g. 128B) byte by byte in the range [4KB, 12KB] (see Fig. 1). If the content of the window satisfies a predetermined condition, the end point of the window is defined as being *satisfied* and becomes a breakpoint candidate. In this way, each window uniquely corresponds to its end point and thus, a one-to-one mapping between the points and the windows is established. A point x is defined as *satisfied* if the equation "rolling-hash(w) % n = k" holds true, where rolling hash is the judgment function, x represents the end point of the window, w represents the content of the window, n and k are predetermined constants (e.g. n = 4K = 4×1024, k = 0). Sliding-window-based CDC only adopts one predetermined condition; we call it the first condition. In latter sections, we will show that there are algorithms adopting secondary condition, third condition, and so on.

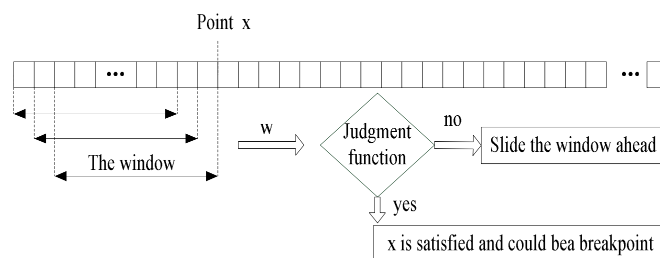

Fig. 1. Streamline of the sliding window-based CDC algorithm. w represents the content of the window, and x represents the end point of the window.

The sliding-window-based CDC algorithm starts from the position of the minimum chunk size and the first satisfied point in the range [4KB, 12KB] will be chosen as the breakpoint. If there is no satisfied point in this range, a breakpoint will be forced at the position of the maximum chunk size. Because of the restriction on the chunk size, not all the satisfied points may become breakpoints. For example, if both point d1 and point d2 satisfy the condition, but the distance between them is 2KB, only point d1 can possibly become a breakpoint. After choosing point d1 as a breakpoint, the next 4KB will be neglected to assure the minimum chunk size.

It is clear that the sliding-window-based CDC algorithm satisfies the content defined condition and, due to the property of the rolling hash, it also satisfies the equal probability condition. However, it has to perform the rolling hash computation at each byte in the range [4KB, 12KB] until it finds the breakpoint. Although the complexity per computation of rolling hash is not high, large numbers of computations could make the sliding-window-base CDC algorithm a potential performance bottleneck in certain application scenarios. To help address this issue, we introduce a leap-based CDC algorithm that reduces the executing times of the judgment function.

The sliding-window-based CDC algorithm only adopts one predetermined condition; however, this could result in a high proportion of forced breakpoints. This will cause an undesired affect on the deduplication ratio. Therefore, TTTD adds a secondary condition to the sliding-window-based CDC algorithm to solve this problem. RC [4] even adopts

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

secondary, ⋯, fifth condition to further reduce the proportion of the forced breakpoints.

B. Adding a Secondary Condition

If some chunks are significantly bigger than others, they are unlikely to be duplicated. In which case, the deduplication ratio will be decreased. Therefore, when the sliding-window-based CDC algorithm cannot find a satisfied point in the given range, it will force a breakpoint at the position of the maximum chunk size. Since forced breakpoints are not content defined, they will also impact the deduplication ratio. A practical way to alleviate this problem is to introduce a secondary condition, which is the main contribution of TTTD [3]. If there is no point satisfying the first condition, TTTD choose the point that satisfies the secondary condition and is closest to the point of the maximum chunk size as the breakpoint. If there is no point satisfying the secondary condition, a breakpoint will be forced at the position of the maximum chunk size.

The secondary condition uses the same sliding window and rolling hash as the first condition, but with a relaxed condition. Point x satisfies the secondary condition if it satisfies the condition "rolling-hash(w) % n' =k' ". Usually, when the length of n is L bit, n' and k' will be chosen as the numbers of the low L-1 bit of n and k, respectively (e.g. n'=2K, k=0). In this way, one can mark the points satisfying the secondary condition when searching for the points satisfying the first condition, which avoids further searches if there is no point satisfying the first condition.

TTTD is still content defined and all the points have the same probability of satisfying the first or the secondary condition. In Section IV we will show that the secondary condition can greatly reduce the proportion of forced breakpoints. Therefore, the deduplication ratio can be improved. However, it does nothing to reduce the executing times of rolling hash; it still has to perform the rolling hash computation at each byte in the range [4KB, 12KB] until it finds a breakpoint.

III. The Leap-based CDC algorithm

In order to reduce the executing times of the judgment function, we present the leap-based CDC algorithm. The new algorithm also satisfies the content defined and the equal probability conditions. In Section IV we will show that the distribution of chunk sizes from this algorithm is almost exactly the same as that of the sliding-window-based CDC algorithm. Therefore, our proposed algorithm improves deduplication performance while guaranteeing the same deduplication ratio.

A. Leap-based CDC

First, we will introduce our algorithm without the secondary condition. In contrast to the sliding-window-based CDC algorithm in which every point corresponds to one window, every point corresponds to M (e.g. 24) windows in our algorithm. The point is said to be *satisfied* and thus becomes a breakpoint candidate only when all the M windows are *qualified*. Referring to Fig. 2, the target point k_i corresponds to windows $W_{i1},\cdots,W_{iM}$, where k_i is the end point of the headmost window. If one window is unqualified, we can skip judging the other M-1 windows. The distance between two adjacent windows is one byte, which means that two adjacent target points share the same M-1 windows. One unqualified window can disqualify up to M target points. In this way we can leap over some target points in the process of searching satisfied points when we encounter an unqualified window. Since rolling hash is not applicable in the new algorithm, we adopt the pseudo-random transformation as the judgment function to define whether a window is *qualified*. Refer to Section III.C for the details on the pseudo-random transformation. The number of qualified windows needed for a target point to become satisfied affects the length of each leap, and combined with the possibility of a single window being qualified, this number affects the distribution of chunk sizes. We will explain how we choose these parameters in Section IV.

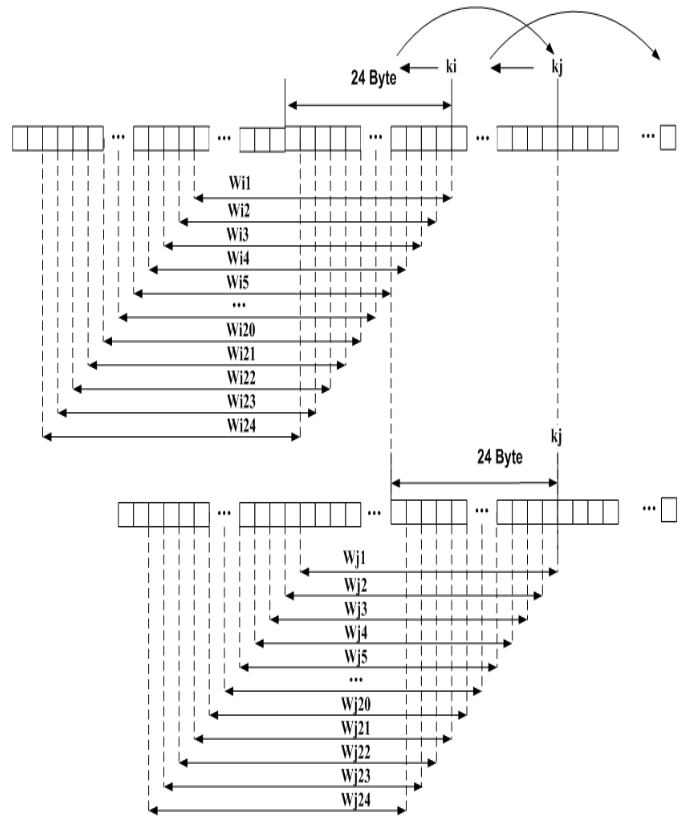

Fig. 2. The target point k_i corresponds to windows $W_{i1},W_{i2},\cdots,W_{i24}$. It is satisfied and becomes a breakpoint candidate when all the 24 windows behind it are qualified.

Here we give an example for a detailed description of the leap technique. Referring again to Fig. 2 and supposing that k_i is the target point, we first judge whether window W_{i1} is qualified. If it is qualified, we then judge window W_{i2}, and so forth. If we find that window W_{i5} is unqualified, it then becomes unnecessary to judge windows $W_{i6},\cdots,W_{iM}$, and we can leap from the end point of window W_{i5}. If we leap less than M bytes and get another target point k_j', the M windows corresponding to point k_j' will include window W_{i5}. Therefore, point k_j' cannot be satisfied. Thus, leaping M bytes and arriving at target point k_j is more efficient. Since we know nothing about the M windows corresponding to this point, we have to judge these windows one by one. As before, we judge

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

them in a reverse order starting from window W_{j1}. If we find M consecutive qualified windows in this process, the corresponding target point will be taken as a breakpoint. If a leap is out of the range of the maximum chunk size, a forced breakpoint will be set at the position of the maximum chunk size.

Generally speaking, leap-based CDC algorithm starts from the position of the minimum chunk size, regards it as a target point, and then moves backward to judge the M corresponding windows one by one. If M consecutive qualified windows are found in this process, the corresponding target point becomes a breakpoint. Otherwise, it will leap from the end point of the first unqualified window it encounters. Then, it gets another target point and starts to judge the windows corresponding to this point. When there is no satisfied point in the predetermined range, a forced breakpoint will be set at the position of the maximum chunk size. In fact, there would be three possible actions after window W_{ix} is judged:

- If window W_{ix} is unqualified, we leap M bytes forward from the end point of window W_{ix} to get another target point and start to judge the M windows corresponding to this point, where $1 \leqslant x \leqslant M$.
- If window W_{ix} is qualified but x is less than M, we slide one byte backward to judge window W_{ix+1}, where $1 \leqslant x \leqslant M-1$.
- If window W_{ix} is qualified and x is equal to M, the corresponding target point becomes a breakpoint, where x=M.

Our leap-based algorithm satisfies the content defined and the equal probability condition as all the windows are tested with the same predetermined condition. It is clear that the new algorithm can greatly reduce the executing times of the judgment function with the leap technique we introduce. However, the leap procedure has a significant impact on the distribution of chunk sizes. How can we control the distribution? Is the distribution of chunk size in our algorithm the same as that of the sliding-window-based CDC algorithm? And how much complexity in computing does our algorithm alleviate? We provide the answers to these and other questions in Section IV.

B. *Adding a Secondary Condition*

We also introduce a secondary condition to the leap-based CDC algorithm to reduce the proportion of the forced breakpoints. As a result, the distribution of chunk sizes becomes smoother and the deduplication ratio improves.

In the sliding-window-based CDC algorithm, the secondary condition uses the same judgment function as the first condition but replaces n' and k' with n and k in the equation. However, the relation between the first condition and the secondary condition in the leap-based CDC algorithm is different from that of the sliding window-based CDC algorithm. In the new algorithm, when the first condition requires M qualified windows, the secondary condition only requires M-T qualified windows (e.g. M=24 , T=2). Additionally, the relative positions of the target point and the corresponding windows also change. Referring to Fig. 3, if windows

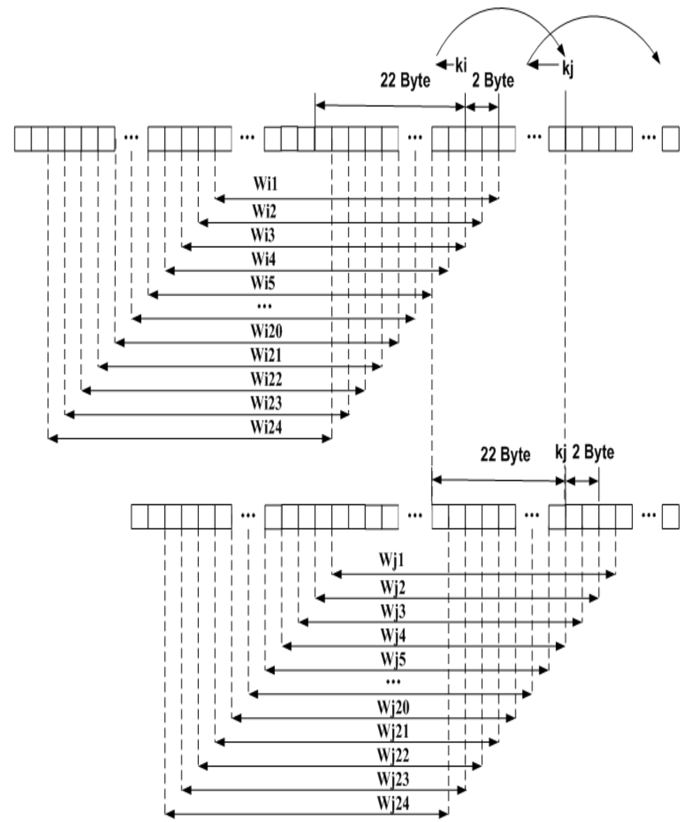

Fig. 3. The target point k_i corresponds to windows $W_{i1},\cdots, W_{i24}$. If windows $W_{i1}, \cdots ,W_{iM}$ are qualified, it satisfies the first condition. If windows $W_{i3},\cdots,W_{iM}$ are qualified, it satisfies the secondary condition.

$W_{i1},\cdots,W_{iM}$ are all qualified, the target point k_i satisfies the first condition and becomes a breakpoint candidate. If windows $W_{i3},\cdots,W_{iM}$ are all qualified, the target point k_i satisfies the secondary condition and also becomes a breakpoint candidate. The windows $W_{i3},\cdots,W_{iM}$ are behind the target point k_i, and the windows W_{i1} and W_{i2} are before the target point k_i. The distance between two adjacent windows is still one byte. For the secondary condition, one unqualified window can only disqualify up to M-T points. Thus, when we look for points satisfying the secondary condition, we can only leap M-T bytes. We still use the pseudo-random transformation to define whether a window is qualified. Again, we will explain how we choose the parameters in Section IV.

Similarly, not all the points satisfying the secondary condition become breakpoints. When there is no point satisfying the first condition, we choose the point that satisfies the secondary condition and is closest to the position of the maximum chunk size as a breakpoint. If there is no point satisfying the secondary condition in the predetermined range, we then force a breakpoint at the position of the maximum chunk size. The secondary condition of leap-based CDC algorithm can also be seen as a relaxed mode of the first condition. In the process of searching the points satisfying the first condition, we must simultaneously mark the points satisfying the secondary condition to avoid another search round in case there is no point satisfying the first condition. In the implementation of our algorithm, the secondary condition

is actually checked first. If the secondary condition fails to be met, a leap is taken. If the secondary condition is satisfied, the first condition will be checked further and a breakpoint will be set accordingly.

Here we give a detailed example of the leap procedure after adding the secondary condition. Referring again to Fig. 3 and supposing k_i is the target point, we first judge whether window W_{i3} is qualified. If it is qualified, we then judge window W_{i4}, and so on. If we find that window W_{i5} is unqualified, since for the secondary condition one unqualified window can disqualify up to M-T points, we can leap up to M-T bytes from the end point of window W_{i5}. After leaping, we can get a new target point k_j and the M new windows corresponding to k_j can be identified. The procedure then continues to repeat. On the opposite case, if windows $W_{i3},\cdots,W_{iM}$ are all qualified, we then judge windows W_{i1} and W_{i2}. If one of them is unqualified, we only find one or two points satisfying the secondary condition. They will be marked and we will leap M-T bytes from the end point of the unqualified window. Then, the new target point can be determined and the process continues. If W_{i1} and W_{i2} are both qualified, then k_i satisfies the first condition and we choose it as a breakpoint. If a leap reaches a point outside the range of the maximum chunk size, either the point satisfying the secondary condition and is closest to the point of the maximum chunk size is set as the breakpoint or the point of the maximum chunk size is set as the breakpoint providing that there is no point satisfying the secondary condition.

Generally speaking, there are five possible actions after the algorithm judges window W_{ix}:

- If window W_{ix} is unqualified, we leap M-T bytes forward from the end point of window W_{ix} to get another target point and start to judge the M windows corresponding to this point, where $T+1 \leqslant x \leqslant M$.
- If window W_{ix} is qualified but x is bigger than T and less than M, we slide one byte backward to judge window W_{ix+1}, where $T+1 \leqslant x \leqslant M-1$.
- If window W_{ix} is qualified and x equals to M, we then judge windows $W_{i1},\cdots,W_{iT}$, where x=M.
- If one of the windows $W_{i1},\cdots,W_{iT}$ is unqualified, we only find one or more points satisfying the secondary condition. They will be marked and we will leap M-T bytes from the end point of the unqualified window to get another target point and start to judge the M windows corresponding to this point, where $1 \leqslant x \leqslant T$.
- If windows $W_{i1},\cdots,W_{iT}$ are qualified, the corresponding target point becomes a breakpoint, where $1 \leqslant x \leqslant T$.

Adding the secondary condition to the leap-based CDC algorithm is much more complicated than adding it to the sliding window-based CDC algorithm. It is difficult to control the distribution of chunk sizes in the leap-based CDC algorithm with a secondary condition. Although rather complicated, the math behind the new algorithm is demonstrated in Section IV.

C. *Pseudo-random Transformation*

In the new leap-based CDC algorithm, the rolling hash is not applicable. Thus, we need to find a proper replacement featuring a light weight computation with the hashing property. We use pseudo-random transformation as the replacement in our algorithm. The following several paragraphs explain the principles behind this replacement.

The idea of pseudo-random transformation came from Locality-sensitive hashing (LSH) [20] and the theorem that the sum or the difference of normal distributions is still a normal distribution. We take advantage of this special property of normal distribution to randomize input data. However, it should be noticed that this transformation is not suitable for encryption. We predetermine two 255×8 matrices $H=\begin{pmatrix} h_{1,1} & \cdots & h_{1,8} \\ \vdots & \vdots & \vdots \\ h_{255,1} & \cdots & h_{255,8} \end{pmatrix}$ and $G=\begin{pmatrix} g_{1,1} & \cdots & g_{1,8} \\ \vdots & \vdots & \vdots \\ g_{255,1} & \cdots & g_{255,8} \end{pmatrix}$, where each element of these matrices is a generated random number of the normal distribution N(0,1). Then, we let the content window determines how to combine – add to or subtract from – these random numbers. Referring to Fig. 4, for every window we choose 5 bytes out of an interval of 42 bytes. The shape ■ belongs to window W_{i1}, the shape ⊠ belongs to window W_{i2}, and so on. We can obtain the 5 bytes of window W_{i2} by sliding the 5 bytes of window W_{i1} one byte backward. We repeat the 5 bytes chosen from a single window 51 times and get 255 bytes. These 255 bytes is the input of the pseudo-random transformation. Each byte has 8 bits, so we get a 255×8 matrix $A=\begin{pmatrix} a_{1,1} & \cdots & a_{1,8} \\ \vdots & \vdots & \vdots \\ a_{255,1} & \cdots & a_{255,8} \end{pmatrix}$. We define $v_{ai,n}$= 1 or = -1, depending on whether $a_{i,n}$=1 or = 0. Therefore, accordingly, we get the matrix $V_a=\begin{pmatrix} v_{a1,1} & \cdots & v_{a1,8} \\ \vdots & \vdots & \vdots \\ v_{a255,1} & \cdots & v_{a255,8} \end{pmatrix}$. We then compute $E_{ai}=v_{ai,1} \times h_{i,1}+\cdots+ v_{ai,8} \times h_{i,8}$. After that, we get 255 values: $E_{a1},\cdots,E_{a255}$. The number of positive values in these 255 values is counted and denoted as E_a. Similarly, we compute $F_{ai}=v_{ai,1} \times g_{i,1}+\cdots+ v_{ai,8} \times g_{i,8}$ and get F_a. If both E_a and F_a are even, we define the window as disqualified. Otherwise, the window is *qualified*.

In the above design, we have utilized the special property of normal distribution: the sum or difference of normal distributions is still a normal distribution. Similar designs and ideas are promoted in other research, like LSH [20]. The pseudo-random transformation aims to get randomized output values for different input data sets. Due to the special symmetry of the normal distribution, E_{ai} is symmetrically distributed in positive and negative values. Therefore, the probability of E_a being even is 1/2 and the probability of a window being qualified is 3/4.

Although the theory behind the pseudo-random transformation is complicated, it reduces the number of needed computations. Here, we analyze its computation complexity. In the transformation, there are 5 bytes for each window, and there are 256 possible values for each byte. Therefore, we build a 5×256 table to record all the possible values that might occur in the calculation of the pseudo-random transformation and thus, accelerate the calculation process. For a specified

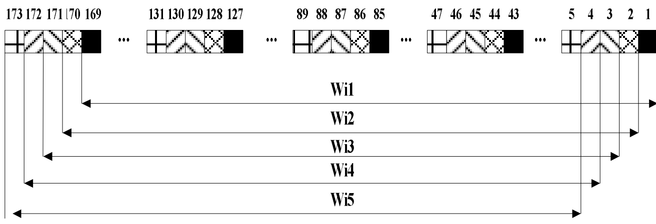

Fig. 4. We choose 5 bytes at intervals of 42 bytes for each window.

byte in a specified place of the window, since this byte has been used 51 times, we can get the 51 corresponding lines of Va. Thus, 51 out of the 255 values (Ea1, ⋯, Ea255) can be computed, and we can count the number of positive values in these 51 values and denote it as Ei. We can compute the values for Fi in the same manner. Since we only care about the parity of Ei and Fi, only one bit is needed where 0 represents even numbers and 1 represents odd numbers. Moreover, we can catenate E_i and F_i. Therefore, the 5×256 table is built which takes E_iF_i as its elements. For each window with the specified 5 bytes, we look up the table 5 times and get the 5 outputs E_1, ⋯,E_5. Then, $E_a= E_1+\cdots+E_5$. The computation for F_a is similar. Actually, the sum operation can be replaced by Xor operation and the computation of E_a and F_a can be completed simultaneously. After the 5 table lookup operations are completed, we can compute $EF= E_1F_1\hat{}\ldots\hat{}E_5F_5$. EF=0 means that both E and F are even and the window is disqualified. EF=1,2,3 means that either or both E and F are odd and the window is qualified. Therefore, one pseudo-random transformation can be done by 5 table lookup operations (the size of the table is 1.25KB) and 4 Xor operations. In contrast, one BUZ hash computation can be done by 2 table lookup operations (the size of the table is 4KB), 2 Xor operations, 2 subtract operations, 2 OR operations and 4 shift operations. Therefore, it is safe to assume the computational complexity of each judgment in our algorithm with the pseudo-random transformation is at most 2.5 times that of the sliding-window-based CDC algorithm with the help of rolling hash. However, the executing times of the judgment function in our algorithm is much less than that in the sliding window-based CDC algorithm. The overall reduction to the computational complexity will be analyzed tin Section IV.

IV. THEORETICAL ANALYSIS

A. Analysis of sliding-window-based CDC

In this section, we will analyze the distribution of chunk sizes and the average chunk size of sliding-window-based CDC algorithm without the secondary condition. Our idea of analyzing the distribution of chunk sizes originates from RC [4]. However, RC only gives a experimental result. We provide a both mathematical analysis and experimental result in this and latter sections.

For a large number of unknown windows, the output values of rolling-hash(w) will subject to the uniform distribution. As a result, the probability that rolling-hash(w)%n=k is 1/n. In the sliding-window-based CDC algorithm, every point corresponds to one window. So the probability of a point being satisfied is 1/n. Thus, the sliding-window-based CDC algorithm satisfies the equal probability condition. We denote x as the point at the

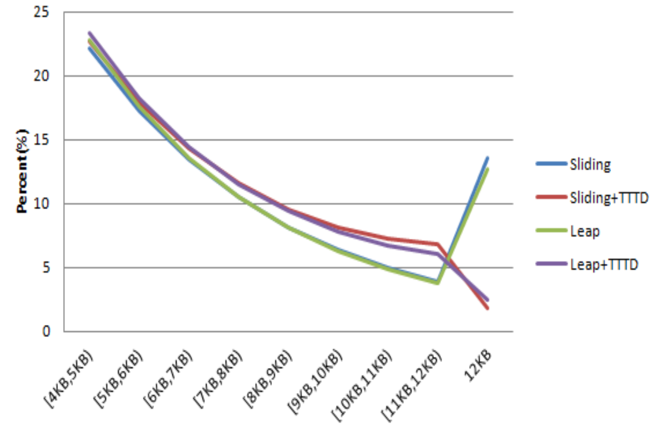

Fig. 5. Distributions of chunk sizes of the four algorithms. The interval is divided into 9 subintervals. The PDF is cumulated inside each subinterval.

position between x byte and x+1 byte, where x∈[4K, 12K]. If point x is a breakpoint, none of the points behind point x should be satisfied as point x is satisfied. Thus, the probability of point x being a breakpoint is $q_{a1}(x)=1/n\times(1-1/n)^{x-4K}$. Since we search for the breakpoint from the minimum chunk size, we will first encounter a satisfied point that is closest to the minimum chunk size. Therefore, although the probability of a point being satisfied is equal, there are more small chunks than large chunks. If we cannot find a satisfied point before reaching the maximum chunk size, we will force a breakpoint at the position of the maximum chunk size. The probability of forced breakpoint partition is $q_a=(1-1/n)^{12K-4K}$. Thus, we can get the probability density function (PDF) of the distribution of chunk sizes for this algorithm, referring to the curve titled "Sliding" in Fig. 5 (e.g. n = 4K). We divide the interval [4KB, 12KB] into 9 subintervals [4KB, 5KB),⋯, [11KB, 12KB), and [12KB, 12KB]. The probability density is cumulated inside these subintervals. It can be seen that the curve is not smooth. It rises suddenly at [12KB, 12KB]. The proportion of forced breakpoints is as high as 13.53%.

The computation of the average chunk size of this algorithm could be divided into two cases: $S_{a1}=\Sigma\, xq_{a1}(x)$ represents the chunk size in case a satisfied point is taken as the breakpoint; $S_{a2}=q_a\times 12K$ represents the chunk size in case of forced partition. Therefore, the average chunk size of this algorithm is $S_a=S_{a1}+S_{a2}$=7.46KB.

B. Adding a secondary condition

In this section, we will analyze the distribution of chunk sizes and the average chunk size of the sliding-window-based CDC algorithm with a secondary condition.

In the sliding-window-based CDC algorithm, the secondary condition does not affect how the first condition determines the breakpoint. Therefore, $q_{b1}(x)$ and S_{b1} are the same as $q_{a1}(x)$ and S_{a1} in the previous section. We define $q_{b2}(x)$ and S_{b2} as the probability and chunk size when the secondary condition is used to determine a point as being a breakpoint and define q_b and S_{b3} as the probability and chunk size when a forced break is to be executed.

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

The probability of the secondary condition determining a breakpoint is a conditional probability – the first condition must first fail to determine a breakpoint. Under the condition hash(w)%n!=k, the probability of hash(w)%n'=k' is 1/(n-1). When we can't find a point satisfying the first condition, we choose the point that satisfies the secondary condition and is closest to the maximum chunk size as a breakpoint. Thus, for the secondary condition, we actually search backwards starting from the point of the maximum chunk size to the point of the minimal chunk size. The probability that point x only satisfies the secondary condition and becomes a breakpoint is $q_{b2}(x)=(1-1/n)^{12K-4K}\times 1/(n-1)\times(1-1/(n-1))^{12K-x}$, where $x\in[4K, 12K]$. The probability that we can't find a point satisfying the secondary condition is $q_b=(1-1/n)^{12K-4K}\times(1-1/(n-1))^{12K-4K}=(1-2/n)^{12K-4K}$. Thus, we can get the PDF of the distribution of chunk sizes for this algorithm, refer to the curve titled "Sliding + TTTD" in Fig. 5 (e.g. n = 4K). The curve is much smoother than that of sliding-window-based CDC algorithm without the secondary condition. Since the subinterval [12KB, 12KB] only contains one point and represents a forced partition, it is best to keep the proportion down to around 1.92%

S_{b1} is the same as S_{a1}. $S_{b2}=\Sigma\, xq_{b2}(x)$. $S_{b3}=q_b\times 12K$. Therefore, the average chunk size is $S_b=S_{b1}+S_{b2}+S_{b3}=7.14KB$. After introducing the secondary condition, the probability of forced partition decreases from 13.53% to 1.92% and the average chunk size decreases from 7.46KB to 7.14KB. Both of these properties will improve the deduplication ratio.

C. Analysis of pseudo-random transformation

The empirical results from testing the pseudo-random transformation are presented in this section followed by a brief discussion on the idea for adopting pseudo-random transformation and how it originates from LSH [20] and Simhash [21].

Observably, the larger the N, the more stable the pseudo-random transformation, where N represents the number of the lines of $N\times 8$ matrices H or G. For example, we must not use a $1\times Z$ matrix, even when Z is much larger than 8. Continuing, suppose we define $H=(h_{1,1}\cdots h_{1,Z})$, compute $E_a=v_{a1,1}\times h_{1,1}+\ldots+v_{an,8}\times h_{1,Z}$, output 0 when $E_a<0$ and output 1 when $E_a\geqslant 0$. There is a probability that $\{h_{1,1}\cdots h_{1,Z}\}$ contains one or two big numbers, then the output will be greatly influenced by these numbers. However, for a $N\times 8$ matrix, when one or two lines of the matrix contain big numbers, only one or two E_{ai} are affected; there remains an equal probability that E_a will be even or odd. The larger the N, the more likely E_a will be symmetric in terms of even and odd occurrences.

In another observation, the bigger the r, the more stable the pseudo-random transformation, but the worse the performance, where r represents the number of bytes we chosen from each window. We repeat these r bytes c times, where $r\times c=N$. To be effective, however, r cannot be too small. For example, when r=1, a specific byte will occur too frequently in a data stream, the output of the pseudo-random transformation will become skewed. It is also necessary to choose the r bytes in intervals to make the pseudo-random transformation more resistant to frequent long phrases. The opposite is also true: r cannot be too big because there are r table lookup operations and r-1 Xor operations required in one calculation of the pseudo-random transformation. If the value for r is too big, performance may significantly deteriorate. Therefore, the value for r must be carefully chosen to balance stability with performance in the pseudo-random transformation.

Our experiments show that, when both N and r are of the appropriate size (for example, set N=755, r=755, c=1) for 60 predetermined matrix pairs $\{(H_i, G_i)|\ i=1,\cdots,60\}$ which are randomly generated by Matlab, almost every pair demonstrates good adaptability in the leap-based CDC algorithm. However, when both N and r are set to an overly small value (for example, N=255, r=5, c=51), only 1/10 of the 60 predetermined matrix pairs $\{(H'_i, G'_i)|\ i=1,\cdots,60\}$ demonstrate good adaptability, indicating that the matrix pair must be carefully chosen. We use an optimum pair in the following experiments. It can be seen that the pseudo-random transformation acts poorly when both N and r are small, and it acts well when both N and r are appropriately large. This is consistent with our analysis results.

It should be noted that the technique of adopting numbers generated by normal distribution has been used in LSH [20]. Both LSH and Simhash [21] aim to get similar output values from similar input data sets. However, our pseudo-random transformation works in the opposite direction; that is, it attempts to get random output values for different input data sets. LSH and Simhash also use a factor resembling the E_a, but there is a small difference: LSH and Simhash care whether two counters E_a and E'_a are close to each other, but pseudo-random transformation pays attention to the parity of E_a. It is interesting that such a small change can result in such a big difference.

D. The analysis of leap-based CDC

Different from the sliding-window-base CDC algorithm, the distribution of chunk sizes of the leap-based CDC algorithm is determined by parameters M and P_w, where M represents the number of qualified windows needed for a point to become satisfied and P_w represents the possibility that a window is qualified. The former affects the length of each leap and the latter affects the frequency of the leap. These two parameters determine both the distribution of chunk sizes and the performance of the leap-based CDC algorithm. After some theoretical analysis and a large amount of experiments, we were able to choose the optimal parameter values: M=24; $P_w=3/4$.

How M and P_w determine the distribution of chunk sizes is very similar to a multi-step Fibonacci sequence. We define F(x) as the probability that there is no satisfied point at or before point x. The leap-based CDC algorithm starts from point 4096, and the probability that all the M windows corresponding to this point are qualified is $(3/4)^{24}$. Thus, $F(4096)=1-(3/4)^{24}$. There is no breakpoint before point 4096, so F(4095)= 1, …,F(4073)=1.

In the following, when we say a window is corresponding to a point, it means the point is the end point of the window. For example, the window corresponding to point x is a window that ends at point x. This concept should not be confused with

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

the windows-point corresponding relation in the definition of our leap-based CDC algorithm.

Referring to Fig. 6, at point x, the computation of F(x) can be divided into 24 cases:

Case 1: the window corresponding to point x is unqualified, and there is no satisfied point at or before point x-1. The probability of this case is 1/4×F(x-1). F(x) includes this case. The case that “the window corresponding to point x is unqualified” and “there is at least one satisfied point at or before point x-1” has no affect on F(x).

Case 2: the window corresponding to x point is qualified, the window corresponding to point x-1 is unqualified, and there is no satisfied point at or before point x-2. The probability of this case is 3/4×1/4×F(x-2). F(x) includes this case. The case that “the window corresponding to point x is qualified”, “the window corresponding to point x-1 is unqualified” and “there is at least one satisfied point at or before point x-2” has no affect on F(x).

……

Case 24: the window corresponding to point x is qualified,⋯, the window corresponding to point x-22 is qualified, the window corresponding to point x-23 is unqualified, and there is no satisfied point at or before point x-24. The probability of this case is $(3/4)^{23}\times 1/4\times F(x-24)$. F(x) includes this case. The case that “the window corresponding to point x is qualified,⋯, the window corresponding to point x-22 is qualified, the window corresponding to point x-23 is unqualified” and “there is at least one satisfied point at or before point x-24” has no affect on F(x). The case that “the window corresponding to point x is qualified,⋯, the window corresponding to point x-23 is qualified” has no affect on F(x).

Putting all the results from the above cases together, $F(x) = 1/4\times F(x-1)+1/4\times(3/4)\times F(x-2)+\ldots\ldots +1/4\times (3/4)^{23}\times F(x-24)$. And, 1-F(x) is the probability that we find at least one satisfied point at or before point x. Thus, $q_{d1}(x)=(1-F(x))-(1-F(x-1))=F(x-1)-F(x)$ is the probability that the point x is a satisfied point and there is no satisfied point at or before point x-1 (thus, point x is a breakpoint). If we can’t find a satisfied point before reaching the maximum chunk size, we will force a breakpoint at the position of the maximum chunk size. The probability of this case is $q_d=F(12K)=12.64\%$. Thus, we can get the PDF of the distribution of chunk sizes for this algorithm (refer to the curve titled “Leap” in Fig.5). The curve is quite similar to that of the sliding-window-based CDC algorithm without a secondary condition.

Consequently, the computation of the average chunk size of the leap-based CDC algorithm can be calculated below: $S_d=S_{d1}+S_{d2}$, where $S_{d1}=\sum x q_{d1}(x)$ and $S_{d2}=q_d\times 12K$, corresponding to non-forced breakpoint case and forced breakpoint case, respectively. The average chunk size is S=7.38KB, which is a little smaller than that of the sliding-window-based CDC algorithm without a secondary condition.

For the leap-based CDC algorithm with a secondary condition, the computation of the distribution of chunk sizes and the average chunk size is similar to the above calculation, but a bit more complicated. The result is shown in Fig. 5; refer

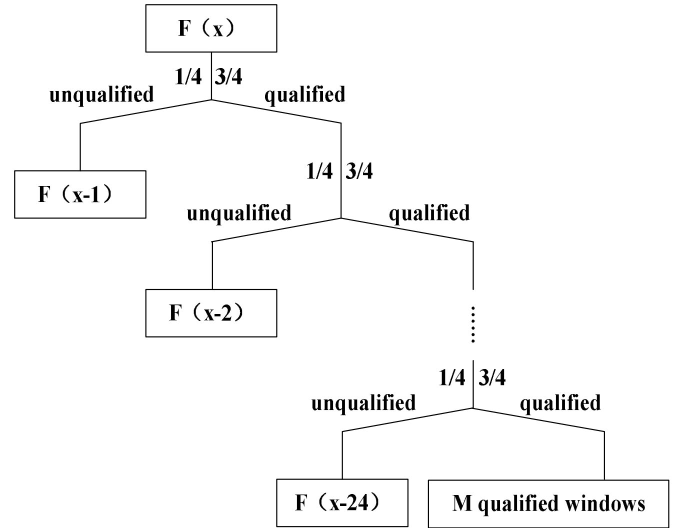

Fig. 6. F(x) only contains 24 cases. Other cases have no affect on it.

to the curve titled “Leap + TTD”. The PDF of the distribution of chunk sizes is similar to that of the sliding-window-based CDC algorithm with a secondary condition. The average chunk size is 7.08KB. Therefore, our leap-based CDC algorithm with a secondary condition acts almost exactly the same as the sliding-window-based CDC algorithm with a secondary condition. Therefore, the deduplication ratios of the two algorithms are also nearly exactly the same.

E. Executing times of the judgment function

The rolling hash and the pseudo-random transformation are used as the judgment functions for the sliding-window-based CDC algorithm and the leap-based CDC algorithm, respectively. In this section, we will analyze the computational complexity for executing these judgment functions in the corresponding algorithms.

In the sliding-window-based CDC algorithm without a secondary condition, the judgment function has to be executed once at each byte in the interval [4KB, 12KB] until a breakpoint is reached. Since the average chunk size is 7.46KB, the judgment function has to be executed 3.46K times on average during chunking of an average size chunk. In the sliding-window-based CDC algorithm with a secondary condition, because the first and the secondary condition use the same rolling hash, the judgment function is also executed once at each byte in the interval [4KB, 12KB] until a breakpoint is reached. Because the average chunk size is 7.14KB, the judgment function is executed 3.14K times on average during chunking an average sized chunk.

Now, we turn to consider the case of the leap-based CDC algorithm without a secondary condition. After each leap, we have to judge window W_{i1} first. According to our parameter selection, we know that the probability that this window is unqualified and that the calculation will leap forward is 1/4; the probability that this window is qualified and that the calculation slides one byte backward and continues to judge the next window is 3/4. Based on these probabilities, the judgment function is executed $1+3/4+(3/4)^2+\cdots+(3/4)^{24}=4$ times on average after a leap. In the meantime, the calculation slides

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

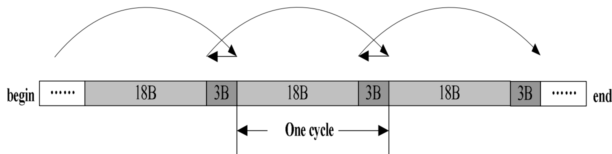

Fig. 7. For the leap-based CDC algorithm without a secondary condition, the average leap length is 24 bytes with a backward slide of 3 bytes; 4 windows are judged then another 24 bytes are leaped, and so on.

back 3 bytes on average (refer to Fig. 7). Because the average leap distance is 24 bytes, the calculation procedure follows the following pattern: leaps 24 bytes, slides back 3 bytes, executes 4 judgment functions on average, and so on. Therefore, the judgment function is executed 4 times for every 21 bytes in the interval [4KB, 12KB] until the breakpoint is set. Since the average chunk size is 7.38KB, the judgment function is executed 4/21×3.38K times on average for every 7.38KB. Therefore, the number of times the judgment function is executed in the leap-based CDC algorithm without a secondary condition is about 1/5 that of the sliding-window-based CDC algorithm with or without a secondary condition. Since the computational complexity of the pseudo-random transformation at most 2.5 times that of the rolling hash, the leap-based CDC algorithm without a secondary condition reduces the computational complexity by half when compared to the sliding-window-based CDC algorithm .

For the leap-based CDC algorithm with a secondary condition, the conclusion is the same. Considering the length of the more complex analysis required with the leap-based algorithm, we have decided to skip presentation of such information herein and maintain the conclusion that the leap-based CDC algorithm with a secondary condition reduces the computation overhead by half when compared to the sliding-window-based CDC algorithm.

V. Experiment

In this section, we show the experimental results from our algorithm and compare them with those from the sliding-window-based CDC algorithm. Additionally, to improve the deduplication ratio, we only compare the results when both algorithms use a secondary condition. Although Rabin hash [18] is widely used in academia papers, it can be replaced by faster functions in industry. We decided to use BUZ hash [19] as the rolling hash for the sliding-window-based CDC algorithm in the experiments because of its lighter CPU overhead than the Rabin hash [18]. Some experiments show that a sliding-window-based CDC algorithm with BUZ hash is much faster than a sliding-window-based CDC algorithm with Rabin hash, while the deduplication ratios provided by the two algorithms are almost the same.

A. Environment

The sliding-window-based CDC algorithm processes the data in a streamlined manner, but the leap-based CDC algorithm determines whether or not to leap depending on the judgment result. The probability that the sliding-window-based CDC algorithm continues to slide forward is as high as 1-1/4K, but there are much more branches when judging the windows in the leap-based CDC algorithm and all these branches have reasonable probabilities. Thus, the sliding-window-based CDC algorithm performs well if the CPU is specially optimized for streamline input. However, the leap-based CDC algorithm can significantly improve the performance if the CPU is powerful when dealing with branches. These two algorithms will behave differently under different CPU architectures.

We use the Westmere CPU architecture and Sandy Bridge CPU architecture in our experiment, where Sandy Bridge CPU architecture represents the newer generation architecture and is more powerful when dealing with branches while Westmere CPU architecture represents the older generation architecture and is less powerful when dealing with branches. We choose Intel E5520 as the representative of Westmere CPU architecture and Intel E5-2450 as the representative of Sandy Bridge CPU architecture. It should be noted that although Sandy Bridge CPU E5-2450 has a newer architecture, its frequency is lower. Detailed environments are shown in Table 1.

Only the performance of chunking step is measured. When the data stream is too big, we run the experiment in multiple rounds. In each round, we read 256MB of data into the memory, start the timer, chunk the data, stop the timer, compute the fingerprints and then find duplicated chunks. The performance of all rounds is accumulated. In this way, we shield other uncertain factors. But the deduplication ratio, the distribution of chunk sizes and the average chunk size are measured through the whole data stream.

B. Datasets

We tested the two algorithms on 10 datasets (see Table 2). All datasets were collected from real production environments. The VMware dataset is collected from 10 Windows7 VMs using Symantec NetBackup software. The Oracle-Rman dataset is collected from a real database using the Rman interface. The Oracle-Dmp and Oracle-dbf datasets are the original forms of the database. The ISO dataset is collected from 20 Windows system installation files. The Sys dataset is collected from the C disk of 20 users. The Office, PDF, Music and Video datasets are also collected from real environments.

C. Distributions of chunk sizes and average chunk size

The distribution of chunk sizes has no relation to the CPU architecture, and the results would presumably be exactly the same for the two CPU architectures. Referring to Fig. 8, the distributions of chunk sizes for the sliding-window-based CDC algorithm with a secondary condition and the leap-based CDC algorithm with a secondary condition are similar, which are also similar to the curves of the theoretical analysis. Referring to Fig. 9, the Music and Video datasets are essentially random data sets, so the average chunk sizes of the two algorithms agree with our analysis. But for other datasets, the average chunk sizes of both algorithms fluctuate between [6.4KB, 7.6KB]. However, the relation is maintained that the average chunk size of the leap-based CDC algorithm with the secondary condition is a little smaller than that of the sliding-window-based CDC algorithm with the secondary condition. In any case, the two algorithms have similar distributions of chunk sizes and average chunk sizes, which then implies they will have almost exactly the same deduplication ratios.

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

TABLE I. TESTING ENVIRONMENT.

operating system	CPU			memory (GB)	disk
	type	cache size	cpu cores		
SUSE Linux Enterprise Server 11 SP1	Sandy Bridge E5-2450 0 @ 2.10GHz	20480(KB)	8	47	2TB SATA 7.2K rpm * 1
SUSE Linux Enterprise Server 11 SP1	Westmere E5520 @ 2.27GHz	8192(KB)	4	47	2TB SATA 7.2K rpm * 1

TABLE II. TESTING DATASETS

Type	Size(KB)	the way we generated them
vmware	81300750	This dataset is gotten by backuping 10 VMvare files of Windows7 system by NetBackup software.
oracle_tbs_rman	14427720	This dataset is gotten by backuping a real database by RMAN interface.
oracle_tbs_dmp	10602880	This is the dmp file of a real database.
oracle_tbs_dbf	15990792	This is the dbf file of a real database.
sys	153871100	This dataset collects data of 20 C disks. The data is packed together without compression.
ISO	45486080	This dataset collects 20 ISO install files different versions of Windows operating system.
office	18114600	This dataset collects all kinds of office files, including doc, xls, ppt and so on. These files are packed together without compression.
music	4556260	This dataset collects all kinds of music files. These files are packed together without compression.
video	11327510	This dataset collects all kinds of video files. These files are packed together without compression.
pdf	4714870	This dataset collects all kinds of pdf files. These files are packed together without compression.

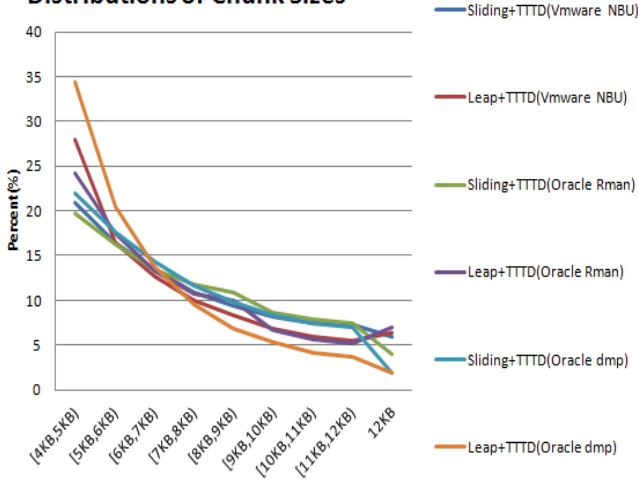

Fig. 8. Distributions of chunk sizes of the two algorithms with a secondary condition (They behave similarly).

D. Deduplication ratio

The deduplication ratio also has no relation to the CPU architecture, and the results would presumably be exactly the same for the two CPU architectures. Referring to Table III, the deduplication ratio of the two algorithms are almost exactly the same (there are no duplicated chunks in Oracle dmp and Oracle dbf datasets in either of the two algorithms). As long as two chunking algorithms are content defined and have similar distributions of chunk sizes, their deduplication ration will

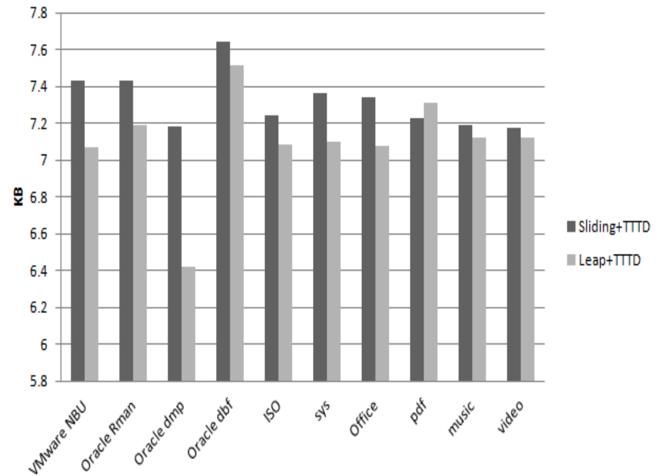

Fig. 9. Average chunk sizes of of the two algorithms with a secondary condition.

TABLE III. DEDUPLICATION RATIO

	VMware NBU	Oracle Rman	ISO	sys	Office	pdf	music	video
Sliding+ TTTD	7.84640	1.00025	2.01905	3.56711	1.28722	1.05202	1.10806	1.00009
Leap+ TTTD	7.81146	1.00058	2.01983	3.55052	1.28828	1.05269	1.10852	1.00001

likely be similar. To have a flat distribution of chunk sizes, the chunking algorithm must satisfy the equal probability condition. Therefore, as we mentioned before, the chunking algorithm must satisfy the content defined and the equal probability condition to achieve a good deduplication ratio. As such, the conclusion that the leap-based CDC algorithm with a secondary condition has a similar deduplication ratio as that of the sliding-window-based CDC algorithm with a secondary condition is proven to be valid by the experiments.

E. Performance

The performance of the two algorithms is measured when the CPU is free from other tasks. Although in a real system such a condition cannot hold and CPU should provide all kinds of services simultaneously including computing fingerprints, we only use the relative experimental results to accurately compare the two algorithms. Referring to Fig. 10 and Fig. 11, the two algorithms behave differently under Westmere and Sandy Bridge CPU architectures. Under the newer Sandy Bridge CPU architecture, the leap-based CDC algorithm with a secondary condition improves performance by 50%~100% compared to the sliding-window-based CDC algorithm with a secondary condition. But under the older Westmere CPU architecture, the leap-based CDC algorithm only improves performance by 10%~30%. We believe that this distinction in performance based on different CPUs is due to the fact that Sandy Bridge CPU architecture is more powerful when dealing with branches and the leap-based CDC algorithm with a secondary condition involves lots of branches in its calculation.

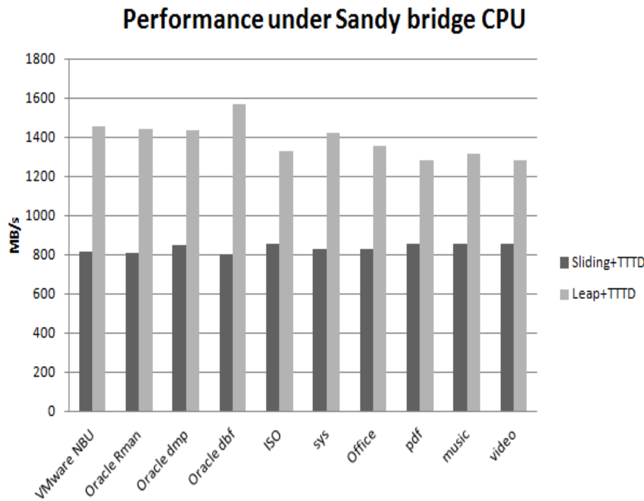

Fig. 10. Performance of the two algorithms under Sandy Bridge CPU. For small datasets, the experiment is run several times to obtain an average value. Our algorithm improves performance by 50~100%.

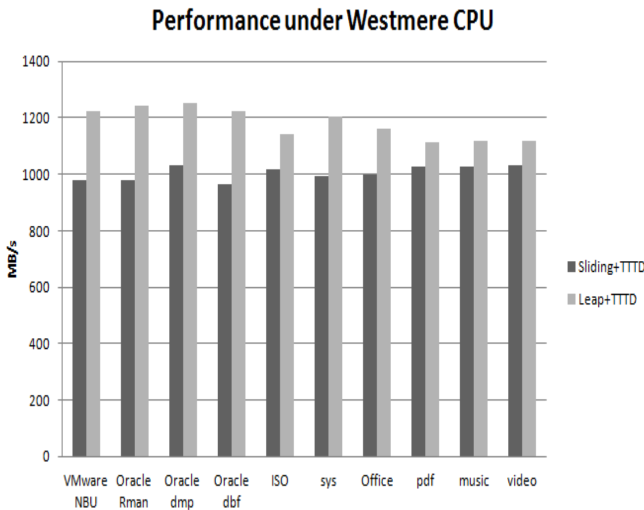

Figure 11: Performances of the two algorithms under Westmere CPU. For small datasets, the experiment is run several times to obtain an average value.Our algorithm improves performance by 10~30%.

Therefore, the leap-based CDC algorithm runs faster under the newer Sandy Bridge CPU architecture. It should also be noted that although Sandy Bridge CPU is the newer CPU architecture, the sliding-window-based CDC algorithm runs slower under it.

VI. Related Work

The chunking algorithm is one of most the important modules in the deduplication system. The sliding-window-based CDC algorithm [1] and its variants have been the most popular CDC algorithms for the last 15 years. These algorithms satisfy the content defined condition and equal probability condition, so the deduplication ratio can be guaranteed. However, their performance is limited in certain application scenarios since they have to compute the judgment function once at each byte. LBFS [2] proposed limiting the minimum chunk size and the maximum chunk size to help eliminate chunks that are too small or too large. The LBFS approach is appealing because it helps make the deduplication ratio more stable and improves deduplication performance with skipping of the minimum chunk size when searching for breakpoints. Our algorithm followed this limitation. The secondary condition of breakpoint first appeared in TTTD [3]. It reduces the proportion of the forced breakpoints and improves the deduplication ratio. RC [4] adopted secondary, third, fourth, and fifth conditions to further reduce the proportion of forced breakpoints. They also compared the distribution of chunk sizes of the RC algorithm and the sliding-window-based CDC algorithm. Our idea of analyzing the distribution of chunk sizes and average chunk size originates from this paper. However, LBFS, TTTD, and RC are all based on the sliding window CDC algorithm, so they all have to compute the judgment function once at each byte for almost half of the whole data stream.

Bimodal CDC [5] also used the same sliding-window-based CDC algorithm, but it mixes chunks of different average sizes together. This algorithm first chunks the data stream into large chunks and then splits part of them into small chunks. The reverse is also true as it can first chunk the data stream into small chunks and then combine part of them into large chunks. This algorithm can significantly reduce the amount of metadata that needs to be indexed but at the cost of a slight loss in the deduplication ratio. However, they have to check the fingerprint index to determine whether to split large chunks or merge small chunks. Similarly, Lu [6] also mixed chunks of different average size together, but determined whether to chunk the data stream into large chunks or small chunks according to the reference count. Meyer and Bolosky [7] compared the deduplication ratio of chunking algorithms adopting different average chunk sizes.

Zhu et al [8] proposed the locality keeping technique which stores the fingerprints sequentially in containers to avoid disk bottlenecks. Sparse indexing [9], extreme bin [10], SILO [11] sampled the fingerprints of chunks to index them. These techniques can greatly reduce the consumption of memory in the querying step. DBLK [12], BloomStore [13], chunkstash [14], and delta index [15] discussed some methods for memory organization of the sampled fingerprints. These algorithms can further reduce memory consumption. Idup [16] only deduplicated sequences of duplicated chunks. Lillibridge et al [17] limited the number of containers that a group of chunks can refer to. These two techniques can reduce fragments and keep the locality for a long time. The above algorithms can greatly alleviate the load on disks and memory, but in certain application scenarios, the chunking step and the chunk fingerprinting step which cost a lot of CPU resources could become new bottlenecks.

The rolling hash helps increase the calculation speed in the sliding-window-based CDC algorithm. Both Rabin hash [18] and BUZ hash [19] are the popular rolling hash functions. We adopted BUZ hash [19] in the sliding--window-based CDC algorithm due to its lighter CPU overhead than the than Rabin hash [18]. As the complexity of one computation of BUZ hash is too small to be further compressed, the only way to alleviate possible bottlenecks in the chunking step is to reduce the executing times of judgment function.

The technique of adopting numbers from normal distribution has been used in LSH [20]. LSH used a more generalized distribution called p-stable distribution instead of normal distribution. Research from Manku et al [21] and Datar et al [20] shows that Simhash and LSH are similar. We built the pseudo-random transformation based on these two algorithms.

VII. CONCLUSION

The chunking algorithm affects not only the deduplication ratio but also deduplication performance. Since the sliding-window-based CDC algorithm executes the judgment function once at each byte for almost half of the whole data stream, its heavy computing overhead provides an area for further optimization. In this paper, we presented the leap-based CDC algorithm and added a secondary condition to it in order to reduce the computing overhead and maintain the same deduplication ratio. Our algorithm satisfies both the content defined condition and the equal probability condition. As we illustrated and verified through experiments, the leap-based CDC algorithm with or without a secondary condition can significantly reduce the computing overhead while maintaining the same deduplication ratio. To resolve the technique issue of not being able to use the rolling hash in the new algorithm, we introduced the pseudo-random transformation to replace the role of rolling hash. The analysis and the experiments have shown that the pseudo-random transformation is an appropriate replacement.

We then analyzed the distribution of chunk sizes, the average chunk size, and the computational complexity of these algorithms. The theoretical analysis shows that the distribution of chunk sizes among all analyzed algorithms are fairly similar; the average chunk sizes from all analyzed algorithms are very close; and the computational complexity of the leap-based CDC algorithm is approximately half that of the sliding-window-based CDC algorithm.

The experimental results substantiate our theoretical analysis.

ACKNOWLEDGMENTS

We are grateful to our shepherd Philip Shilane and the anonymous reviewers of this paper. We thank Guangbin Yan, Jianghai Gao, Mingshun Liu and Feng Li for their support. Our test team includes Peixue Liu, Yong Liu, Zhiyong Liu, and Qilong He. Our deduplication team includes Zhang Zongquan, Yanghuadi, Qiang Liu, Linbo Xu, You Jun, Xiaobo Liu, Quancheng Sun, Yanhui Zhong, Xudong Fu, and Zhenwen Xue. We thank those named above and the many others who have remained unnamed.

REFERENCES

[1] R N. Williams, “Method for partitioning a block of data into subblocks and for storing and communcating such subblocks”, U.S. Patent 5,990,810. 1999-11-23.

[2] A. Muthitacharoen, B. Chen, D. Mazieres, “A low-bandwidth network file system”, ACM SIGOPS Operating Systems Review, ACM, 2001, 35(5): 174-187.

[3] K. Eshghi, H. K. Tang, “A framework for analyzing and improving content-based chunking algorithms”, Hewlett-Packard Labs Technical Report TR, 2005, 30: 2005.

[4] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, & S. Sengupta, “Primary data deduplication-large scale study and system design”, USENIX annual technical conference, USENIX Association, 2012: 285-296.

[5] E. Kruus, C. Ungureanu, C.Dubnicki, “Bimodal Content Defined Chunking for Backup Streams”, Conference on File and Storage Technologies, USENIX Association, 2010: 239-252.

[6] Lu Guanlin, “An Efficient Data Deduplication Design with Flash Memory Based SSD”, A Dissertation Submitted to th Faculty of the Graduate School of the University of Minnesota.

[7] D. T. Meyer, W. J. Bolosky, “A study of practical deduplication”, ACM Transactions on Storage, 2012, 7(4): 14.

[8] B. Zhu, K. Li, R. H. Patterson, “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System”, Conference on File and Storage Technologies, USENIX Association, 2008,8:1-14.

[9] M. Lillibridge, K. Eshghi, D. Bhagwat, V Deolalikar, G.Trezis& P.Camble. “Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality”, File and Storage Technologies, USENIX Association,2009, 9: 111-123.

[10] D. Bhagwat, K. Eshghi, D. D. Long, & M. Lillibridge, “Extreme binning: Scalable, parallel deduplication for chunk-based file backup”, Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009: 1-9.

[11] W. Xia, H. Jiang, D. Feng, Y.Hua, “Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput”, USENIX annual technical conference, USENIX Association, 2011: 26-28.

[12] Y. Tsuchiya, T.Watanabe, “Dblk: Deduplication for primary block storage”, Mass Storage Systems and Technologies, IEEE, 2011: 1-5.

[13] G. Lu, Y. J. Nam, D. H. C. Du, “BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash”, Mass Storage Systems and Technologies, IEEE, 2012: 1-11.

[14] B. Debnath, S. Sengupta, J.Li, “ChunkStash: speeding up inline storage deduplication using flash memory”, USENIX annual technical conference, USENIX Association, 2010: 16-16.

[15] N. H. Margolus, E. Olson, M. Sclafani, C. J. Coburn & M. Fortson, “Storage system for randomly named blocks of data”, U.S. Patent RE45,350, 2015-1-20..

[16] K.Srinivasan, T.Bisson, G. R.Goodson & K.Voruganti, “iDedup: Latency-aware, inline data deduplication for primary storage”, File and Storage Technologies, USENIX Association, 2012: 24-24.

[17] M. Lillibridge, K. Eshghi, D. Bhagwat, “Improving restore speed for backup systems that use inline chunk-based deduplication”, File and Storage Technologies, USENIX Association, 2013.

[18] M. O. Rabin, “Fingerprinting by random polynomials”, Center for Research in Computing Techn., Aiken Computation Laboratory, 1981.

[19] http://www.serve.net/buz/hash.adt/java.000.html

[20] M. Datar, N. Immorlica, P. Indyk & V. S.Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions”, Computational geometry, ACM, 2004: 253-262.

[21] G. S. Manku, A. Jain, A. D. Sarma, “Detecting near-duplicates for web crawling”, World Wide Web, ACM, 2007: 141-150.

978-1-4673-7619-8/15/$31.00 ©2015 IEEE