Frequency Based Chunking for Data De-Duplication

Guanlin Lu, Yu Jin, and David H.C. Du
Department of Computer Science and Engineering
University of Minnesota, Twin-Cities
Minneapolis, Minnesota, USA
(lv, yjin, & du)@cs.umn.edu

Abstract— **A predominant portion of Internet services, like content delivery networks, news broadcasting, blogs sharing and social networks, etc., is data centric. A significant amount of new data is generated by these services each day. To efficiently store and maintain backups for such data is a challenging task for current data storage systems. Chunking based deduplication (dedup) methods are widely used to eliminate redundant data and hence reduce the required total storage space. In this paper, we propose a novel Frequency Based Chunking (FBC) algorithm. Unlike the most popular Content-Defined Chunking (CDC) algorithm which divides the data stream randomly according to the content, FBC explicitly utilizes the chunk frequency information in the data stream to enhance the data deduplication gain especially when the metadata overhead is taken into consideration. The FBC algorithm consists of two components, a statistical chunk frequency estimation algorithm for identifying the globally appeared frequent chunks, and a two-stage chunking algorithm which uses these chunk frequencies to obtain a better chunking result. To evaluate the effectiveness of the proposed FBC algorithm, we conducted extensive experiments on heterogeneous datasets. In all experiments, the FBC algorithm persistently outperforms the CDC algorithm in terms of achieving a better dedup gain or producing much less number of chunks. Particularly, our experiments show that FBC produces 2.5 ~ 4 times less number of chunks than that of a baseline CDC which achieving the same Duplicate Elimination Ratio (DER). Another benefit of FBC over CDC is that the FBC with average chunk size greater than or equal to that of CDC achieves up to 50% higher DER than that of a CDC algorithm.**

Keywords- Data Deduplication; Frequency based Chunking; Content-defined Chunking; Statistical Chunk Frequency Estimation; Bloom Filter

I. INTRODUCTION

Today, a predominant portion of Internet services, like content delivery networks, news broadcasting, blogs sharing and social networks, etc., is data centric. A significant amount of new data is generated by these services each day. To efficiently store and maintain backups for such data is a challenging problem for current data storage systems.

In comparison to the compression technique which does not support fast retrieval and modification of a specific data segment, chunking based data deduplication (dedup) is becoming a prevailing technology to reduce the space requirement for both primary file systems and data backups. In addition, it is well known that for certain backup datasets, dedup technique could achieve a much higher dataset size reduction ratio comparing to compression techniques such as gzip (E.g. the size folding ratio of chunking based dedup to that of gzip-only is 15:2. [25]). The basic idea for chunking based data dedup methods is to divide the target data stream into a number of chunks, the length of which could be either fixed (Fix-Size Chunking, FSC) or variable (Content-Defined Chunking, CDC). Among these resulted chunks, only one copy of each unique chunk is stored and hence the required total storage space is reduced.

The popular baseline CDC algorithm employs a deterministic distinct sampling technique to determine the chunk boundaries based on the data content. Specifically, CDC algorithm adopts a fix-length sliding window to move onwards the data stream byte by byte. If a data segment covered by the sliding window sampled by the CDC algorithm satisfies certain condition, it is marked as a chunk cut-point. The data segment between the current cut-point and its preceding cut-point forms a resultant chunk.

In this paper as a first step to assess the duplicate elimination performance for chunking based dedup algorithms, we advance the notion of *data dedup gain*. This metric takes into account the metadata cost for a chunking algorithm and hence enable us to compare different chunking algorithms in a more realistic manner. Based on the data dedup gain metric, we analyze the performance of the CDC algorithm in this paper. Though being scalable and efficient, content defined chunking is essentially a random chunking algorithm which does not guarantee the appeared frequency of the resultant chunks and hence may not be optimal for data dedup purpose. For example, in order to reduce the storage space (i.e., to increase the dedup gain), the only option for the CDC algorithm is to reduce the average chunk size. That is, to increase the distinct sampling rate to select more cut-points. However, when the average chunk size is below a certain value, the gain in the reduction of redundant chunks is diminished by the increase of the metadata cost.

Being aware of the problems in the CDC algorithm, in this paper, we propose a novel Frequency Based Chunking (FBC) algorithm. The FBC algorithm is a two-stage algorithm. At the first stage, the FBC applies the CDC algorithm to obtain coarse-grained chunks, i.e., the resulted chunks are of a larger average chunk size than desired. At the second stage, FBC identifies the globally appeared frequent data segments from each CDC chunk and used these data segments to further selectively divide the CDC chunks into fine-grained chunks. In this way, the FBC chunks consist of two parts, the globally appeared frequent chunks and the chunks formed by the remaining portions of the CDC chunks. Of course, an entire CDC chunk may not be further partitioned if it does not contain any frequently appeared data segments. Intuitively, the globally

appeared frequent chunks are highly redundant and the redundancy of the remaining chunks is close to that of the original CDC chunks. Therefore, the FBC algorithm is expected to have a better data dedup performance than the CDC algorithm.

To obtain the globally appeared frequent chunks used by the FBC algorithm, direct counting of the chunk frequencies is not feasible due to the typically huge volume of the possible data streams. In this paper, we employ a statistical chunk frequency estimation algorithm for this task. Based on the observation of the chunk frequency distribution in the data stream, That is, high frequency chunks only account for a small portion of all the data chunks. The proposed algorithm utilizes a two-step filtering (a pre-filtering and a parallel filtering) to eliminate the low frequency chunks and hence to reserve resource to accurately identify the high frequent chunks. Due to the powerful filtering mechanism, the proposed statistical chunk frequency estimation algorithm only requires a few megabytes to identify all the high frequency chunks with a good precision, even though the threshold for defining the high frequency chunks can be moderate (e.g., 15). Therefore, the performance of the proposed algorithm is beyond the capability of most state-of-the-art heavy hitter detection algorithms.

In order to validate the proposed FBC chunking algorithm, we utilize data streams from heterogeneous sources. Experimental results show that the FBC algorithm consistently outperforms the CDC algorithm in terms of the dedup gain or in terms of the number of chunks produced. Particularly, our experiments verify that FBC produces 2.5 ~ 4 times less number of chunks than that of a baseline CDC does when achieving the same DER. FBC also outperforms CDC in terms of DER. With the average chunk size no less than that of CDC, FBC achieves up to 50% higher DER.

Our contribution in this paper can be summarized as follows:

1) We advance the notion of *data dedup gain* to compare different chunking algorithms in a realistic scenario. The metric takes both the gain in data redundancy reduction and the metadata overhead into consideration.

2) We propose a novel frequency based chunking algorithm, which explicitly considers the frequency information of data segments during the chunking process. To the best of our knowledge, our work is the first one to incorporate the frequency knowledge to the chunking process.

3) We design a statistical chunk frequency estimation algorithm. With small memory footprint, the algorithm is able to identify data chunks above a moderate frequency in a stream setting. This algorithm itself can be applied to other application domains as a streaming algorithm for cardinality estimation purpose.

4) We conduct extensive experiments to evaluate the proposed FBC algorithm using heterogeneous datasets, and the FBC algorithm persistently outperforms the widely used CDC algorithm.

The rest of the paper is organized as follows. In the next section, we present an overview of the related work. Section III provides the motivation of proposing a FBC algorithm. The proposed FBC algorithm is described in Section IV with an extensive analysis presented for each component. Section V presents experimental results on several different types of empirical datasets. Finally, we describe our future work and some conclusions in Section VI.

II. RELATED WORK

As in the domain of data deduplication, chunking algorithms can be mainly categorized into two classes: Fix-Size Chunking (FSC) and Content-Defined Chunking (CDC). The simplest and fastest approach is FSC which breaks the input stream into fix-size chunks. FSC is used by rsync [1]. A major problem with FSC is that editing (e.g. insert/delete) even a single byte in a file will shift all chunk boundaries beyond that edit point and result in a new version of the file having very few repeated chunks. A storage system Venti [2] also adopts FSC chunking for its simplicity. CDC derives chunks of variable size and addresses boundary-shifting problem by posing each chunk boundary only depending on the local data content, not the offset in the stream. CDC will enclose the local edits to a limited number of chunks. Both FSC and CDC help data deduplication systems to achieve duplication elimination by identifying repeated chunks. For any repeated chunks identified, only one copy will be stored in the system.

CDC was first used to reduce the network traffic required for remote file synchronization. Spring et al. [3] adopts Border's [4] approach to devise the very first chunking algorithm. It aims at identifying redundant network traffic. Muthitacharoen et al. [5] presented a CDC based file system called LBFS which extends the chunking approach to eliminate data redundancy in low bandwidth networked file systems. You et al. [6] adopts CDC algorithm to reduce the data redundancy in an archival storage system. Some improvements to reduce the variation of chunk sizes for CDC algorithm are discussed in TTTD [7]. Recent research TAPER [8] and REBL [9] demonstrate how to combine CDC and delta encoding [10] for directory synchronization. Both schemes adopt a multi-tier protocol which uses CDC and delta encoding for multi-level redundancy detection. In fact, resemblance detection [4, 11] combined with delta encoding [12] is usually a more aggressive compression approach. However, finding similar or near identical files [13] is not an easy task. Comparison results on delta encoding and chunking are provided in [14, 15].

The concept of identifying high-frequency identities among large population is discussed in the area of Internet traffic analysis. Manku et al. [16] proposed an inferential algorithm to find and count frequent item in the stream. However, their work is very general and do not consider particular frequency distribution of counted items. The idea of using parallel stream filter to identify high cardinality Internet hosts among existing host pairs is recently proposed in [17]. In this paper, we adopt the concept of parallel stream filter and design a special parallel bloom filter [18] based filter to identify high-frequency chunks and to obtain their frequency estimates. We are also aware of that bloom Filter was used in [19] and [20] to memorize data chunks that have been observed. However, they use only one bloom Filter and thus can only tell whether a chunk is observed or not. Metadata overhead is a big concern as in chunking

based data deduplication [21]. Kruus et al. [20] presents a work similar to ours. Provided chunk existence knowledge, they propose a two-stage chunking algorithm that re-chunks transitional and non-duplicated big CDC chunks into small CDC chunks. The main contribution of their work is to be able to reduce the number of chunks significantly while attaining as the same duplicate elimination ratio as a baseline CDC algorithm. Alternative work to reduce metadata overhead is to perform local duplicate elimination within a relatively large cluster of data. Extreme binning [22] detects file level content similarity by a representative hash value, and shows to be able achieving close to original CDC performance on data sets with no locality among consecutive files. Another recent work [23] presents a sparse indexing technique to detect similar large segments within a stream.

III. MOTIVATION

In this section, we define the *data dedup gain* as a general metric to measure the effectiveness of a dedup algorithm. Based on this metric, we point out the problems in the popular CDC algorithm which hinders the algorithm from achieving a better dedup gain. This analysis motivates the proposed FBC algorithm in Section IV.

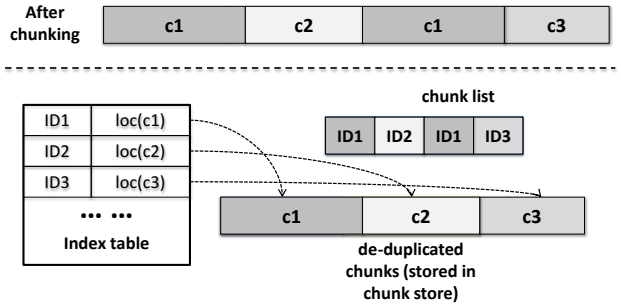

Figure 3.1. Data Structures Related to Chunking Dedup

A. Data Dedup Gain

Let s denote the byte stream that we want to apply the dedup algorithm for. Fig. 3.1 shows the results of data dedup using chunking-based algorithm. After a specific chunking algorithm (e.g., CDC), s is divided into n non-overlapping data chunks $s = \{c_1, c_2, c_3, \dots, c_n\}$. We then assign each unique chunk an ID, denoted as $ID(c_i)$. The data stream s is mapped to a chunk list $\{ID(c_1), ID(c_2), \dots, ID(c_n)\}$, which is used to support file retrieval from stored chunks. Practically, SHA-1 hash algorithm is used to produce an ID for a chunk based on its content. We store the de-duplicated chunks in a chunk store and use an index table to map chunk IDs to the stored chunks. Apparently, the benefit from dedup happens when we only store one copy called unique chuck of the duplicated chunks (e.g., chunk c_1 in Fig. 3.1). However, we also need extra storage space for the index table and chunk list, which we refer to as the metadata overhead. Metadata is a system dependent factor that affects the duplicate elimination result. The final gain from the chunking-based dedup algorithm, which we call the *data dedup gain*, is the difference between the amount of duplicated chunks removed and the cost of the metadata.

Denote the set of distinct chunks among these n chunks to be $D(s) = distinct\{c_1, c_2, \dots, c_n\}$. Let the set size to m, i.e. $m = |D(s)|$. We now formally define the data dedup gain on data stream s using a specific chunking based dedup algorithm A as:

$$gain_A(s) = folding_{factor\ (s)} - metadata(s) \quad (3.1)$$

$$folding_{factor\ (s)} = \sum_{c_i \in distinct\ (s)} |c_i| \cdot (f(c_i) - 1) \quad (3.2)$$

The first term in (3.1) is called the folding factor term which stands for the gain by removing duplicated chunks. Since for each unique chunk we only need to store one copy in the chunk store and remove the rest of the copies, the folding factor is calculated as the number of repeated copies of each unique chunk ($f(c_i) - 1$) multiplies the length of that chunk. $f(c_i)$ is the number of times a unique chunk c_i appears in the stream. In this paper we denote $f(c_i)$ the frequency of chunk c_i.

It could be clearly seen that the product of chunk frequency and chunk size determines how much a distinct chunk contributes to $gain_A(s)$. In the extreme case where $f(c_j) = 1$, then chunk j contributes nothing to $gain_A(s)$.

The second term in (3.1) is called metadata overhead, $metadata(s)$ can be expressed as $|\text{index}| + |\text{chunk lists}|$, which is the sum of the length of on-disk chunk index plus the size of chunk lists. The Index table requires at least $m \cdot log_2 m$, where $log_2 m$ is the minimal length of a chunk pointer that points to each unique chunk. Then we rewrite (3.1) as follows:

$$\begin{aligned} gain_A(s) &= \sum_{c_i \in distinct\ (s)} |c_i| \cdot (f(c_i) - 1) \\ &\quad - (|\text{index}| + |\text{chunk lists}|) \\ &= \sum_{c_i \in distinct\ (s)} |c_i| \cdot (f(c_i) - 1) - \left(\frac{1}{8} m log_2 m + 20n\right) \end{aligned} \quad (3.3)$$

We assume 20-byte SHA-1 hash is used to represent a chunk ID and we divide $mlog_2 m$ by 8 to convert bit unit to byte unit. It is worth noting that the index table size we considered here is a theoretically lower-bound, so the corresponding $gain_A(s)$ we calculate is an upper-bound of deduplication gain. (i.e., any real data deduplication system cannot gain more than $gain_A(s)$.) Our data dedup gain definition assumes a chunking based data deduplication model on top of the file system layer. Thus, the metadata overhead referred to in this paper should not be confused with file system metadata overhead such as inode, etc.

B. Discussion on the Baseline CDC algorithm

From the formula for the data dedup gain, given a fixed number of chunks, a good chunking algorithm is expected to generate more (and longer) frequent chunks to maximize the dedup gain. Being aware of this, we study the baseline CDC chunking algorithm.

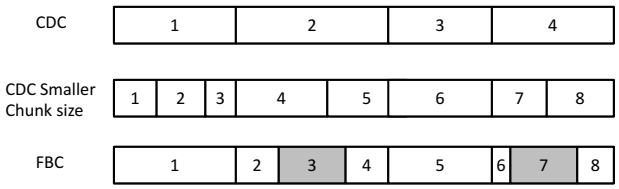

Figure. 3.2. Illustration of Chunks Produced by CDC and FBC Algorithms

Content defined chunking algorithm chops the data stream into chunks based on its local content satisfying a certain statistical condition and this statistical condition behaves like a random sampling process. For example, Fig. 3.2 (top) shows the data stream after the CDC chunking algorithm. In order to increase the dedup gain, the only way is to reduce the chunk size in CDC so that we can find more repeated (also smaller) chunks in the stream. Fig. 3.2 (middle) shows the results of a fine grained CDC after halving the average chunk size. However, because the random manner of the CDC partitioning, such an algorithm cannot guarantee the frequency of the resulted chunks. Instead, these small chunks more likely have low frequencies. As a consequence, the increase in the metadata cost may diminish the gain in the removal of redundant chunks.

However, let us assume that we have the knowledge of appearing frequency associated with each chunk (of a fixed length) in the data stream. Then, we can design a much better chunking scheme. As illustrated in Fig. 3.2 (bottom), after a coarse grained CDC algorithm, we perform a second level (fine-grained) chunking by identifying the chunks with a frequency greater than a predefined threshold. The original Chunks 2 and 4 are divided into three smaller chunks, where the new Chunks 3 and 7 are of high frequencies. These frequent chunks provide us a better the dedup gain. In addition, as we shall see later, after extracting these frequent chunks, the remaining ones (smaller segments) have a similar frequency distribution as the one in the fine grained CDC method.

Such a chunking method, which we refer to as the Frequency Based Chunking scheme (FBC), though sounds appealing, it makes a strong assumption on the knowledge of the individual chunk frequency. To solve this problem, we propose a statistical frequency estimation algorithm to identify the chunks of high frequencies with a low computation overhead and memory consumption. In the following section, we introduce the proposed FBC algorithm in detail.

IV. FBC ALGORITHM

In this section, we first present an overview of the proposed FBC algorithm. We then introduce the statistical frequency algorithm for identifying the high frequency chunks, as well as the counting and bias correction. At the end of the section, we discuss our two-stage chunking algorithm.

A. Overview of Frequency-based Chunking Algorithm

In this paper, we describe the frequency-based chunking method. That is, a hybrid chunking algorithm to divide the byte stream based on chunk frequency. Conceptually, the FBC algorithm consists of two steps: *chunk frequency estimation* and *chunking*. At the first step, FBC identifies fixed-size chunk candidates with high frequency. The chunking step further consists of a coarse-grained CDC chunking and a second-level (fine-grained) frequency based chunking. During coarse-grained chunking, we apply the CDC algorithm to chop the byte stream into large-size data chunks. During the fine-grained chunking, FBC examines each CDC coarse-grained chunk to search for the frequent fix-size chunk candidates (based on the chunk frequency estimation results) and uses these frequent candidates to further break the CDC chunks. If a CDC chunk contains no frequent chunk candidates, FBC outputs it without re-chunking it. It worth noting that coarse-grained CDC chunking step is essentially performing "virtual chunking" in the sense that it only marks the chunk cut-points in the byte stream. The real cut and SHA1 calculation occurs only when FBC outputs a chunk. In the following, we discuss each step in detail.

B. Chunk Frequency Estimation

Due to the large size of the data stream, directly counting the frequency of each chunk candidate is not feasible. For example, assume we use a sliding window to retrieve all the 4KB length chunk candidates in a 1 GB length data stream (we shift the sliding window one byte at a time to obtain a new chunk candidate). Assume the average chunk candidate frequency is 4, there are around 268 million distinct chunk candidates (1 GB / 4). Direct counting the frequencies for such a large number of chunk candidates is apparently not scalable. However, one may notice that the FBC algorithm only requires the knowledge of high-frequency chunk candidates. Fig. 4.1 shows the log-log plot for the chunk frequency distribution. These chunks are of 1KB size from the dataset *Nytimes* (see Section V for the details of the dataset). One interesting observation is that a majority of the chunk candidates are of low frequencies. For example, more than 80% of the chunks only appear once, and only less than 5% of the chunks appear more than 10 times in the stream. Similar observations are made for other chunk sizes.

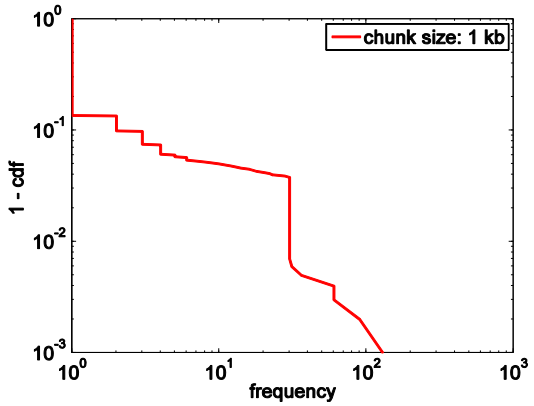

Figure 4.1. Log-log Plot for the Chunk Frequency Distribution

These infrequent chunks are of little interest to us, however, they consumes most of the memory during a direct counting process due to the large number of such chunks. To address this problem, in this paper, we propose a statistical frequency estimation algorithm to identify the high frequency chunks. The basic idea is to apply a filtering process to eliminate as many low-frequency chunks as possible and hence reserve resource to obtain an accurate frequency estimate for the high-frequency chunks.

The chunk frequency estimation algorithm consists of three components, prefiltering, parallel filtering and frequency counting. The first two steps aim at reducing the low frequency chunk candidates as many as possible, while the last step counts the occurrence of high-frequency chunk candidates. Fig. 4.2 displays a schematic view of the frequency estimation algorithm. We next discuss the implementation of these components.

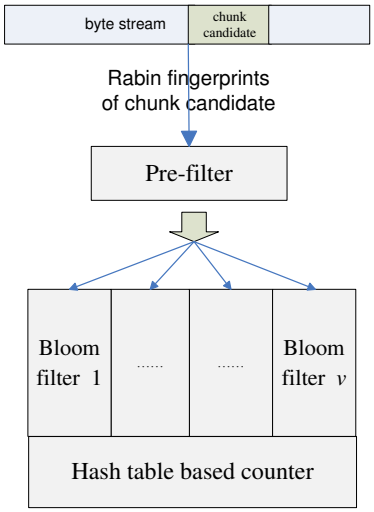

Figure 4.2. Overview of Frequency Estimation Algorithm

1) Prefiltering

We use a sliding window to extract all the fixed-size chunk candidates from the target byte stream. Before we start filtering low frequency chunk candidates with the parallel filter, we apply a pre-filtering step to eliminate the overlapping chunks from the sliding window. For example, suppose there is a data segment with length 2 KB appeared 100 times through the byte stream. Assume we use 1KB as the sliding window size, then that 2KB segment will result in 1,024 overlapping 1KB sized chunk candidates with each one having a frequency of 100. To avoid deriving overlapped chunk candidates, we apply a pre-filtering process, which is a modulo-based distinct sampler with a sampling rate r_0. A chunk has to pass the pre-filtering process first before entering the next parallel filtering step. Such a prefiltering step only requires one XOR operation for filtering decision per chunk candidate it observes. Essentially, r_0 controls the density of sampling chunk candidates in the stream. (For example, r_0 = 1/32 makes a chunk candidate pass if its Rabin's fingerprint [24] value modulo 32 = 1.)

2) Parallel filtering

The parallel filtering is motivated by the idea of dependence filtering in [17]. Before frequency estimation starts, the set of v bloom filters is empty. For each survived chunk candidate from pre-filtering, we first check the appearance of this chunk in all bloom filters. If we find this chunk in each of the bloom filters, we let it pass through the parallel filter and start to count its frequency. Otherwise, we record the chunk in one of the v bloom filters randomly. Apparently, without considering the false positive caused by bloom filters, only chunk candidates with frequency greater than v can possibly pass the parallel filtering process. Due to the efficiency of the bloom filter data structure that only consumes a small amount of memory, in this way we can filter out majority of the low frequency chunk candidates with small amount of resource.

In particular, let X be the number of occurrence before a chunk passes the parallel filtering, the expected number of X can be calculated as follows:

$$E(X) = v\sum_{i=1}^{v}\frac{1}{i} \qquad (4.1)$$

For example, if we use three bloom filters to build the parallel filter ($v = 3$), a chunk is expected to have a frequency of 5.5 in order to pass the parallel filtering. When the number of bloom filters is increased to 5, $E(X) = 11.4$. In general, using more bloom filters for the parallel filtering step can help reduce more low frequency chunks. However, we need to strike a balance between the amount of low frequency chunks reduced and the resource consumed by the bloom filters. In our experiments, we choose $v = 3$, which works best in practice. We present our filtering and counting algorithm in Algorithm 1. Notice the input chunk candidate s_i represents the ones survived pre-filter. Also, we set number of bloom filters to be 3.

There is a rich literature for detecting high frequency items in a data stream. However, in our problem, the threshold for high frequency chunks is usually as low as 15. The classical heavy hitter detection algorithms do not scale under such a low threshold. One salient feature of the proposed parallel filtering algorithm is that the parallel stream filter is able to capture the entities with not-so-high frequency. This can be achieved by the parallel filtering since it is able to eliminate a majority of the lowest frequency chunks.

Algorithm 1 Parallel filtering and frequency counting

```
1: Parameters: Chunk stream s = {si},θ;
2: Output: Chunks with frequency greater than θ.
3: for each si ∈s do
4:    if All 3 Bloom filters contain si then
5:       if Hash table does not contain si then
6:          Add si to the hash table and set freq(si)= 6;
7:       end if
8:       freq(si) := freq(si) + 1;
9:    else
10:      Randomly add si to one of the 3 Bloom filters
11:   end if
12: end for
13: //Output frequent chunks
14: for each si in hash table do
15:   if freq(si) > θ then
16:      Output si
17:   end if
18: end for
```

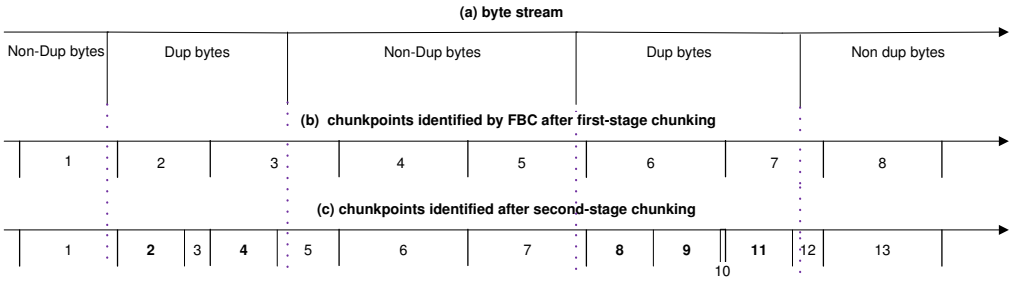

Figure 4.3 FBC Chunking Steps

C. Counting and Bias Correlation

After parallel filtering, for each survived chunk candidate (i.e., this chunk candidate has appeared in all bloom filters), we start counting its frequency. The counting process can be implemented using a simple hash table, where the keys correspond to the chunks and the values represent the frequency of the chunks.

Notice that our counting process does not start until a chunk candidate appears in all bloom filters. Hence a counting bias exists because a number of copies for a particular chunk are lost during the parallel filtering process. This counting bias can be estimated as $E(X)$. In other words, after estimating the frequency for a particular chunk, we need to add $E(X)$ to the frequency to correct the estimation bias.

D. Two-stage Chunking

The concept of combining CDC algorithm as its first-stage chunking makes the FBC algorithm a hybrid two-stage chunking algorithm. It combines the coarse-grained CDC chunking with fine-grained frequency based chunking.

Fig. 4.3 illustrates the operation on a simple input byte stream. Fig. 4.3(a) represents the coarse-grained CDC chunks 4.3(b) are further examined for fine-grained fix-size frequent Chunks 2, 4, 8, 9, and 11 in 4.3(c). Coarse CDC Chunks 4,5, and 8 in Fig. 4.3(b) without containing any frequent fix-size chunks are output as individual chunks (Chunks 6, 7, and 13 in 4.3(c)). Sometime small but infrequent chunk may also be produced. For example, Chunk 6 in Fig. 4.3(b) gets rechunked into frequent Chunks 8 and 9 in Fig. 4.3(c) while the remaining part of Chunk 6 in Fig. 4.3(b) becomes Chunk 10 in Fig. 4.3(c). However, through our experiments, we found that the impairing impact of produced small chunks (e.g. Chunk 10 in Fig. 4.3(c)) to the total number of chunks is easily offset by the generation of large size CDC chunks.

In this paper, we consider a fix-sized chunk to be frequent if its frequency no less than a pre-defined threshold θ which is determined based on domain knowledge on the datasets including data set content, data set type, how many backups the data set contains (if it is a backup dataset), etc. The choice of θ reflects how aggressive the re-chunking should occur. For example, the larger θ is, the smaller portion of CDC coarse grained chunks will be further re-chunked. On the other hand, each re-chunking brings in extra metadata overhead with longer chunk list and possibly larger index table.

V. EXPERIMENTAL RESULTS

In this section, we present the experimental results. We first introduce the datasets used for evaluation and discuss the experimental settings. We then compare the dedup performance of the proposed FBC algorithm with the CDC algorithm in terms of dedup gain and point out the major factors that contribute to the difference between these two methods.

A. Evaluation Datasets

We use three empirical datasets to evaluate the effectiveness and efficiency of proposed FBC algorithm. Two datasets, *Linux* and *Nytimes*, represent the situation when the data streams exhibit high degree of redundancy; while the *Mailbox* dataset is characterized with low detectable redundancy.

Nytimes is web content dataset, corresponding to 30 days of the snapshots of the New York Times website (*www.nytimes.com*) from June 11th to July 10th, 2009 which includes a collection of text, images, binaries and video clips. The data set is collected by crawling software *wget*, which is configured to retrieve the web pages from the website recursively for a maximum depth of two. We store the web pages in 30 tar balls, one for each day. The total data stream size is 0.92 GB. The dataset *Linux* contains the kernel source files of 10 consecutive versions of Linux operating system, from Linux 2.6.30.1 to Linux 2.6.30.10. The size of *Linux* is 3.37 GB. *Mailbox* is a company mail repository used by a group of software engineers. We treat it as a byte stream that has the size of 0.48 GB.

B. Experiment Enviornment

We have implemented CDC and FBC algorithm with GNU C++. We also implemented the parallel bloom filter using Python 2.6. The experiments are carried out on a standard *Linux* build 2.6 SMP x86_64 platforms. The machine has two 2.0GHz cores. The memory size is 2GB.

C. Evaluation Criteria

We focus on evaluating the performance of the proposed algorithm at chunk level, thus we do not compress any resultant chunks after chunking. We apply a widely adopted metric Duplicate Elimination Ratio (DER) for evaluation purpose. DER is calculated as the input stream length divided by the total length of all output chunks after dedup while no metadata overhead is considered.

$$DER = \frac{input\ stream\ length}{total\ sum\ of\ the\ sizes\ of\ chunks\ produced}$$

As we have pointed out in the previous sections, in practice, metadata overhead is a key factor that affects the duplicate elimination result. A special feature of the proposed FBC algorithm is that it is designed explicitly to reduce the metadata cost. To demonstrate this, we define a new version of the DER metric by taking into account the metadata overhead.

Recall that from (3.3), the metric $gain_A(s)$ represents the amount of storage space saved by applying the dedup algorithm A on data stream s after subtracting the cost of the metadata. Denote the length of stream s to be $|s|$, therefore, we define the new DER metric DER_{meta} as:

$$DER_{meta} = \frac{|s|}{|s| - gain_A(s)} = \frac{|s|}{\sum_{c_i \in distinct\ (s)} |c_i| + (|\text{index}| + |\text{chunk lists}|)}$$

Another important metric we used is the *Average Chunk Size* (*ACS*), which is defined as the data stream size divided by the total number of chunks after the chunking algorithm. Clearly, the total number of produced chunks is inversely proportional to ACS and hence the storage overhead of metadata that is required to describe the chunks is also inversely proportional to ACS.

We evaluate FBC in both DER and ACS factors. That is, we want to see how much advantage FBC could gain when comparing with the baseline CDC algorithms, in terms of DER and ACS.

Another reason for considering ACS as a metric is that it is directly related to the operation cost. Fewer chunks mean less disk accesses, which further imply less I/O transactions. In addition, under the networked environment, fewer chunks lead to less network queries and possibly less network transmissions. At last, less file reconstruction time should be expected for a bigger ACS. Given a specific DER, we hope to maximize the average chunk size.

We realize such comparison may be a little unfair because FBC requires an extra pass of the byte stream. However, we believe such a cost is deserved. First, FBC takes a highly efficient frequency estimation algorithm which only requires very small memory footprint to get the desired accuracy. In addition, the estimation speed is comparable to chunking process. Since at the second-stage the chunking algorithm requires examining at least a portion of chunks produced at the first-stage, we plan to further reduce the frequency estimation to only a smaller portion of the stream in our future work.

D. Tuning the Parameters

We next discuss the parameter settings for our FBC algorithm in the following experiments. We group the parameters used at different stages.

Prefiltering step uses sampling rate r_0 determines the sampling density. By extensive testing on several datasets, we choose r_0 to be modulo 32 (i.e. Prefilter lets a chunk candidate to pass only if its Rabin's fingerprint modulo 32 equals 1). We argue that this value strikes a balance between identifiable duplicated size and degree of overlapping for all identified chunks. The sliding window size for the prefiltering step is set according to the specific datasets which we will discuss in the subsequent subsections.

Parallel filtering step takes the parameters such as the number of bloom filters v, the bloom filter size and the chunk candidate frequency threshold θ. As we have mentioned, we set $v = 3$, the memory consumption for the bloom filters used by the three datasets is 2.4MB respectively. The calculation for the bloom filter size is based on the optimal bloom filter memory allocation discussed in [17] and the specific chunk frequency distribution of the datasets. As a future work, we may use bloom filters with dynamic memory allocation to accommodate for different data streams. We note that when the data is highly redundant, most chunks will pass the parallel filtering. One solution in this case is to apply a random sampling with rate r_1 before a chunk is inserted into one of the bloom filters. In this way, the real frequency is calculated as the frequency estimate from the counting process divided by r_1. The parameter θ is set to 5 in the following experiments.

Two-stage counting: CDC algorithm at the first stage takes parameters including the minimum/maximum chunk sizes and the expected chunk size. From our extensive experiments, we observed that when setting minimum/maximum chunk size to 1 Byte/100 KB, and varying the expected chunk sizes among 0.25KB, 0.5KB, 1KB, 2KB, 4KB and 8KB under each run, the resulting average chunk size is very close to the expected chunk size. Further examination on the chunk size distribution demonstrates that none of chunk distributions is pathological, i.e., a significant portion of chunks with size close to 1. Hence we apply 1Byte/100KB minimum/maximum chunk size setting to both CDC and FBC in all our experiments. The CDC algorithm employs its own 64-byte length sliding window to generate chunk cut-points.

Extensive experiments on multiple data sets suggest that when the ratio of average CDC-type chunk size to frequent chunk size equals 16, it yields the best result. So we adopt this ratio in all the following experiments.

E. Experimental Results

1) Chunk candidate frequency estimation

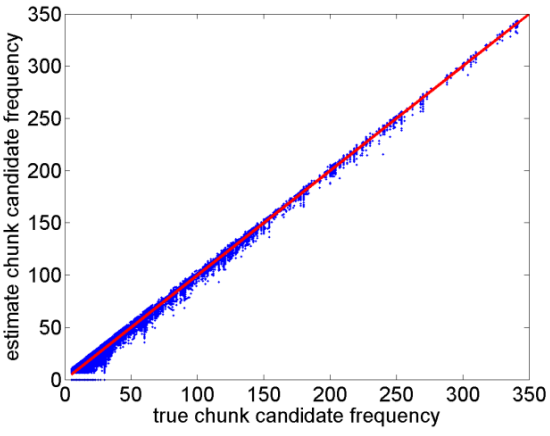

Figure 5.1. Accuracy of Chunk Frequency Estimation

Major concerns regarding to our candidate frequency estimation are its accuracy and memory consumption. We examine the parameter settings for our parallel bloom filter through multiple runs on the *Nytimes* dataset. Fig. 5.1 plots the estimated chunk candidate frequency against true chunk candidate frequency under parameter setting $v = 3$ and bloom filter size equals 0.8MB. The chunk candidate frequency threshold $\theta = 5$. We could clearly see the estimation result is closely around the true frequency values, even at very small filter size (i.e. $3 \times 0.8 = 2.4$MB). Further examination on all other datasets demonstrates similar results.

2) Correlation between frequent-chunk size and total saved size

It is interesting to see the correlation between frequent-chunk size and its impact on the total saved size. We assume $\theta = 5$ (i.e. all chunks with frequency no less than 5 are considered to be frequent.) and we fixed CDC-type expected chunk size used in FBC algorithm to be 8KB. Fig. 5.2 plots the saved size vs. frequent-chunk size used on *Nytimes* data set. Solid curve indicates the total size for all detected duplicated frequent chunk candidates (overlapping chunk candidates are removed) while dashed curve indicates the FBC result. The former is essentially an indicator of saved size that FBC algorithm achieves, since it demonstrates how much duplication could be discovered with corresponding frequent chunk size.

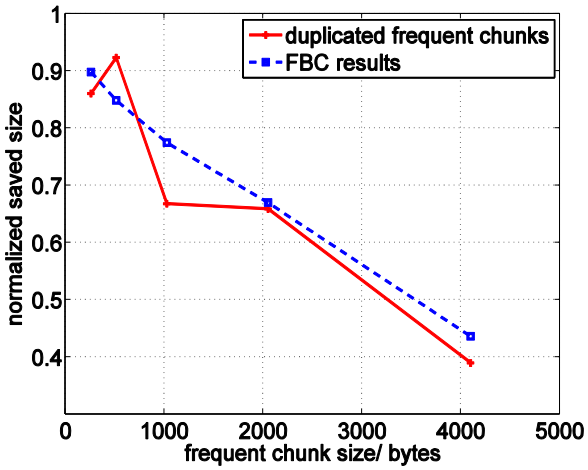

Figure 5.2. Frequent Chunk Sizes vs. FBC Result

We do see the positive correlation between solid and dashed curves, validating that FBC does exploit the gain of frequent chunks. Furthermore, it can be observed that as frequent chunk size doubles from 128bytes to 4KB, the saved size decreases for both curves, which means small frequent-chunk size could result in more duplication being detected.

3) Comparing results for FBC algorithm and CDC algorithm

We compare FBC and CDC with respect to DER and average chunk size on *Linux* and *Nytimes* datasets. Fig.5.3 shows the results on *Linux* dataset, which summarizes five runs of each algorithm. Fig. 5.4 shows a similar result on Nytimes dataset, summarizing six runs of each algorithm.

For each test, we collect two DERs, with respect to whether metadata overhead is considered or not. Solid and dashed curves (top 2 curves) represent the results for FBC while dash-dotted and dotted curves (bottom 2 curves) represent the results of CDC. In addition, for either FBC or CDC, the curve with upward-pointing triangle marker represents experiment results without considering metadata overhead while the curve with downward-pointing triangle marker represents results with metadata overhead considered.

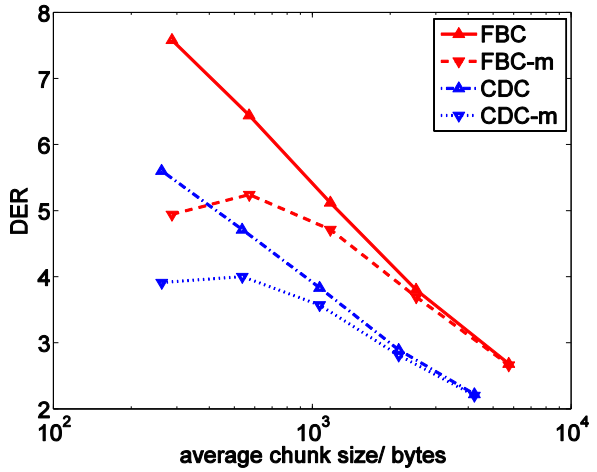

Figure 5.3. Baseline CDC and FBC are compared under a wide range of chunk sizes on *Linux*. We run CDC with expected chunk sizes doubled from 0.25KB up to 4KB; we run FBC with fix-size frequent chunk sizes doubled from 0.5KB to 8KB correspondingly. This is to assure both algorithms produce the same number of chunks in each comparison.

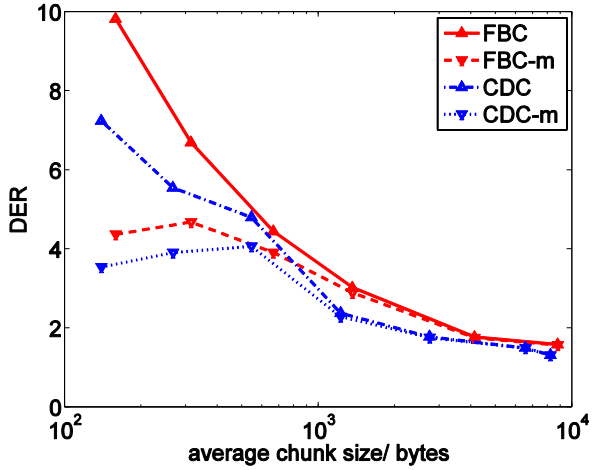

Figure 5.4. Baseline CDC and FBC are compared under a wide range of chunk sizes on Nytimes. We run CDC with expected chunk sizes doubled from 0.125KB up to 4KB; correspondingly we run FBC with fix-size frequent chunk sizes doubled from 0.25KB to 8KB. This is to assure that both algorithms produce the same number of chunks in each comparison.

Several important observations could be made for both figures:

a) When the average chunk size grows beyond 2KB, FBC achieves almost 1.5 times DER than that of CDC achieves on the same average chunk size and this gain even increases further as larger chunk sizes are used. Furthermore, since metadata overhead tends to be negligible as average

chunk size grows beyond 2KB, we further conclude that our FBC algorithm is expected to perform better when large chunks are desired. On *Nytimes* dataset, FBC outperforms CDC algorithm as well, but the difference is less than that of *Linux* dataset.

b) For both FBC and CDC algorithms, as the average chunk size grows, the result considering metadata overhead is getting close to the result without considering it. This observation essentially justifies the preference of large chunks over small chunks for many chunking based deduplication systems.

c) If we do not consider metadata overhead, DER increases monotonoically as we choose smaller chunks size for both FBC and CDC. However, when considering the metadata overhead, FBC performane peaks at frequent chunk size 0.5KB while CDC perfomance peaks at expected chunk size 0.5KB on both datasets. This means we cannot arbitrarily reduce the chunk size in CDC in order to achieve better DER. When the chunk size is too small, the loss in the metadata cost outgrows the gain in the increase of redudant chunks.

d) If we consider the the benefit of FBC in terms of the number of chunks produced, then a even more significant result can be obtain. This is easy to see from Figs 5.3 and 5.4. Please note that the x-axis is in log scale. For example, from Fig 5.3 we could see, when taking into account of metadata overhead, the FBC with an average chunk size closed to 2KB has the same DER 4.0 as CDC with an average chunk size 0.5KB, which is 4 times improvement on chunk size. This result applies to Nytimes dataset as well. From Fig 5.4 we can see, regardless considering metadata overhead or not, to achieve the same DER 3.9, the FBC produces chunks with an average chunk size 663 bytes; while CDC produces the ones with an average chunk size 266 bytes. Hence FBC has 2.5 times improvement on chunk size.

We also examine the result on Mailbox dataset. However, we find both CDC and FBC perform equally bad. For example, even the chosen expected chunk size to be as small as 0.25KB, the DER for CDC and FBC algorithm is only 1.5 which is much worse than 7.8 and 10 achieved on *Linux* and *Nytimes* datasets with the same expected average chunk size. Investigation on the Mailbox dataset shows that this mailbox mainly consists of very short messages that are carbon-copies from different engineers. The duplicated data pieces are too small to be good candidates for data deduplication. Such datasets essentially reveal the limitation of all chunking based algorithms which requires other types of data dedup methods or data compression approaches.

VI. CONCLUSION

In this paper, we proposed a novel chunking algorithm, the Frequency-Based Chunking (FBC), for data deduplication. The FBC algorithm is explicitly made use of the chunk frequency information from the data stream to enhance the data deduplication gain especially when the metadata overhead was taken into consideration. To achieve this goal, the FBC algorithm first utilized a statistical chunk frequency estimation algorithm to identify the globally appeared frequent chunks. It then employed a two-stage chunking algorithm to divide the data stream, in which the first stage applied the CDC algorithm to obtain a coarse-grained chunking results and the second stage further divided the CDC chunks with the identified frequent chunks. We conducted extensive experiments on heterogeneous datasets to evaluate the effectiveness of the proposed FBC algorithm. In all experiments, the FBC algorithm persistently outperformed the CDC algorithm in terms of achieving a better dedup gain or producing less number of chunks. Specifically, it produces 2.5 ~ 4 times less number of chunks than that of a baseline CDC does to achieve the same DER. Another benefit of FBC over CDC is that the FBC with average chunk size greater than or equal to that of CDC achieves up to 50% higher DER than that of CDC.

As mentioned in the end of Section IV, a given frequency threshold θ determines how aggressive the re-chunking should be carried out. In fact, a further thinking shows from a different aspect, if some coarse grained CDC chunks already appear to be high frequent (i.e. its frequency substantially exceeds the average chunk frequency), it may be desirable to skip rechunking those frequent large sized chunks because re-chunking them may result in no gain but more metadata overhead. In the future, we plan to refine our FBC algorithm by setting the escape frequency for coarse grained CDC chunks. Also, our existing approach requires an extra pass on the dataset to obtain frequency information. We plan to design a one-pass algorithm which conducts the frequency estimation and the chunking process simultaneously. An intuitive idea is to track top k frequent segments instead of tracking all segments with frequency greater than a certain threshold. This tracking can be easily integrated into FBC algorithm to make online frequency estimation.

VII. ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by grants from NSF (NSF Awards: 0960833 and 0934396)

REFERENCES

[1] **TRIDGELL, A., AND MACKERRAS, P.** The rsync algorithm. *Technical report TR-CS-96-05, Deparment of Computer Science.* 1996*)*

[2] **Sean Quinlan, Sean Dorward.** Venti: a New Approach to Archival Storage. *In Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02).* 2002.

[3] **Neil T. Spring, David Wetherall.** A Protocol-Independent Technique for Eliminating Redundant Network Traffic. *In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00).* 2000, pp. 87-95.

[4] **Broder, Andrei Z.** On the resemblance and containment of documents. *In Proc. of compression and complexity of sequences (SEQUENCES'97).* 1997.

[5] **A. Muthitacharoen, B. Chen, and D. Mazi`eres.** A low-bandwidth network file system. *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01).* October 2001, pp. 174-187.

[6] **Lawrence L. You, Kristal T. Pollack, Darrell D. E. Long.** Deep Store: An Archival Storage System Architecture. *in Proceedings of the 21st International Conference on Data Engineering* . April 2005, pp. 804--815.

[7] **K Eshghi, HK Tang.** A Framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR.* 2005.

[8] **Navendu Jain, Mike Dahlin, Renu Tewari.** TAPER: Tiered Approach for Eliminating Redundancy in Replica synchronization. *In Proceedings of the 2005 USENIX Conference on File and Storage Technologies (FAST'05).* 2005.

[9] **Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey.** Redundancy Elimination Within Large Collections of Files. *In Proceedings of 2004 USENIX Technical Conference.* 2004.

[10] **Calicrates Policroniades, Ian Pratt.** Alternatives for detecting redundancy in storage systems data. *In Proceedings of the 2004 Usenix Conference.* June 2004.

[11] **Manber, U.** Finding similar files in a large file system. *In Proceedings of the USENIX Winter 1994 Technical Conference.* January 1994, pp. 1-10.

[12] **Fred Douglis, Arun Iyengar.** Application-specific Delta-encoding via Resemblance Detection. *In Proceedings of 2003 USENIX Technical Conference.* 2003, pp. 113-126.

[13] **Deepak R. Bobbarjung, Suresh Jagannathand and Cezary Dubnicki.** Improving Duplicate Elimination in Storage Systems. *ACM Transaction on Storage.* 2006, Vol. 2, 4.

[14] **Lawrence L. You, Christos Karamanolis.** Evaluation of efficient archival storage techniques. *In Proceedings of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST).* April 2004.

[15] **Nagapramod Mandagere, Pin Zhou, Mark A Smith, Sandeep Uttamchandani.** Demystifying data deduplication. *In Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion.* 2008.

[16] **G. Manku, R.Motwant.** Approximate frequency counts over data streams. In Proceedings of the 28th International Conference on Very Large Data Bases, August, 2002.

[17] **Jin Cao, Yu Jin, Aiyou Chen, Tian Bu, Zhi-Li Zhang.** Identifying High Cardinality Internet Hosts. *The 28th Conference on Computer Communications (IEEE INFORCOM).* 2009.

[18] **Andrei Broder, Michael Mitzenmacher.** Network Applications of Bloom Filters: A Survey. *Internet Mathematics.* 2002, pp. 636--646.

[19] **Benjamin Zhu, Kai Li, Hugo Patterson.** Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *In Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'08).* 2008.

[20] **Erik Kruus, Cristian Ungureanu, Cezary Dubnicki.** Bimodal Content Defined Chunking for Backup Streams. *In Proceedings of of 8th USENIX Conference on File and Storage Technologies.* Feb. 2010.

[21] **K Eshghi, M Lillibridge, L Wilcock, G Belrose, R.Hawkes.** Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service. *Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST'07).* 2007.

[22] **Bhagwat, D., Eshghi, K., Long, D. D. E., and Lillibridge, M. .** Extreme binning: Scalable, parallel deduplication for chunk-based file backup. *In Proceedings of 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2009).*

[23] **Lillibridge, M. Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Gamble, P.** Sparse Indexing: large scale, inline deduplication using samping and locality. *In proceedings of the 7th conference on File and storage technologies.* 2009, pp. 111-123.

[24] **Rabin, Michael O.** Fingerprinting by random polynomials. Tech Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University. 1981.

[25] **Tony Summers.** Hardware Compression in Storage and Network Attached Storage. SNIA Tutorial, Spring 2007 (http://www.snia.org/education/tutorials/2007/spring/)