Estimation of Deduplication Ratios in Large Data Sets

Danny Harnik Oded Margalit Dalit Naor Dmitry Sotnikov Gil Vernik
IBM Research, Haifa, Israel.
{*dannyh,odedm,dalit,dmitrys,gilv*}*@il.ibm.com*

***Abstract*—We study the problem of accurately estimating the data reduction ratio achieved by deduplication and compression on a *specific* data set. This turns out to be a challenging task – It has been shown both empirically and analytically that essentially all of the data at hand needs to be inspected in order to come up with a accurate estimation when deduplication is involved. Moreover, even when permitted to inspect all the data, there are challenges in devising an efficient, yet accurate, method. Efficiency in this case refers to the demanding CPU, memory and disk usage associated with deduplication and compression. Our study focuses on what can be done when scanning the entire data set.**

We present a novel two-phased framework for such estimations. Our techniques are provably accurate, yet run with very low memory requirements and avoid overheads associated with maintaining large deduplication tables. We give formal proofs of the correctness of our algorithm, compare it to existing techniques from the database and streaming literature and evaluate our technique on a number of real world workloads. For example, we estimate the data reduction ratio of a 7 TB data set with accuracy guarantees of at most a 1% relative error while using as little as 1 MB of RAM (and no additional disk access). In the interesting case of full-file deduplication, our framework readily accepts optimizations that allow estimation on a large data set without reading most of the actual data. For one of the workloads we used in this work we achieved accuracy guarantee of 2% relative error while reading only 27% of the data from disk. Our technique is practical, simple to implement, and useful for multiple scenarios, including estimating the number of disks to buy, choosing a deduplication technique, deciding whether to dedupe or not dedupe and conducting large-scale academic studies related to deduplication ratios.

I. Introduction

a) Problem Statement and Challenges: With the explosion of data stored by organizations, deduplication and compression techniques are becoming ever more popular. The question of how much these techniques actually gain is becoming more and more relevant. This is exemplified by recent publications [25], [10], [22] trying to get a better understanding of the effect of deduplication, compression and their combination on some "real world scenarios". While we can gain quite a bit from general studies and get some rule-of-thumb estimations on what to expect from different techniques on various workloads, in practice, this is far from giving an accurate estimation for a specific workload and for a specific user. Experts working in the field of deduplication for backup claim they have seen deduplication ratios vary from 2:1 to 50:1 for the same application by the same vendor. The variations originate from many factors, including backup regimes, different setups, and user behavior and tendencies. So one cannot expect to get a precise estimate without examining the actual data at hand.

This work aims at giving techniques for *accurately* estimating deduplication and compression ratios for given data sets. A naïve approach would be to simply run the data reduction technique over the whole data set (perhaps just recording the savings rather than storing the reduced data set). While being very accurate, for large data sets this approach is prohibitively expensive (in terms of time, CPU/memory consumption and disk accesses). Instead, we look for techniques that give accurate estimates while not being excessive in terms of the required resources for this estimation.

As we show in this paper, this problem is far from being trivial. The most straightforward approach to overcome the resource limitations is to sample a subset of the data and compute exhaustively the data reduction ratio of the sampled data. However, this approach is bound to fail: it is common wisdom among practitioners who are skilled in the art that without incorporating specific knowledge on the repetitive structure of the data, it is impossible to predict the deduplication ratio accurately by looking only at a random subset of the data. Moreover, there are analytical proofs that such an approach can give arbitrarily skewed results. A simple test we have done on our data (the personal data workload) shows an error of more than 7% in dedupe ratio estimation when sampling as much as 37% of the data, and an error of 13% when looking at 21% of the data. Therefore, given this limitation, there are two ways to approach this problem: either incorporate major knowledge about the structure of the data to smartly sample it and then extrapolate what the overall ratio should be, or else to devise methods that look at all the data yet perform this efficiently with limited resources. In this paper we consider the latter (leaving the approach of "educated-sampling" for specific data types out of the scope of our paper).

b) Motivation and Applications: Why is this an important problem? Section VI provides an in-depth discussion on the motivation with practical examples. For example, when considering a new deduplication system, one needs to de-

The research leading to these results is partially supported by the European Communitys Seventh Framework Programme (FP7/2001-2013) under grant agreement n 257019 - VISION Cloud Project.

978-1-4673-1747-4/12/$31.00 ©

termine what would be the benefits, if any, of the systems; whether to dedupe or not; if so, what type of technique to use (fixed vs. variable, block vs. file etc). Even more so, one needs to decide how much storage to buy – a decision that translates directly to saving money. In a more advanced scenario, it may be questionable whether consolidated deduplication across storage pools is worth the cost associated with enlarging the deduplication domain. Moreover, in some extreme big-data scenarios (e.g. [25]), the meta-data itself is so big that even running simple tests on it requires an extensive effort.

c) Our Contributions: Our contributions in this paper are two fold. First, we set the grounds on the limitations and inherent difficulties (both analytical and in practice) of sampling techniques, and relate our results to known studies in other fields such as databases and streaming algorithms. We then devise a general framework that provides an efficient, yet provably accurate estimation method. The framework is general and is not tied to one particular deduplication technique or another. We provide an in-depth empirical study of this framework, based on four different workloads: personal workstations, enterprise file system repository, and backup data of two types. The largest data set contains approximately 7 TB of data. Our empirical study demonstrates that the accuracy obtained in practice is very close to the stated analytical bound, and that in some special cases only a small portion of the data is inspected. We finally elaborate on practice-and-experience issues related to implementing this estimation system.

A. The Challenges of Efficient Estimation

We now give a more in depth account of the challenges that the estimation problem poses. The starting point of our work comes from the realization that going over a small part of the data set is insufficient for estimating deduplication ratios accurately. In particular, sampling parts of the data set, whether randomly or according to various sampling methodologies can yield arbitrarily inaccurate deduplication ratios. This underlying fact is backed up both by practical experience and also by analytical proofs. Specifically, it was formalized in [8] who show both formal proofs and empirical tests and ultimately in [27] that present a near-linear lower bound on the sample size required to estimate the number of distinct elements in a set (near linear in the size of the set). In a nutshell, the difficulty stems from two central issues: i) unlike compression, in order to see the effects of data reduction that come from deduplication, one has to include in the sample more than one replica of a repeating item. Without prior knowledge on the locations of replication, one has a small chance of hitting the same element twice, and an extremely low probability of hitting k replicas of the same element (unless the sample size is very big). ii) Even if replicas are observed in the sample (e.g., if the sample is relatively large or due to prior knowledge) this still does not suffice to distinguish if the replication is a local phenomena or a global one. Hence, extrapolating from the sample to the full data set might be misleading. We stress that sampling based approaches can fail badly on workloads and distribution that are quite likely to appear in practice and not only in theory. One direction of overcoming these limitations is by gathering enough prior knowledge on the typical behavior of different workloads. This seems challenging since deduplication of similar data types may be highly influenced by the individuals involved (how much collaboration actually exists in the environment); see for example [30] on peer-assisted deduplication. Rather, in this paper, we study what can be done under the assumption that *essentially the whole data set is scanned*.

Even under the assumption that all data (or at least meta-data) in the data set is scanned, it is not clear how to estimate deduplication and compression ratios accurately in an efficient manner. The naïve approach to compute dedupe-ratios is to actually mimic the data reduction process. Namely, to record hash values for each element in the system (be it file or a chunk, depending on the technique studied), and its compression rate, in a *hash index-table*. However, this methodology has a high toll in terms of resources, a price that may be acceptable when running a system designed for deduplication, but not when one simply wants an estimate.

The main resources that are required when considering this approach are:

- **Memory:** The hash index table which can be very large either needs to be stored in main memory, thus consuming a lot of RAM, or paged on disk, thus entailing a major slow down in performance. For instance, a moderate sized data set of 7 TB requires in an optimal implementation around 24 GB of memory for 8 KB chunks stats. A naïve implementation using Python's dictionary structure (based on open addressing) required as much as 17 GB for a repository of only 1.2 TB. For larger data-sets, with hundreds of TBs this would become infeasible to store in memory, no matter what the implementation is.
- **Time:** The main bulk of time consumed by an estimation scan is taken up by the following operations: reading the data from disk, computing the hash, running a compression algorithm and updating the index table. The heaviest operations are access to the disk and compression. The computation of the hash function is far cheaper than both in this sense (see Appendix A). When a large index table is placed on disk, the random access to the disk causes substantial slow down. Real life deduplication systems (e.g. [31], [24], [12]) use an array of techniques ranging from clever caching to use of flash in order to improve the update time to the table. Most of these rely heavily on locality properties of backup streams which may not necessarily be as pronounced in primary data. The compression overhead is also very substantial (as demonstrated in Appendix A).

Another naïve approach would be to simply log all metadata on disk and process it off-line at a later stage. However, this method requires both the appropriate disk space and additional off-line processing to analyze large metadata files (which in turn consumes more time, disk accesses and CPU). This solution is inferior to our solution that provides an

answer immediately at the end of the scan and with very light resource consumption. In addition our method achieves improved scan times when compression is involved and in the full-file deduplication, improvement that are not gained with an off-line processing technique.

B. Our Results

We present a general framework for space efficient estimation of deduplication and compression techniques. The framework consists of two basic phases:

1) **Sample phase:** A base sample of elements are taken from the entire data set. The sample is taken at random where each element appears independently with probability that is relative to the element size (that is, identical probability for fixed chunk size but varies substantially for files). Hash values and compression rates are computed for each element in the sample.
2) **Scan phase:** Go over the entire data set and store statistics only about elements *in the base sample*. The hash is computed for each element but it is only recorded if it matches a hash of an element in the base sample.

The statistics recorded in the scan phase are used to derive the data-reduction ratio estimate.

We give an analytical proof that, using a small base sample, our framework indeed estimates the data-reduction ratio with high accuracy (Section IV), and back our claims with experimental results on a number of different workloads (Section V). With as little as 200 MB of RAM, we can provably guarantee estimations to within a 1% relative error for workloads with high compression and deduplication ratios. One key observation is that we can do better for data sets with smaller deduplication ratios. In fact, our experimental results demonstrate that with as little as 1 MB of memory we can estimate the deduplication ratio of a 7 TB data set to within 1% of its true value. Moreover, our guarantees work with *the same sample size* also for much larger data-sets, even, on the order of petabytes. Thus the benefits become much more pronounced as the data set grows and our methods more useful with the current trend of growth in data stores.

Our methodology is general and works for a multitude of deduplication techniques. In the paper we study specific methods including fixed and variable sized chunking, full-file deduplication and combinations with compression. Our techniques can also be adapted for other cases such as distributed systems (for example the real life system presented in [14]). The low memory requirement allows the process to place all of its information in main memory and thus perform the scan without extra disk accesses. Moreover, our frameworks naturally accommodates efficiency improvements in several scenarios. For instance, our algorithms benefit from the fact that they need to compute the heavy compression algorithms only on the base sample. This saves one of the most costly operations, as demonstrated in Appendix A. Note also that the scan phase is highly parallelizable using a standard map-reduce framework and thus can perform well in highly distributed storage systems.

Finally, in the special case of full-file deduplication and compression we present an adaptation of our framework that uses file length and hashes on the first block of data in each file (typically this is part of the file's inode) and avoid reading the actual data for disk for almost all data that is not relevant to the base sample. In our examples, we manage to read only 27% of the data from disk, yet achieve accuracy of at least 2%.

C. Related Work

The question of estimating the number of distinct elements in a large collection of elements has been well studied over the years both from an analytical aspect and from a practical point of view. This problem has received attention in two main fields of computing, the first is in studies of databases and more recently in the realm of streaming algorithms. A good overview of these works can be found in [18]. Both communities observe that sampling based algorithms may fail for many relevant inputs with the strong formal lower bounds proved in [8], [27]. The problem was raised in the context of databases where estimating the number of distinct elements in a column serves as a tool for performing several operations. There are numerous algorithms that estimate the number of distinct elements using small space. A central underlying technique in most of these works stems from the work of Flajolet and Martin [16], by which all elements are mapped onto a small numerical segment (using a hash function onto, say, $[0, 1]$). At the end, the minimal value that is mapped to in the segment serves as an indicator to the estimated number of elements. On this basic techniques, a large number of variations and stochastic mappings were built to provide better estimates with low space (E.g., [4], [6], [20], [23] is a partial list). Another closely related approach introduced in [19], [17] maintains at all times a bounded size sample of distinct elements. This approach differs from ours as its sample is uniform on the distinct elements set, rather than the entire data set. Most of these techniques are comparable to ours with respect to memory requirements, except for the second algorithm of [4] and the recent optimal bound of [23]. The techniques can be readily applied to estimate the *fixed size chunk* deduplication ratio. However, they need to be adapted in order to succeed with variable size chunks and even more so for the case of full files (where the variance in element size is much larger). Moreover, in order to adapt these techniques to deal with compression as well, one would have to compress *all* of the elements in the data-set (in contrast to our solutions that require compressing only the elements in the base sample. Note that computing the deduplication and compression ratios separately and taking their multiplication as the overall ratio is an inaccurate method, since it is quite common to have a correlation between what dedupes well and what compresses well. Indeed in some of our workloads we observed errors of over 7% when using this practice. In the distinct sampling techniques [17], the number of compressed elements can be reduced dramatically, but is still higher than the fraction compressed in our method. Finally, all the above

mentioned techniques do not benefit from our optimizations for full-file deduplication.

In the realm of deduplication, there is a large body of "deduplication calculators" which do not examine the actual data; they are surveyed in Section VI. A recent paper of Constantinescu and Lu [11] raises similar questions to ours. They suggest a sampling method, but only for the easier task of estimating compression rates (and not deduplication). For full-file deduplication, they also suggest using the file length and first hash, but in their method the occurrence of undesired collisions results in reduced accuracy, whereas we pay by reading a slightly larger fraction of the data, but do not compromise the estimations precision. In addition, their scheme does not cope well with the problem of a large index for huge data sets.

Finally, the use of sampling is not new to deduplication. It has been used to improve efficiency in handling of the hash index (e.g. [24]), and to help in making distribution decisions in large distributed deduplication systems (e.g., [7], [30], [14]).

II. Preliminaries

Deduplication and compression are widely used in storage systems. In this paper we discuss either techniques that consider deduplication alone, or combine deduplication with "local" compression. In compression we refer techniques that look at a single file or chunk of data and compress it on its own, typically using variants of the Lempel-Ziv algorithm (e.g. [32]). The common approach to combining deduplication and compression is to first define a deduplication element, with the main choices being fixed of variable sized chunks (usually on the order of 2-16 KBs each) or full files/objects. Compression is typically performed after deduplication, on a per element basis so that it does not interfere with the benefits of deduplication. This is the type of compression and deduplication combo that we discuss in this paper. The basic methodology for deduplicating data is by fingerprinting each chunk/file using a good cryptographic hash function (such as the 160 bit SHA1). The hash function practically ensures no inadvertent collisions. Now the deduplication process maintains a hash index table of all elements (hash values) that already appeared in the data set. When a new element arrives, it is either found in the table, and deduplicated against an existing element, or it is added to the index table (along with a pointer to where it is stored).

III. Our Estimation Technique

A. The General framework

Our framework works in two phases, a sampling and a scanning phase. We use the term elements to describe the basic deduplication unit – a file or a chunk of fixed or variable size. The two phases are outlined below:

1) The Sampling Phase: From the entire data-set (denoted $\mathcal{S}$) we choose m elements randomly where m is a parameter chosen in advance (in Section IV-A we elaborate on how to choose m). For each element, we calculate its hash value and add it to a set that we call the *base sample* (denoted $\mathcal{B}$). Multiple appearances of the same hash value are merged into one entry in $\mathcal{B}$ and each such entry holds a counter of how many instances of this hash were in the sample. Each chosen element is taken in random from the whole data set, giving each element a probability that is proportional to its size in the data set. For each element in the base sample, indexed $i \in \mathcal{B}$, the following data is computed and recorded: (1) h_i – A hash signature of the element; (2) ρ_i – The compression ratio of the element (in case compression is used. $\rho = 1$ otherwise); (3) $base_i$ – The number of appearances of an element with this hash signature in the base sample; and (4) $count_i$ – set initially to zero.
2) The Scanning Phase: This is the heavier part of the algorithm in which the entire data-set is scanned. For each element $e \in \mathcal{S}$ its hash signature h_e is computed. If this signature matches h_i for some $i \in \mathcal{B}$ then $count_i$ is incremented by 1. If h_e does not match any element in the base sample then it is ignored. Note that there is no relevance to the specific order of the scan (in fact it can be run in parallel). In addition at this stage, no compression statistics are gathered, but rather signatures are computed. This is crucial since compression is a much heavier task than computation of a typical cryptographic hash function.

At the end of the scan, the following data reduction estimate is computed:

$$Est = \frac{1}{m} \sum_{i \in \mathcal{B}} \frac{base_i \cdot \rho_i}{count_i}$$

Note that the length of the elements are not recorded, but come into play during the sampling phase.

B. Chunk Level Deduplication

We turn to describe in more detail the algorithm for the case of fixed size chunk deduplication. The data-set $\mathcal{S}$ consists of n chunks of equal size (for example 8 KBs each). For the sampling phase, this means that each chunk has independent probability $\frac{m}{n}$ to be in the base sample. Note that this is not *distinct element sampling* (see [17]) in which the sample is uniform over the set of distinct elements. In our case, an element that has two replicas in the data set has double the probability of being included in the base sample. Moreover, more than one replica of an element can be taken to the base sample (since these are merged into one in the base sample, the size of the set $\mathcal{B}$ can be smaller than m).

d) Sampling Chunks: Knowledge of n, the total number of chunks in the data set, is essential in the sampling process. The overall size of the data set can be computed by a standard traversal of the file system (e.g., unix `du` command), or extracted from existing metadata statistics on the data set (e.g., unix `df` command). There are several approaches that can be used for sampling. We list the main options:

1) One can choose m random numbers in $\{1, ..., n\}$ (one can eliminate repetitions here although this is not crucial to the success of the estimation). Now go over some ordering of the data set in order to find the chosen

chunks. In a file system, for instance, this would require a traversal of the directory tree, and the use of the file sizes in order to figure out which files need to be read and at what offset.

2) An alternative approach is to traverse the whole directory tree and make decisions on a per file basis. Let ℓ denote the number of chunks in the file. Now generate a random number k according to the binomial distribution of ℓ Bernouli trials with probability $\frac{m}{n}$ per trial. Namely, choose $k \sim B(\ell, \frac{n}{m})$ (there are standard libraries for generating such distributions). If $k = 0$ then no chunk was chosen from the file (and the file can be ignores). If $k \geq 1$ choose k random chunks in $\{1, ..., \ell\}$ and add them to the sample. This approach requires more random coin tosses than the first approach and also return m samples on average, but maybe slightly less (this can be easily remedied by using $\frac{m'}{n}$ with m' slightly larger than m and then choosing m of the chosen chunks at random). The benefit is that this approach does not need to store the m chosen indices at any point, and is essentially stateless and thus can be run in parallel.

Having the base sample prepared, we are ready for the scan phase. The entire algorithm is described in the following pseudocode (Algorithm III.1).

Algorithm III.1: CHUNK ESTIMATE($\mathcal{S}$)

Chunk Sample($\mathcal{S}$)
Choose sample of m *random elements* $\in \mathcal{S}$
for each e *in sample*
  do if $\exists\ i \in \mathcal{B}$ *s.t.* $h_e = h_i$
  then $base_i \leftarrow base_i + 1$
  else $\begin{cases} add\ e\ to\ \mathcal{B}\ and\ record: \\ h_e \leftarrow hash\ signature\ of\ e \\ \rho_e \leftarrow the\ compression\ ratio\ of\ e \\ base_e \leftarrow 1 \\ count_e \leftarrow 0 \end{cases}$

Chunk Scan($\mathcal{S}$)
for each e *in* $\mathcal{S}$
  do $\begin{cases} \textbf{if } \exists\ i \in \mathcal{B}\ s.t.\ h_e = h_i \\ \quad \textbf{then } count_i \leftarrow count_i + 1 \end{cases}$

C. Full file deduplication

This scenario in which deduplication is only done between identical files has its disadvantages mainly as it achieves less than optimal deduplication ratios in many cases. Yet it is a popular choice since it is typically easier to implement and was shown to perform sufficiently well in some workloads , especially when combined with compression (see [25], [10]). Our methodology translates well in this setting, and using some additional tweaks can actually reduce significantly the amount of data actually read from disk. Although the metadata for all files will be scanned, the actual data needs to be read only for a small fraction of the files, which is related to the base sample.

e) Sampling Files: In the case of files one needs to take into account the length of a file since there is a great variance between file sizes (this is true also, to a lesser extent, for variable sized chunks). Moreover, the total size of the data-set is no longer naturally counted in terms of chunks, but rather in terms of a common denominator of the lengths in which files are stored. This can either be the page size of the file system (if files are stored as a collection of full pages), or in the most general case in terms of single bytes. Denote by N the total number of *bytes* in the data-set. In our handling of files, each such byte has independent probability $\frac{m}{N}$ to be chosen, and for each chosen byte the owning file (or chunk) is included in the base sample. Note that the same file might be chosen more than once for the base sample, and this duplication should be recorded.

The actual sampling follows closely the two options raised for chunks (Section III-B). In option 1, m offsets are chosen in $\{1, ..., N\}$ and a file is chosen to be part of the base sample if it contains a chosen offset. If it contains more than one offset then the base counter of this file reflects this. In option 2, the base counter is the result of the corresponding binomial random variable (where 0 means it is not in the sample).

f) Scan phase optimizations: In the case of full files we take advantage of metadata that is readily available in a typical file system in order to reduce the need to read all data from disk in the scan phase. The point is that we only need to process files that are relevant to the base sample (their hash is in the base sample). By simply looking at the *file length*, we can rule out for many files the possibility that they are relevant, since a file can only be there if a file with the same length is ready in the bases sample. This serves in a similar role as a Bloom filter, namely, only look at the data if the data has a chance of being relevant. A second filter is a *hash on the first block of the file*. This could be as short as a single page of the file system. In many file systems this first block resides in the i-node of the file and thus can be read quickly during a metadata scan without the addition of extra disk seeks. Only files that have both length and first hash matching an entry in the base sample need to be read from disk. Formally we add the following information into the base sample. For each file in the base sample, indexed $i \in \mathcal{B}$, we add:

- ℓ_i - The length of the file.
- $h1_i$ - A hash signature on the first block of the file.

Now the process during the *scan phase*, for each element (file) e in the data set do the following:

1) If the length of the file ℓ_e matches ℓ_i for some $i \in \mathcal{B}$ continue. Otherwise ignore the file e.
2) Compute the hash on the first block of e to get $h1_e$. If there exists $i \in \mathcal{B}$ such that $\ell_e = \ell_i$ and $h1_e = h1_i$ then continue. Otherwise ignore the file e.
3) Compute the full hash on the file e to get h_e. If there exists $i \in \mathcal{B}$ such that $h_e = h_i$ then $count_i = count_i + 1$. Otherwise ignore the file e.

The ratio at the end of the process is computed as before. Namely:

$$Est = \frac{1}{m} \sum_{i \in \mathcal{B}} \frac{base_i \cdot \rho_i}{count_i}$$

Again, notice that the length of the files is ignored in the computation of the ratio (although it is recorded). The rationale is that in our sampling method a long file is more likely to appear in the base sample than a short one (and in extreme cases will even appear twice or more), which is desirable as a long file is more influential on the overall compression rate than a short file. This bias in the sampling allows us to avoid adding a bias in the ratio calculation. The detailed pseudocode is given in Algorithm III.2.

Algorithm III.2: FULL-FILE ESTIMATE($\mathcal{S}$)

File Sample($\mathcal{S}$)
for each $file\ e\ \in \mathcal{S}$
  do $k \leftarrow B(\ell_e, \frac{m}{N})$
if $k > 0$
  then if $\exists\ i \in \mathcal{B}\ s.t.\ h_e = h_i$
  then $base_i \leftarrow base_i + k$
  else
$add\ e\ to\ \mathcal{B}\ and\ record:$
$h_e \leftarrow hash\ signature\ of\ e$
$\rho_e \leftarrow the\ compression\ ratio\ of\ e$
$\ell_e \leftarrow length\ of\ e$
$h1_e \leftarrow hash\ of\ the\ first\ block\ of\ e$
$base_e \leftarrow k$
$count_e \leftarrow 0$

Full File Scan($\mathcal{S}$)
for each $file\ e\ in\ \mathcal{S}$
do
if $\exists i \in \mathcal{B}\ s.t.\ \ell_e = \ell_i$
then
$compute\ h1_e$
if $\exists i \in \mathcal{B}\ s.t.\ \ell_e = \ell_i$ **and** $h1_e = h1_i$
then
$compute\ h_e$
if $\exists i \in \mathcal{B} s.t.\ \ell_e = \ell_i$ **and** $h1_e = h1_i$ **and** $h_e = h_i$
then $count_i = count_i + 1$

g) Variable sized chunking: Variable sized chunking poses a challenge, since one can neither figure out how many chunks are in a file nor at what offset the j^{th} chunk might be, without reading and chunking the entire file. Instead, the sampling should choose exact offsets in the files, and then choose the chunk which contains this offset. Suppose that an offset k was chosen in a file, then this can be implemented by reading the file at an offset $k - max_{chunk}$ and chunking from there until the chunk containing the relevant offset is found. This both relieves the need to read entire files and achieves the desired result of giving each chunk a probability that is linear to the chunk's actual length.

D. Systems Implementation Issues

h) Maintaining the base sample: There are numerous ways to hold the base sample during the scan phase. The most economic in terms of memory space is by sorting the base sample according to the hash value at the end of the sample phase. Since no insertions are needed during the scan phase, then updating the counters in the base sample only entails lookups, that can be performed by searching over a sorted array (at the cost of $\log m$ lookups in the RAM table). Thus at run time one can get by with as little as $m \cdot 20$ bytes of memory for the hash values (the output length of a SHA1 cryptographic hash function), plus $m \cdot 4$ more bytes for the count and compression ratio numbers.

An alternative is to use more sophisticated hash based structures that can work with good memory utilization such as cuckoo hashing or open addressing. These will gain faster average lookup times (constant number per lookup), but will require more memory since their performance deteriorates when they become full. In addition there are some overheads in handling the data structures (the open addressing implementation used for Python's dictionary demonstrated a 7 fold overhead to the dictionary size).

We note that the fact that our algorithm only does lookups in the scan phase, rather than insertions and deletions of items to the base sample, alleviates the run time of the scan of the heaviest operations associated with maintaining data structures for fast lookup.

i) Parallel execution: Note that the scan phase can be run in parallel on a distributed system. The base sample needs to be circulated to all the scanning nodes, and each node will do the scan locally and accumulate the count for the data adjacent to this node. At the end of the process, all of the counts are accumulated centrally and the data reduction ratio is calculated. This fits naturally in the Map Reduce framework for parallel computing. Note that we cannot avoid holding the entire base sample at each node, so a process running on k nodes in parallel will require holding k simultaneous copies of the base sample.

IV. ANALYSIS OF ACCURACY

In this section we present a formal proof that our estimation technique is successful. We start by defining what it means to be a successful estimation algorithm.

Definition 1. *A probabilistic algorithm $Est(\cdot)$ is said to be an (ε, δ)-estimation scheme if for every data-set $\mathcal{S}$ with data reduction ratio r, we have $Pr[\frac{|Est(\mathcal{S}) - r|}{r} > \varepsilon] < \delta$ where the probability is taken over the randomness of Est.*

Our basic accuracy claim is stated in the following Theorem:

Theorem 1. *The algorithm presented in section III when run over a data set with reduction ratio r and uses a base sample of size $m > \frac{\ln 2 + \ln \frac{1}{\delta}}{2\varepsilon^2 r^2}$ is an (ε, δ)-estimation scheme.*

The proof is given in Appendix A.

A. Choosing the Size of the Base Sample

Given the analytical result in Theorem 1, we can deduce the size that would be sufficient for the base sample. Note however that the ratio r which is the target of the estimation process

appears in the bound of how many sample points are required. Specifically, the higher the data-reduction ratio is (and the better the data compresses) the harder it becomes to give an accurate estimation (this also makes sense since an error of 1% of 1 TB is much smaller than the same proportional error of 10 TBs). In order to choose the sample size we must first give a bound r_{max} on the expected compression ratio and plug it into the formula from Theorem1 and taking $m = \left\lceil \frac{\ln 2 + \ln \frac{1}{\delta}}{2\varepsilon^2 r_{max}^2} \right\rceil$. We can also predict that the memory overhead required for each entry in the base sample to be as little as $m \cdot 24$ bytes of memory (as explained in Section III-D).

Some examples of choices for sample sizes are given in Table I.

r_{max}	ε	δ	m	Memory
3:1	0.01	0.0001	44557	1 MB
5:1	0.01	0.0001	1237938	28.3 MB
15:1	0.01	0.0001	11142000	255 MB

TABLE I
The table summarizes the required sample size and memory needed to hold this sample for achieving an (ε, δ)-estimation with ratio at most r_{max}

Note that the estimated values do not grow with the size of the data set at hand. This means that our techniques are more appealing as the data set grows. While the benefits may not be as pronounced when looking at a system with a few hundreds of GBs, it is quite clear that for systems with many TBs of data (let alone PBs) the size of the required base sample is very small.

V. Empirical Evaluation

We evaluate our technique by testing it on a number of real world workloads. The workloads are taken from varying environments in order to capture different behaviors of data and to test that our algorithms work well on all of these scenarios. We used four different data sets as described below:

1) **Personal workstations:** This workload includes storage dumps of laptops and workstations of 16 R&D employees within a company. Some are within the same organizational unit, but some are not. The data-set includes 6.3 Million files amounting to a total capacity of 1.1 TB.
2) **Major file repository of enterprise organization:** This is the organization's main file repository and is utilized by a multitude of users for various purposes. It contains 16.4 Million files and a total capacity of 7 TB was tested. The data was collected in 4 KB chunks but no compression data was collected, so the tests for this data set consider only deduplication and no compression.
3) **Backup storage of small enterprise organization:** This is the backend repository of an organization's backup system. The repository contains backup of user files as well as development environment. We collected 250 GB of data from this repository in 1.2 Million files.
4) **Periodic backup of exchange DB:** This is a classical backup setting of an exchange DB. The data set consists of 13 periodical backups of the entire 17 GB DB taken on an almost daily basis. All together the data set contains $13 \cdot 17 = 221$ GB. This data was used in the chunk based tests only, and not in the full-file deduplication case, since it only contains 13 large files.

A. Empirical Accuracy Results

We ran our tests on the aforementioned workloads to validate our algorithms accuracy and to evaluate how this accuracy behaves when real world distributions are involved. We validated this behavior across all of our workloads, by running 1000 independent tests per workload (each time the sampling algorithm uses fresh random coins). The algorithm runs with arbitrarily growing sample sizes in order to view the behavior as the sample size grows. The graphs show the general trends that we have seen across all workloads (for each such behavior we present only a single representative graph in order to avoid repetition and save space).

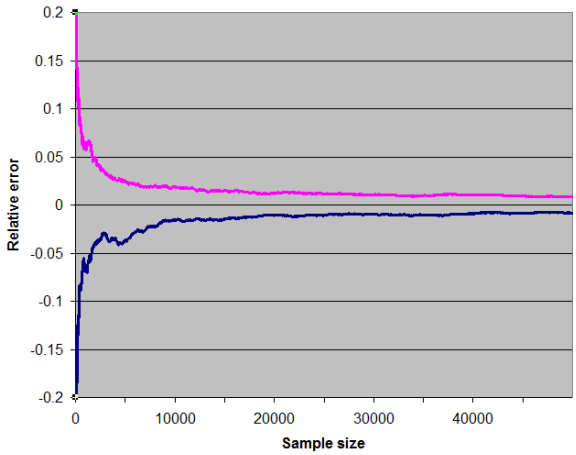

Fig. 1. The graph depicts our test on the file repository data for fixed chunk deduplication. We run 1000 different test and depict the highest and lowest deviations from the actual data as a function of the base sample size. We observe that after 43000 samples the error always remains below 1%.

Figure 1 shows the behavior of our estimation method as a function of the sample size. This is for the file repository workload and we see a nice converging behavior of the algorithm as the sample size grows with the error going well below 1% at approximately 43000 points in the base sample. This depicts the extreme errors out of 1000 tests, where the majority of the tests yielded far better estimations. A more comprehensive look at the tests looks at the standard deviation of the 1000 samples. We show this (Figure 2) on a different workload, the *personal data* workload with compression. This workload shows the diminishing standard deviation as a function of the sample size, yet at a slower rate (an artifact of the better overall ratio).

Moreover, we can see that the behavior for a fixed sample size shows a nice looking gaussian as seen in Figure 3. This is expected since our algorithm produces the sum of (essentially) independent random variables, and this sum should behave

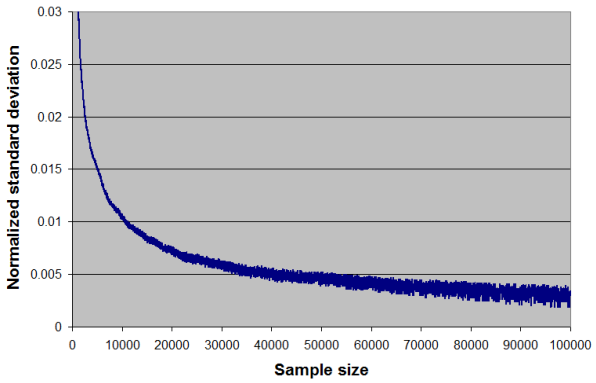

Fig. 2. The graph shows the normalized standard deviation as a function of the sample size, as computed for 1000 tests on the personal data workload for fixed chunk deduplication and compression. The normalization is by the target ratio. That is, a standard deviation of 0.01 equals 1% of the target ratio.

according to the normal distribution. Combined with the standard deviation measure we can deduce the assurance δ that our estimation will indeed fall within an ε error of the target ratio.

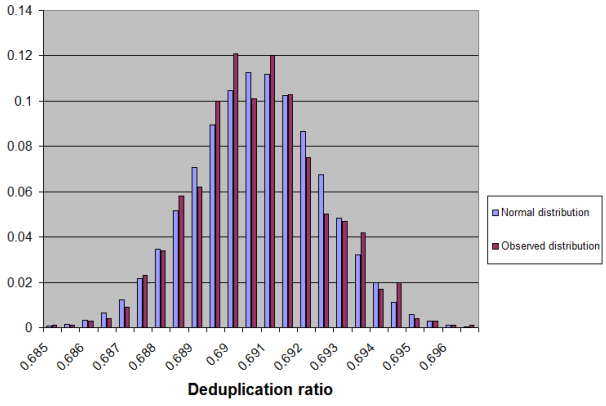

Fig. 3. We show the distribution of results of the estimation process as obtained from 1000 executions with sample size 50000. The tests were run on the file repository data which has deduplication ratio 0.691. We compare the distribution to a true normal distribution with the same expectation and variance.

Our findings regarding chunk based deduplication are detailed in Table II both for deduplication and for the combination of deduplication and compression.

The results that we get are only slightly better than those presented by the analytical bounds. The improvement can be explained due to some slackness in the concentration bounds that we use. These bounds take into account arbitrarily bad distributions (and in fact assume nothing on the variance of the underlying distribution. The distribution displayed in real life situations is in general a nice distribution. For example, while the highest frequency of a chunk in the 7 TB data set is on the order of 121 Million, the next in line is only of order 1 Million. This isn't close to what it could potentially be, given that the total number of chunks is approximately 1260 Million.

j) The ratio matters: A key point that we see both here and in the analytical bounds is that the data reduction ratio plays a big role in setting the bound of how many samples need to be used. Specifically, for bad ratios, such as $r = 0.69$ for the files repository deduplication, the number of samples needed can be as little as 40000. However, when the reduction is effective, such as the case of the backup repository with $r = 0.107$, this balloons to 419000. This phenomena forces us to obtain a reasonably good idea of what the actual ratio maybe in order to be more efficient. Rule of thumb estimations can be used here as initial, very rough, estimators in order to pick a satisfactory sample size. Note that if the chosen sample size was too small, this will be discovered once the estimation results are produced. In such a case, we can still make use of the estimation, but the confidence level will decrease and a larger potential error should be taken into account.

B. Full-File Deduplication

As in the case of chunks, we evaluate our full-file algorithm on the relevant workloads (excluding the exchange DB). The results appear in Table III. The results are very similar to those presented in the case of chunks.

The enticing feature of our method for full-file deduplication and compression is the ability to forgo reading a large portion of the data from disk. So in addition to the accuracy results, we run estimates on how big this portion can actually be in a real life scenario. Basically, only files related to the base sample are read. If they have the length of a file in the base sample then their first block is read, and if their first block is identical to one in the base sample (with the same length) then the whole file is read. This means that the workload's characteristics have a large influence on the fraction read. For example, if the lengths of files vary a lot or tend to repeat. Or even more so, if files in the base sample have large multiplicities (a high dedupe ratio) leads to reading more files. We estimate the benefit for the largest data set (7 TBs of data) and learn that one can get guarantees of up to 2% relative error when reading 26.6% of the actual data. For a 1% relative error guarantee we our algorithm requires reading 46% of the data. The full tradeoff between accuracy and percentage read is depicted in Figure 4.

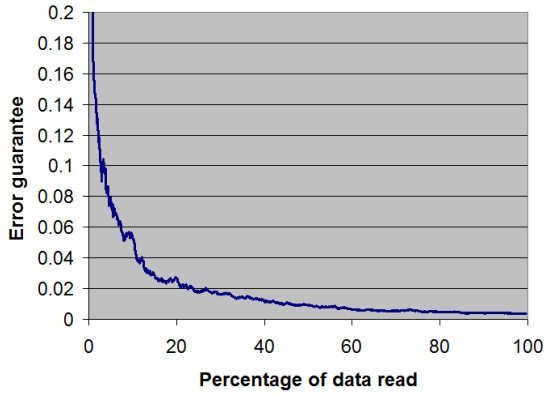

Fig. 4. The graph depicts the tradeoff between percentage read and estimation accuracy. The error is calculated according to the worst error of 1000 independent tests, and the percentage read is the average over these tests.

Workload	Dedupe only			Dedupe+compression		
	ratio	$\varepsilon = 0.01$	$\varepsilon = 0.015$	ratio	$\varepsilon = 0.01$	$\varepsilon = 0.015$
Personal data	0.559	69000 (1.6 MB)	25600 (0.6 MB)	0.359	98000 (2.2 MB)	50600 (1.2 MB)
File repository	0.691	43000 (1 MB)	13000 (0.3 MB)	–	–	–
Backup repository	0.571	60000 (1.4 MB)	22300 (0.5 MB)	0.107	419000 (9.6 MB)	242000 (5.5 MB)
Exchange backup	0.109	300800 (6.9 MB)	101000 (2.3 MB)	0.076	213200 (4.9 MB)	84100 (1.9 MB)

TABLE II

Chunk based deduplication: *The sample size required (and memory requirement in parenthesis) for different accuracy levels as indicated by 1000 executions over each of the workloads.*

Workload	Dedupe only			Dedupe+compression		
	ratio	$\varepsilon = 0.01$	$\varepsilon = 0.015$	ratio	$\varepsilon = 0.01$	$\varepsilon = 0.015$
Personal data	0.665	44500 (1 MB)	16200 (0.4 MB)	0.377	94300 (2.2 MB)	43600 (1 MB)
File repository	0.820	18700 (0.4 MB)	12500 (0.3 MB)	–	–	–
Backup repository	0.685	38700 (0.9 MB)	13000 (0.3 MB)	0.128	400000 (9.2 MB)	318000 (7.3 MB)

TABLE III

Full-file deduplication: *The sample size required (and memory in parenthesis) for different accuracy levels as indicated by 1000 executions over each of the workloads.*

Recall that a straightforward, yet representative, sampling of 37% of the data-set that we ran on our personal data repository yielded an error of 7% (this was for a single test, and there is no guarantee that this cannot be occasionally much worse if more tests were run). In contrast, our algorithm can ensure an error of no more than 5% when reading just 10.5% of the data. The potential of this technique is immense for data sets that are much larger than the 7 TB, since the sample size does not have to grow. Extrapolating the results we have here to a repository of 500 TBs would yield guarantees of 1% on by reading as little as 5% of the data. We hope to be able to validate this when data of such proportions will become available.

VI. Motivation and Further Applications

There are a number of reasons why one should strive for an accurate measure of what is in store when planning to use a data reduction technology. We list some of these motivations:

- **How many disks to buy:** This question comes up in practice when data reduction is used, but is paramount when a customer starts using a backup device with deduplication (e.g., [2], [1] among many others). These devices take pride in saving massive amounts of storage space with their advance data reduction techniques, allowing the user to buy far fewer disks for his backup. Reports of how much one can gain move from a 2:1 ratio up to 100:1 in extreme cases. Getting a good read on this ratio before hand can have major consequences: Over estimate results in overspending , under estimate and you may end up without enough space for your backups. The most common approach to address this problem, is to use one of the myriad "deduplication calculator" designed to answer this challenge [3], [5], [9], [13], [26]. The practice in these calculator is for the user to answer a number of questions about their application types and backup habits and to come up with an estimate of how much storage space they would likely consume. None of these techniques actually require to look at the users real data which, in many cases is accessible. Experts in the field claim that these estimation techniques are useful in general (especially with lack of a better alternative) but at times are terribly off the mark. For example, it had been observed that the deduplication factor of a major vendor's database may vary from 2:1 to 50:1, depending on the configuration used by the site admin and the actual data involved, whether it is backed up daily or weekly. A comprehensive discussion about the factors that should be taken into account and affect in practice the deduplication ratio can be found in [15]. In conclusion, relying solely on the "deduplication calculators" approach may lead to significant cost implications when estimating how many disks to buy.

  k) Note:: Our methodology does not always fit directly for the case of backup since a) typically the data as a whole is inaccessible but rather one has access to a few versions of a data set (or just to the dynamically changing data-set); and b) the typically very high deduplication ratios require more memory in our methods. Yet our techniques may prove quite useful in answering accurately two questions that are crucial to the success of estimation calculators: a) what is the change rate between backed up versions of the same data set? and b) what is the data reduction potential within a single version of the data-set?
- **What technique to use:** Studies (e.g. [29]) have shown that the reduction ratio may vary highly according to the specific workload at hand, but also according to the underlying data reduction technique (i.e., compression, full file deduplication, fixed chuck size vs. variable chunk size in deduplication and combinations of all of the above). The best technique for one user is not necessarily the optimal one for another. One can of course try all techniques and figure out which is best, but this may be prohibitively costly. Since our estimation can be taylored for many deduplication techniques, it can be used as an efficient and feasible tool for understanding which technique is best for an existing data set, by estimating the ratio using any one of the methods.

- **To dedupe or not to dedupe?** Deduplication and compression do not come for free [31], [24]. Neither the technology (the price of the software/hardware involved) nor the system resources involved once the technology is in place (such as response times, CPU and memory usage). With this in mind, one should make a conscious decision of how much to invest in data reduction, and knowledge of how much is to gain by employing such a technology is key to making a wise decision. With primary data, it is often questioned whether any data reduction technique is worthwhile? We aim at giving tools to evaluate this on a given data set.

In addition our technique is particularly useful to handle the following applications efficiently:

l) Consolidation of storage pools: In many distributed systems deduplication is done locally in each node/server. An interesting question is how much there is to be gained by consolidating the deduplication efforts across nodes. Such a consolidation is more challenging to implement due to the substantially larger hash table and requires a higher amount of interaction between nodes that one might hope for (see designs of such efforts in [7], [14], [24]). Our methods can be used to accurately estimate the benefits that can be achieved from such an effort, serving as a tool for deciding whether to invest in such a direction. Since the metadata (hash values) exist in every node separately, our method gives a cheap way for estimating the deduplication across all nodes without the need to create the full hash index table. It can also be used to decide which nodes, if any, to consolidate with which.

m) Scientific studies: An example of the difficulties involved in estimating deduplication ratios on a large scale system can be seen in studies such as the work of Meyer and Bolosky [25]. Because of the huge amounts of metadata involved in the computations, several techniques (such as bloom filters, throwing out all metadata on singletons and shortening the hash signatures) were necessary in order to make the estimation possible. The engineering efforts invested in such an estimation can be greatly relieved when employing our methods.

n) Systems with built in disk scrubbing: One of the problems with methods that scan entire data sets is that simply reading the data is prohibitively expensive. We note that many storage systems today actually do this anyhow as a mean for detecting latent errors in their data – a practice known as disk scrubbing [28]. In such systems, one can piggy back the scrubbing scan in order to also give precise estimations of data reduction potential of the data in the system.

VII. Conclusions

We tackle the problem of estimating deduplication and compression ratios, a problem that is becoming more and more relevant with the admission of more deduplication techniques to large storage repositories. We view our techniques as a tool that will become more and more effective as such systems become more widely used and in ever growing scales. Anyone who has tried to analyze data at the scale of single terabytes knows how much simple engineering efforts are needed to cope with the scale. This becomes prohibitively challenging at even larger scales (e.g. in the research of [25]). We hope our techniques will lead the way to getting accurate estimates on very large data sets in future research.

Directions for further research include using known properties of the data in order to devise methods than can do educated sampling of the data-set and extrapolate from this to the entire data-set. Note that our work is orthogonal to such effort, since it can be used in combo with any successful sampling technique. This is because the sample itself may be quite large and our methods can be used in order to analyze it efficiently without taxing memory requirements. One direction is to use our methodology as a tool for dedupe estimation calculators for backup. Our methods can give accurate estimations on the change rate and in-volume compression, as a mean for calculating the overall capacity required for a backup mechanism.

References

[1] EMC Data Domain. http://www.datadomain.com/.

[2] IBM ProtecTIER. http://www-03.ibm.com/systems/storage/news/center/deduplication/index.html.

[3] Acronis. http://www.acronis.com/backup-recovery/deduplication-roi-calculator.html.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[5] Avamar. http://thebackupblog.typepad.com/thebackupblog/2008/06/deduplication-calculator—avamar-aware.html.

[6] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

[7] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS*, pages 1–9, 2009.

[8] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[9] Commvault. http://www.commvault.com/deduplication/calculator/index.asp.

[10] C. Constantinescu, J. S. Glider, and D. D. Chambliss. Mixing deduplication and compression on active data sets. In *DCC*, pages 393–402, 2011.

[11] C. Constantinescu and M. Lu. Quick estimation of data compression and de-duplication for large storage systems. In *CCP 2011 - 1st International Conference on Data Compression, Communications and Processing*, Salerno, Italy, 2011.

[12] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 16–16. USENIX Association, 2010.

[13] D. Domain. http://www.dedupecalculator.com/.

[14] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *FAST*, pages 15–29, 2011.

[15] M. Dutch and L. Freeman. *Understanding data de-duplication ratios*. SNIA, February 2009. http://www.snia.org/forums/dmf/news/articles/SNIA_DeDupe_Ratio_Feb09.pdf.

[16] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[17] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[18] P. B. Gibbons. Distinct-values estimation over data streams. In *Manuscript*, 2009.

[19] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA*, pages 281–291, 2001.

[20] F. Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.
[21] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 301(58):13–30, 1963.
[22] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *SYSTOR*, page 7, 2009.
[23] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52, 2010.
[24] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *FAST*, pages 111–123, 2009.
[25] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *FAST*, pages 1–13, 2011.
[26] Netapp. http://www.secalc.com/do_calc.php.
[27] S. Raskhodnikova, D. Ron, A. Shpilka, and A. Smith. Strong lower bounds for approximating distribution support size and the distinct elements problem. *SIAM J. Comput.*, 39(3):813–842, 2009.
[28] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *MASCOTS*, pages 409–418, 2004.
[29] K. Tangwongsan, H. Pucha, D. G. Andersen, and M. Kaminsky. Efficient similarity estimation for systems exploiting data redundancy. In *Proc. IEEE INFOCOM*, San Diego, CA, Mar. 2010.
[30] Y. Xing, Z. Li, and Y. Dai. Peerdedupe: Insights into the peer-assisted sampling deduplication. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–10. IEEE, 2010.
[31] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, pages 269–282, 2008.
[32] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

APPENDIX

Proof:

The proof follows by considering the random variable generated by our estimation algorithm and presenting it with a different formulation. We view this random variable as an average of m random variables x_j, where each x_j is generated by randomly and uniformly choosing an element $i \in \mathcal{S}$ and setting $x_j = \frac{\rho_i}{count_i}$. The choice of a new x_j is done taking into account the previous choices as to not repeat already chosen elements. Thus the x_j's are taken from what is termed *sampling without replacement*. In the case of files (or variable sized chunks), the random choice is weighted according to the length of the file. The estimation random variable is defined as $Est = \frac{1}{m}\sum_{j=1}^{m} x_j$.

Note that this formulation differs slightly from the formulation given in Section III. Specifically this formulation does not include the base counter $base_i$, but this is only a semantic change that groups identical elements together in the base sample in order to get a more efficient implementation and the formulations are in fact equivalent. In order to use this formulation we first observe that the expectation of output equals the data-reduction ratio of the data-set.

Lemma 1. $E(Est) = r$

Given the above lemma we can apply Hoeffding's bound [21], to bound the deviation of $\bar{X}$ from its expectation, the desired ratio.

Theorem 2 (Hoeffding [21])**.** *Let $x_1, ..., x_k$ be independent random variables each in the domain $x_i \in [a, b]$, and denote by $\bar{X} = \frac{1}{k}\sum_{i=1}^{k} x_i$ then*

$$Pr[|\bar{X} - E(\bar{X})| > t] \leq 2e^{-\frac{2kt^2}{(b-a)^2}}$$

Moreover, the above holds if $x_1, ..., x_k$ are random samples from a population taken without replacement.

Taking $a = 0$ and $b = 1$, $k = m$, $\bar{X} = Est$, $t = \varepsilon r$ and $E(\bar{X}) = r$ we get:

$$Pr[|\bar{X} - r| > \varepsilon r] < 2e^{-2m\varepsilon^2 r^2}$$

Plugging in $m > \frac{\ln 2 + \ln\frac{1}{\delta}}{2\varepsilon^2 r^2}$ we get $Pr[|\bar{X} - r| > \varepsilon r] < 2e^{-\frac{2(\ln 2 + \ln\frac{1}{\delta})\varepsilon^2 r^2}{2\varepsilon^2 r^2}} = 2e^{-(\ln 2 + \ln\frac{1}{\delta})} = \delta$. □

Proof:

[of Lemma 1]

The lemma follows by since the expectation also holds for the individual random variables x_j. That is, for all j we have $E(x_j) = r$ and from the linearity of expectations $Est = \frac{1}{m}\sum_{j=1}^{m} E(x_j) = r$.

Showing that $E(x_j) = r$ is straightforward in the case of fixed size chunks. Suppose the data set $\mathcal{S}$ contains n elements and denote by $\mathcal{D}$ the set of distinct elements in $\mathcal{S}$, containing D elements. The data reduction ratio can be denoted as $r = \frac{1}{n}\sum_{i\in\mathcal{D}} \rho_i$ where ρ_i is the compression ratio of this element (in case no compression is used then $\rho_i = 1$ for all i and $r = \frac{D}{n}$). In reality, each element $i \in \mathcal{D}$ is constituted of $count_i$ replicas in $\mathcal{S}$ and it can be viewed that each of the $count_i$ replicas contributes $\frac{\rho_i}{count_i}$ to the overall compressed representation of the data set. So an alternative way to view the ratio is as an average over all n elements, where each element contributes $\frac{\rho_i}{count_i}$ to the sum. Now by definition of x_j we have $E(x_j) = \frac{1}{n}\sum_{i\in\mathcal{S}} \frac{\rho_i}{count_i} = r$.

In the full-file case, the ratio is $r = \frac{1}{n}\sum_{i\in\mathcal{D}} \rho_i \ell_i$ where ℓ_i is the length of file i and n denotes the overall length of the data set rather then the number of elements in it. Alternatively, $r = \frac{1}{n}\sum_{i\in\mathcal{S}} \frac{\rho_i \ell_i}{count_i}$. Since x_j takes a random element, and the probability of the i^{th} element is $\frac{\ell_i}{n}$, then $E(x_j) = \sum_{i\in\mathcal{S}} \frac{\ell_i}{n}\frac{\rho_i}{count_i} = r$. A similar argument holds for the case of variable sized chunks. ■

To complement our study, we test the performance overhead of some key operations needed for the estimation scheme to run. These are reading from the disk, computing of the hash signature and compressing a chunk. The test was run once with a moderate file of 40 MBs and once with a large 8 GB file. We used standard python libraries on a Intel Core2 Duo CPU, with 2.33 GHz and 1.96 GB of RAM. The compression is a standard LZ process run at 8 KB chunks. Our results are detailed in the Table IV.

Process	8 GB file	40 MB file
Disk read+overheads	16.4%	20.5%
Compute sha1 hash	5%	4%
LZ compression	78.6%	75.5%

TABLE IV
Percentage of the overhead for key operations