Estimating Unseen Deduplication – from Theory to Practice

Danny Harnik, Ety Khaitzin and Dmitry Sotnikov
IBM Research–Haifa
{dannyh, etyk, dmitrys}@il.ibm.com

Abstract

Estimating the deduplication ratio of a very large dataset is both extremely useful, but genuinely very hard to perform. In this work we present a new method for accurately estimating deduplication benefits that runs $3X$ to $15X$ faster than the state of the art to date. The level of improvement depends on the data itself and on the storage media that it resides on. The technique is based on breakthrough theoretical work by Valiant and Valiant from 2011, that give a provably accurate method for estimating various measures while seeing only a fraction of the data. However, for the use case of deduplication estimation, putting this theory into practice runs into significant obstacles. In this work, we find solutions and novel techniques to enable the use of this new and exciting approach. Our contributions include a novel approach for gauging the estimation accuracy, techniques to run it with low memory consumption, a method to evaluate the combined compression and deduplication ratio, and ways to perform the actual sampling in real storage systems in order to actually reap benefits from these algorithms. We evaluated our work on a number of real world datasets.

1 Introduction

1.1 Deduplication and Estimation

After years of flourishing in the world of backups, deduplication has taken center stage and is now positioned as a key technology for primary storage. With the rise of all-flash storage systems that have both higher cost and much better random read performance than rotating disks, deduplication and data reduction in general, makes more sense than ever. Combined with the popularity of modern virtual environments and their high repetitiveness, consolidating duplicate data reaps very large benefits for such high-end storage systems. This trend is bound to continue with new storage class memories looming, that are expected to have even better random access and higher cost per GB than flash.

This paper is about an important yet extremely hard question – How to estimate the deduplication benefits of a given dataset? Potential customers need this information in order to make informed decisions on whether high-end storage with deduplication is worthwhile for them. Even more so, the question of sizing and capacity planning is deeply tied to the deduplication effectiveness expected on the specific data. Indeed, all vendors of deduplication solutions have faced this question and unfortunately there are no easy solutions.

1.2 The hardness of Deduplication Estimation and State of the Art

The difficulty stems from the fact that deduplication is a *global* property and as such requires searching across large amounts of data. In fact, there are theoretical proofs that this problem is hard [12] and more precisely, that in order to get an accurate estimation one is required to read a large fraction of the data from disk. In contrast, *compression* is a local procedure and therefore the compression estimation problem can be solved very efficiently [11].

As a result, the existing solutions in the market take one of two approaches: The first is simply to give an educated guess based on prior knowledge and based on information about the workload at hand. For example: Virtual Desktop Infrastructure (VDI) environments were reported (e.g. [1]) to give an average 0.16 deduplication ratio (a 1:6 reduction). However in reality, depending on the specific environment, the results can vary all the way between a 0.5 to a 0.02 deduplication ratio (between 1:2 and 1:50). As such, using such vague estimation for sizing is highly inaccurate.

The other approach is a full scan of the data at hand. In practice, a typical user runs a full scan on as much data as possible and gets an accurate estimation, but *only for the data that was scanned*. This method is not without challenges, since evaluating the deduplication ratio of a scanned dataset requires a large amount of memory and disk operations, typically much higher than would be allowed for an estimation scan. As a result, research on the topic [12, 19] has focused on getting accurate estimations with low memory requirement, while still reading all data from disk (and computing hashes on all of the data).

In this work we study the ability to estimate deduplication while *not* reading the entire dataset.

1.3 Distinct Elements Counting

The problem of estimating deduplication has surfaced in the past few years with the popularity of

the technology. However, this problem is directly linked to a long standing problem in computer science, that of *estimating the number of distinct elements* in a large set or population. With motivations ranging from Biology (estimating the number of species) to Data Bases (distinct values in a table/column), the problem received much attention. There is a long list of heuristic statistical estimators (e.g. [4, 9, 10]), but these do not have tight guarantees on their accuracy and mostly target sets that have a relatively low number of distinct elements. Their empirical tests perform very poorly on distributions with a long tail (distributions in which a large fraction of the data has low duplication counts) which is the common case with deduplication. Figure 1 shows examples of how inaccurate heuristic estimations can be on a real dataset. It also shows how far the deduplication ratio of the sample can be from that of the entire dataset.

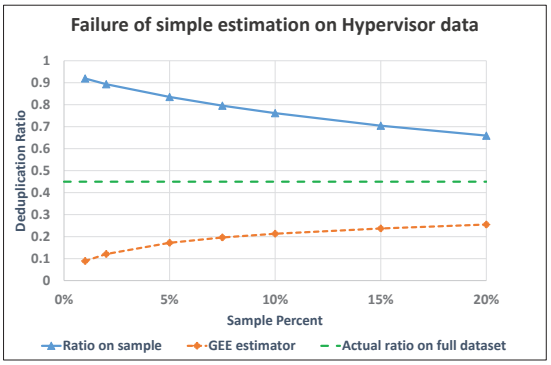


Figure 1: An example of the failure of current sampling approaches: Each graph depicts a deduplication ratio estimate as a function of the sampling percent. The points are an average over 100 random samples of the same percent. We see that simply looking at the ratio on the sample gives a very pessimistic estimation while the estimator from [4] is always too optimistic in this example.

This empirical difficulty is also supported by theoretical lower bounds [16] proving that an accurate estimation would require scanning a large fraction of the data. As a result, a large bulk of the work on distinct elements focused on low-memory estimation on data streams (including a long list of studies starting from [7] and culminating in [14]). These estimation methods require a full scan of the data and form the foundation for the low-memory scans for deduplication estimation mentioned in the previous section.

In 2011, in a breakthrough paper, Valiant and Valiant [17] showed that at least $\Omega(\frac{n}{\log n})$ of the data must be inspected and more over, that there is a matching upper bound. Namely, they showed a theoretical algorithm that achieves provable accuracy if at least $O(\frac{n}{\log n})$ of the elements are examined. Subsequently, a variation on this algorithm was also implemented by the same authors [18]. Note that the Valiants work, titled "Estimating the unseen" is more general than just distinct elements estimation and can be used to estimate other measures such as the Entropy of the data (which was the focus in the second paper [18]).

This new "Unseen" algorithm is the starting point of our work in which we attempt to deploy it for deduplication estimation.

1.4 Our Work

More often than not, moving between theory and practice is not straightforward and this was definitely the case for estimating deduplication. In this work we tackle many challenges that arise when trying to successfully employ this new technique in a practical setting. For starters, it is not clear that performing random sampling at a small granularity has much benefit over a full sequential scan in a HDD based system. But there are a number of deeper issues that need to be tackled in order to actually benefit from the new approach. Following is an overview of the main topics and our solutions:

Understanding the estimation accuracy. The proofs of accuracy of the Unseen algorithm are theoretic and asymptotic in nature and simply do not translate to concrete real world numbers. Moreover, they provide a worst case analysis and do not give any guarantee for datasets that are easier to analyze. So there is no real way to know how much data to sample and what fraction is actually sufficient. In this work we present a novel method to gauge the accuracy of the algorithm. Rather than return an estimation, our technique outputs a range in which the actual result is expected to lie. This is practical in many ways, and specifically allows for a gradual execution: first take a small sample and evaluate its results and if the range is too large, then continue by increasing the sample size. While our tests indicate that a 15% sample is sufficient for a good estimation on all workloads, some real life workloads reach a good enough estimation with a sample as small as 3% or even less. Using our method, one can stop early when reaching a sufficiently tight estimation.

The memory consumption of the algorithm. In real systems, being able to perform the estimation with a small memory footprint and without additional disk IOs is highly desirable, and in some cases a must. The problem is that simply running the Unseen algorithm as prescribed requires mapping and counting all of the distinct chunks in the sample. In the use case of deduplication this number can be extremely large, on the same order of magnitude as the number of chunks in the entire

dataset. Note that low memory usage also benefits in lower communication bandwidth when the estimation is performed in a distributed system.

Existing solutions for low-memory estimation of distinct elements cannot be combined in a straightforward manner with the Unseen algorithm. Rather, they require some modifications and a careful combination. We present two such approaches: one with tighter accuracy, and a second that loses more estimation tightness but is better for distributed and adaptive settings. In both cases the overall algorithm can run with as little as 10MBs of memory.

Combining deduplication and compression. Many of the systems deploying deduplication also use compression to supplement the data reduction. It was shown in multiple works that this combination is very useful in order to achieve improved data reduction for all workloads [5, 13, 15]. A natural approach to estimating the combined benefits is to estimate each one separately and multiply the ratios for both. However, this will only produce correct results when the deduplication effectiveness and compression effectiveness are independent, which in some cases is not true. We present a method to integrate compression into the Unseen algorithm and show that it yields accurate results.

How to perform the sampling? As stated above, performing straightforward sampling at a small granularity (e.g. 4KB) is extremely costly in HDD based systems (in some scenarios, sampling as little as 2% may already take more than a full scan). Instead we resort to sampling at large "super-chunks" (of 1MB) and performing reads in a sorted fashion. Such sampling runs significantly faster than a full scan and this is the main source of our time gains.

Equally as important, we show that our methods can be tuned to give correct and reliable estimations under this restriction (at the cost of a slightly looser estimation range). We also suggest an overall sampling strategy that requires low memory, produces sorted non-repeating reads and can be run in gradual fashion (e.g. if we want to first read a 5% sample and then enlarge the sample to a 10% one).

Summary of our results. In summary, we design and evaluate a new method for estimating deduplication ratios in large datasets. Our strategy utilizes less than 10MBs of RAM space, can be distributed, and can accurately estimate the joint benefits of compression and deduplication (as well as their separate benefits). The resulting estimation is presented as a range in which the actual ratio is expected to reside (rather than a single number). This allows for a gradual mode of estimation, where one can sample a small fraction, evaluate it and continue sampling if the resulting range is too loose. Note that the execution time of the actual estimation algorithms is negligible vs. the time that it takes to scan the data, so being able to stop with a small sampling fraction is paramount to achieving an overall time improvement.

We evaluate the method on a number of real life workloads and validate its high accuracy. Overall our method achieves at least a 3X time improvement over the state of the art scans. The time improvement varies according to the data and the medium on which data is stored, and can reach time improvements of 15X and more.

2 Background and the Core algorithm

2.1 Preliminaries and Background

Deduplication is performed on data chunks of size that depends on the system at hand. In this paper we consider fixed size chunks of 4KB, a popular choice since it matches the underlying page size in many environments. However the results can be easily generalized to different chunking sizes and methods. Note that variable-sized chunking can be handled in our framework but adds complexity especially with respect to the actual sampling of chunks.

As is customary in deduplication, we represent the data chunks by a hash value of the data (we use the SHA1 hash function). Deduplication occurs when two chunks have the same hash value.

Denote the dataset at hand by $\mathcal{S}$ and view it as consisting of N data chunks (namely N hash values). The dataset is made up from D distinct chunks, where the i^{th} element appears n_i times in the dataset. This means that $\sum_{i=1}^{D} n_i = N$.

Our ultimate target is to come up with an estimation of the value D, or equivalently of the ratio $r = \frac{D}{N}$. Note that throughout the paper we use the convention where data reduction (deduplication or compression) is a ratio between in $[0, 1]$ where lower is better. Namely, ratio 1.0 means no reduction at all and 0.03 means that the data is reduced to 3% of its original size (97% saving).

When discussing the sampling, we will consider a sample of size K out of the entire dataset of N chunks. The corresponding sampling rate is denoted by $p = \frac{K}{N}$ (for brevity, we usually present p in percentage rather than a fraction). We denote by $\mathcal{S}_p$ the random sample of fraction p from $\mathcal{S}$.

A key concept for this work that is what we term a Duplication Frequency Histogram (DFH) that is defined next (note that in [17] this was termed the "fingerprint" of the dataset).

Definition 1. *A* **Duplication Frequency Histogram (DFH)** *of a dataset $\mathcal{S}$ is a histogram $x = \{x_1, x_2, ...\}$ in which the value x_i is the number of distinct chunks that appeared exactly i times in $\mathcal{S}$.*

For example, the DFH of a dataset consisting of N distinct elements will have $x_1 = N$ and zero elsewhere. An equal sized dataset where all elements appear exactly twice will have a DFH with $x_2 = \frac{N}{2}$ and zero elsewhere. Note that for a legal DFH it must hold that $\sum_i x_i \cdot i = N$ and moreover that $\sum_i x_i = D$. The length of a DFH is set by the highest non-zero x_i. In other words it is the frequency of the most popular chunk in the dataset. The same definition of DFH holds also when discussing a sample rather than entire dataset.

The approach of the Unseen algorithm is to estimate the DFH of a dataset and from it devise an estimation of the deduplication.

2.2 The Unseen Algorithm

In this section we give a high level presentation of the core Unseen algorithm. The input of the algorithm is a DFH y of the observed sample $\mathcal{S}_p$ and from it the algorithm finds an estimation $\widehat{x}$ of the DFH of the entire dataset $\mathcal{S}$. At a high level, the algorithm finds a DFH $\widehat{x}$ on the full set that serves as the "best explanation" to the observed DFH y on the sample.

As a preliminary step, for a DFH x' on the dataset $\mathcal{S}$ define the **expected DFH** y' on a random p sample of $\mathcal{S}$. In the expected DFH each entry is exactly the statistical expectancy of this value in a random p sample. Namely y'_i is the expected number of chunks that appear exactly i times in a random p fraction sample. For fixed p this expected DFH can be computed given x' via a linear transformation and can be presented by a matrix A_p such that $y' = A_p \cdot x'$.

The main idea in the Unseen algorithm is to find an x' that minimizes a distance measure between the expected DFH y' and the observed DFH y. The distance measure used is a normalized L1 Norm (normalized by the values of the observed y). We use the following notation for the exact measure being used:

$$\Delta_p(x', y) = \sum_i \frac{1}{\sqrt{y_i + 1}} \left| y_i - (A_p \cdot x')_i \right|.$$

The algorithm uses Linear Programming for the minimization and is outlined in Algorithm 1.

The actual algorithm is a bit more complicated due to two main issues: 1) this methodology is suited for estimating the duplication frequencies of unpopular chunks. The very popular chunks can be estimated in a straightforward manner (an element with a high count c is expected to appear approximately $p \cdot c$ times in the sample). So the DFH is first broken into the easy part for straightforward estimation and the hard part for estimation via Algorithm 1. 2) Solving a Linear program with too many variables is impractical, so instead of solving for a full DFH x, a sparser mesh of values is used (meaning that not all duplication values are allowed in x). This relaxation is acceptable since this level of inaccuracy has very little influence for high frequency counts. It is also crucial to make the running time of the LP low and basically negligible with respect to the scan time.

Algorithm 1: Unseen Core

Input: Sample DFH **y**, fraction **p**, total size **N**
Output: Estimated deduplication ratio $\widehat{\mathbf{r}}$
/* Prepare expected DFH transformation*/
$A_p \leftarrow$ prepareExpectedA(p);

Linear program:
Find x' that minimizes: $\Delta_p(x', y)$
Under constraints: /* x' is a legal DFH */
$\sum_i x'_i \cdot i = N$ and $\forall i\ x'_i \geq 0$

return $\widehat{r} = \frac{\sum_i x'_i}{N}$

The matrix A_p is computed by a combination of binomial probabilities. The calculation changes significantly if the sampling is done with repetition (as was used in [18]) vs. without repetition. We refer the reader to [18] for more details on the core algorithm.

3 From Theory to Practice

In this section we present our work to actually deploying the Unseen estimation method for real world deduplication estimation. Throughout the section we demonstrate the validity of our results using tests on a single dataset. This is done for clarity of the exposition and only serves as a representative of the results that where tested across all our workloads. The dataset is the Linux Hypervisor data (see Table 1 in Section 4) that was also used in Figure 1. The entire scope of results on all workloads appears in the evaluation section (Section 4).

3.1 Gauging the Accuracy

We tested the core Unseen algorithm on real life workloads and it has impressive results, and in general it thoroughly outperforms some of the estimators in the literature. The overall impression is that a 15% sample is sufficient for accurate estimation. On the other hand the accuracy level varies greatly from one workload to the next, and often the estimation obtained from 5% or even less is sufficient for all practical purposes. See example in Figure 2.

So the question remains: how to interpret the estimation result and when have we sampled enough? To address these questions we devise a new approach that returns a range of plausible deduplica-

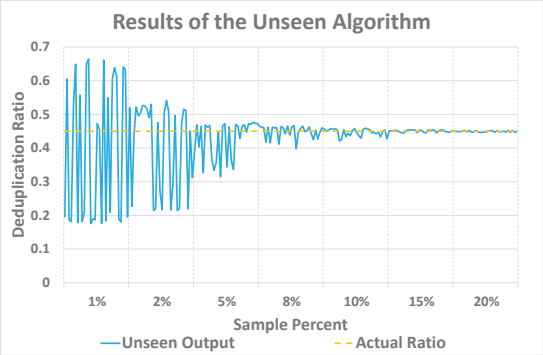


Figure 2: The figure depicts sampling trials at each of the following sample percentages 1%, 2%, 5%, 7.5%, 10%. 15% and 20%. For each percentage we show the results of Unseen on 30 independent samples. The results are very noisy at first, for example, on the 1% samples they range from ~ 0.17 all the way to ~ 0.67. But we see a nice convergence starting at 7.5%. The problem is that seeing just a single sample and not 30, gives little indication about the accuracy and convergence.

tion ratios rather than a single estimation number. In a nut shell, the idea is that rather than give the DFH that is the "best explanation" to the observed y, we test all the DFHs that are a "reasonable explanation" of y and identify the range of possible duplication in these plausible DFHs.

Technically, define criteria for all plausible solutions x' that can explain an observed sample DFH y. Of all these plausible solutions we find the ones which give minimal deduplication ratio and maximal deduplication ratio. In practice, we add two additional linear programs to the first initial optimization. The first linear program helps in identifying the neighborhood of plausible solutions. The second and third linear programs find the two limits to the plausible range. In these linear programs we replace the optimization on the distance measure with an optimization on the number of distinct chunks. The method is outlined in Algorithm 2.

Note that in [18] there is also a use of a second linear program for entropy estimation, but this is done for a different purpose (implementing a sort of Occam's razor).

Why does it work? We next describe the intuition behind our solution: Consider the distribution of $\Delta_p(x, y)$ for a fixed x and under random y (random y means a DFH of a randomly chose p-sample). Suppose that we knew the expectancy $E(\Delta_p)$ and standard deviation σ of Δ_p. Then given an observed y, we expect, with very high probability, that the only plausible source DFHs x' are such that $\Delta_p(x', y)$ is close to $E(\Delta_p)$ (within $\alpha \cdot \sigma$ for some slackness variable α). This set of plausible x' can be fairly large, but all we really care to learn about it is its boundaries in terms of deduplication ratio. The second and third linear programs find out of this set of plausible DFHs the ones with the best and worst deduplication ratios .

Algorithm 2: Unseen Range

Input: Sample DFH **y**, fraction **p**, total size **N**, slackness α (default $\alpha = 0.5$)
Output: Deduplication ratio range $[\mathbf{r}, \bar{\mathbf{r}}]$

/* Prepare expected DFH transformation*/
$A_p \leftarrow$ prepareExpectedA(p);

1^{st} **Linear program:**
Find x' that minimizes: $\Delta_p(x', y)$
Under constraints: /* x' is a legal DFH */
$\sum_i x'_i \cdot i = N$ and $\forall i\ x'_i \geq 0$

For the resulting x' compute: $Opt = \Delta_p(x', y)$

2^{nd} **Linear program:**
Find $\underline{x}$ that minimizes: $\sum_i \underline{x}_i$
Under constraints:

$$\sum_i \underline{x}_i \cdot i = N \text{ and } \forall i\ \underline{x}_i \geq 0 \text{ and } \Delta_p(\underline{x}, y) < Opt + \alpha\sqrt{Opt}$$

3^{rd} **Linear program:**
Find $\overline{x}$ that maximizes: $\sum_i \overline{x}_i$
Under constraints:

$$\sum_i \overline{x}_i \cdot i = N \text{ and } \forall i\ \overline{x}_i \geq 0 \text{ and } \Delta_p(\overline{x}, y) < Opt + \alpha\sqrt{Opt}$$

return $[\underline{r} = \frac{\sum_i \underline{x}_i}{N}, \overline{r} = \frac{\sum_i \overline{x}_i}{N}]$

The problem is that we do not know how to cleanly compute the expectancy and standard deviation of Δ_p, so we use the first linear program to give us a single value within the plausible range. We use this result to estimate the expectancy and standard deviation and give bounds on the range of plausible DFHs.

Setting the slackness parameter.. The main tool that we have in order to fine tune the plausible solutions set is the slackness parameter α. A small α will result in a tighter estimation range, yet risks having the actual ratio fall outside of the range. Our choice of slackness parameter is heuristic and tailored to the desired level of confidence. The choices of this parameter throughout the paper are made by thorough testing across all of our datasets and the various sample sizes. Our evaluations show that a slackness of $\alpha = 0.5$ is sufficient and one can choose a slightly larger number for playing it safe. A possible approach is to use two different levels of

slackness and present a "likely range" along with a "safe range".

In Figure 3 we see an evaluation of the range method. We see that our upper and lower bounds give an excellent estimation of the range of possible results that the plain Unseen algorithm would have produced on random samples of the given fraction.

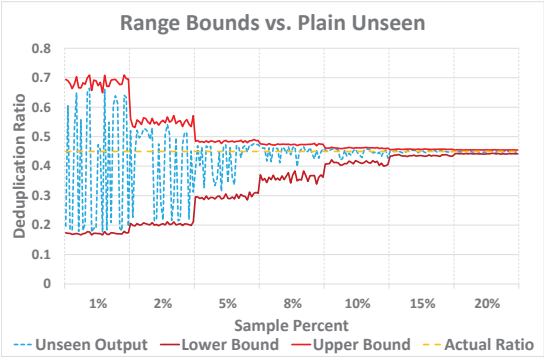


Figure 3: This figure depicts the same test as in Figure 2, but adds the range algorithm results. This gives a much clearer view – For example, one can deduce that the deduplication ratio is better than 50% already at a 5% sample and that the range has converged significantly at this point, and is very tight already at 10%.

An interesting note is that unlike many statistical estimation in which the actual result has a high likelihood to be at the center of the range, in our case all values in the range can be equally likely.

Evaluating methods via average range tightness. The range approach is also handy in comparing the success of various techniques. We can evaluate the average range size of two different methods and choose the one that gives tighter average range size using same sample percentage. For example we compare running the algorithm when the sampling is with repetitions (this was the approach taken in [18]) versus taking the sample without repetitions. As mentioned in Section 2.2, this entails a different computation of the matrix A_p. Not surprisingly, taking samples without repetitions is more successful, as seen in Figure 4. This is intuitive since repetitions reduce the amount of information collected in a sample. Note that sampling without repetition is conceptually simpler since it can be done in a totally stateless manner (see Section 3.4). Since the results in Figure 4 were consistent with other workloads, we focus our attention from here on solely on the no repetition paradigm.

3.2 Running with Low Memory

Running the estimation with a 15% sample requires the creation of a DFH for the entire sample, which in turn requires keeping tab on the duplication frequencies of all distinct elements in the sample.

Figure 4: An average range size comparison of sampling with repetitions vs. without repetitions. For each sampling percentage the averages are computed over 100 samples. Other than the 1% sample which is bad to begin with, the no repetition sampling is consistently better.

Much like the case of full scans, this quickly becomes an obstacle in actually deploying the algorithm. For example, in our largest test data set, that would mean keeping tab on approximately 200 Million distinct chunks, which under very strict assumptions would require on the order of 10GBs of RAM, unless one is willing to settle for slow disk IOs instead of RAM operations. Moreover, in a distributed setting it would require moving GBs of data between nodes. Such high resource consumption may be feasible in a dedicated system, but not for actually determining deduplication ratios in the field, possibly at a customer's site and on the customers own servers.

We present two approaches in order to handle this issue, both allowing the algorithm to run with as little as 10MBs of RAM. The first achieves relatively tight estimations (comparable to the high memory algorithms). The second produces somewhat looser estimations but is more flexible to usage in distributed or dynamic settings.

The base sample approach. This approach follows the low-memory technique of [12] and uses it to estimate the DFH using low memory. In this method we add an additional base step so the process is as follows:

1. **Base sample:** Sample C chunks from the data set (C is a "constant" – a relatively small number, independent of the database size). Note that we allow the same hash value to appear more than once in the base sample.
2. **Sample and maintain low-memory chunk histogram:** Sample a p fraction of the chunks and iterate over all the chunks in the sample. Record a histogram (duplication counts) for all the chunks in the base sample (and ignore the rest). Denote by c_j the duplication count of the j^{th} chunk in the base sample ($j \in \{1, ..., C\}$).

3. **Extrapolate DFH:** Generate an estimated DFH for the sample as follows:

$$\forall i,\ y_i = \frac{|\{j|c_j = i\}|}{i}\frac{pN}{C}.$$

In words, use the number of chunks in the base sample that had count i, extrapolated to the entire sample.

The crux is that the low-memory chunk histogram can produce a good approximation to the DFH. This is because the base sample was representative of the distribution of chunks in the entire dataset. In our tests we used a base sample of size $C = 50,000$ which amounts to less than 10MBs of memory consumption. The method does, however, add another estimation step to the process and this adds noise to the overall result. To cope with it we need to increase the slackness parameter in the Range Unseen algorithm (from $\alpha = 0.5$ to $\alpha = 2.5$). As a result, the estimation range suffers a slight increase, but the overall performance is still very good as seen in Figure 5.

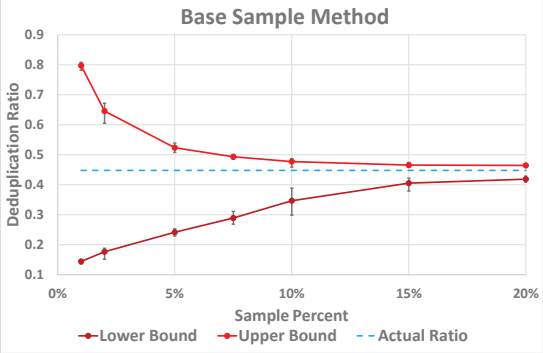


Figure 5: The graph depicts the average upper and lower bounds that are given by the algorithm when using the base sample method. Each point is the average over 100 different samples, and the error bars depict the maximum and minimum values over the 100 tests.

The only shortcoming of this approach is that the dataset to be studied needs to be set in advance, otherwise the base sample will not cover all of it. In terms of distribution and parallel execution, the base sample stage needs to be finished and finalized before running the actual sampling phase which is the predominant part of the work (this main phase can then be easily parallelized). To overcome this we present a second approach, that is more dynamic and amenable to parallelism yet less tight.

A streaming approach. This method uses techniques from streaming algorithms geared towards distinct elements evaluation with low memory. In order to mesh with the Unseen method the basic technique needs to be slightly modified and collect frequency counts that were otherwise redundant. The core principles, however, remain the same: A small (constant size) sample of distinct chunks is taken *uniformly over the distinct chunks* in the sample. Note that such a sampling disregards the popularity of a specific hash value, and so the most popular chunks will likely not be part of the sample. As a result, this method cannot estimate the sample DFH correctly but rather takes a different approach. Distinct chunk sampling can be done using several techniques (e.g. [2, 8]). We use here the technique of [2] where only the C chunks that have the highest hash values (when ordered lexicographically) are considered in the sample. The algorithm is then as follows:

1. **Sample and maintain low-memory chunk histogram:** Sample a p fraction of the chunks and iterate over all the chunks in the sample. Maintain a histogram only of chunks that have one of the C highest hash values:
   - If the hash is in the top C, increase its counter.
   - If it is smaller than all the C currently in the histogram then ignore it.
   - Otherwise, add it to the histogram and discard the lowest of the current C hashes.

   Denote by δ the fraction of the hash domain that was covered by the C samples. Namely, if the hashes are calibrated to be numbers in the range $[0, 1]$ then δ is the distance between 1 and the lowest hash in the top-C histogram.
2. **Run Unseen:** Generate a DFH solely of the C top hashes and run the Range Unseen algorithm. But rather than output ratios, output a range estimation on the number of distinct chunks. Denote this output range by $[\underline{d}, \overline{d}]$.
3. **Extrapolation to full sample:** Output estimation range $[\underline{r} = \frac{\underline{d}}{\delta \cdot N}, \overline{r} = \frac{\overline{d}}{\delta \cdot N}]$.

Unlike the base sample method, the streaming approach does not attempt to estimate the DFH of the p-sample. Instead, it uses an exact DFH of a small δ fraction of the hash domain. The Unseen algorithm then serves as a mean of estimating the actual number of distinct hashes in this δ sized portion of the hash domain. The result is then extrapolated from the number of distinct chunks in a small hash domain, to the number of hashes in the entire domain. This relies on the fact that hashes should be evenly distributed over the entire range, and a δ fraction of the domain should hold approximately a δ portion of the distinct hashes.

The problem here is that the Unseen algorithm runs on a substantially smaller fraction of the data than originally. Recall that it was shown in [18]

that accuracy is achieved at a sample fraction of $O(\frac{1}{\log N})$ and therefore we expect the accuracy to be better when N is larger. Indeed, when limiting the input of Unseen to such a small domain (in some of our tests the domain is reduced by a factor of more than $20,000$) then the tightness of the estimation suffers. Figure 6 shows an example of the estimation achieved with this method.

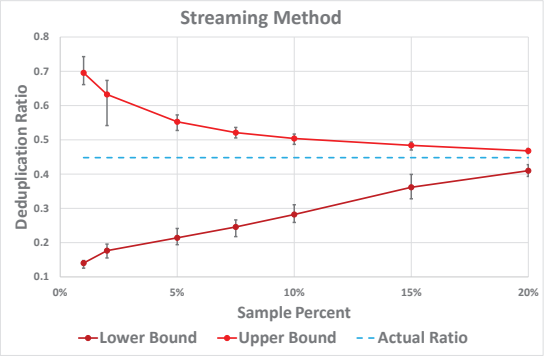


Figure 6: The graph depicts the average upper and lower bounds that are given by the algorithm with the streaming model as a function of sample percentage. Each point is the average over 100 different samples, and the error bars depict the maximum and minimum values over the 100 tests.

In Figure 7 we compare the tightness of the estimation achieved by the two low-memory approaches. Both methods give looser results than the full fledged method, but the base sampling technique is significantly tighter.

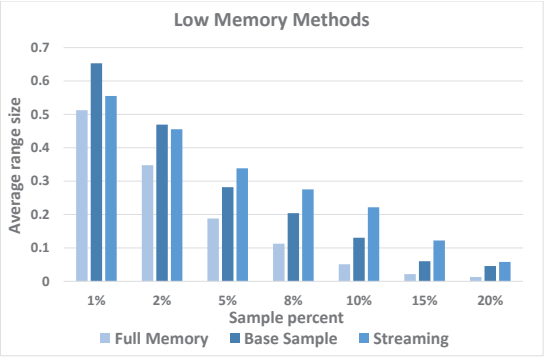


Figure 7: An average range size comparison of the two low-memory methods (and compared to the full memory algorithm). We see that the base method achieves tighter estimation ranges, especially for the higher and more meaningful percentages.

On the flip side, the streaming approach is much simpler to use in parallel environments where each node can run his sample independently and at the end all results are merged and a single Unseen execution is run. Another benefit is that one can run an estimation on a certain set of volumes and store the low-memory histogram. Then, at a later stage, new volumes can be scanned and merged with the existing results to get an updated estimation. Although the streaming approach requires a larger sample in order to reach the same level of accuracy, there are scenarios where the base sample method cannot be used and this method can serve as a good fallback option.

3.3 Estimating Combined Compression and Deduplication

Deduplication, more often than not, is used in conjunction with compression. The typical usage is to first apply deduplication and then store the actual chunks in compressed fashion. Thus the challenge of sizing a system must take into account compression as well as deduplication. The obvious solution to estimating the combination of the two techniques is by estimating each one separately and then looking at their multiplied effect. While this practice has its merits (e.g. see [6]), it is often imprecise. The reason is that in some workloads there is a correlation between the duplication level of a chunk and its average compression ratio (e.g. see Figure 8).

We next describe a method of integrating compression into the Unseen algorithm that results in accurate estimations of this combination.

The basic principle is to replace the DFH by a **compression weighted DFH**. Rather than having x_i hold the number of chunks that appeared i times, we define it as the size (in chunks) that it takes to store the chunks with reference count i. Or in other words, multiply each count in the regular DFH by the average compression ratio of chunks with the specific duplication count.

The problem is that this is no longer a legal DFH and in particular it no longer holds that

$$\sum_i x_i \cdot i = N.$$

Instead, it holds that

$$\sum_i x_i \cdot i = \text{CR} \cdot N$$

where CR is the average compression ratio over the entire dataset (plain compression without deduplication). Luckily, the average compression ratio can be estimated extremely well with a small random sample of the data.

The high level algorithm is then as follows:

1. Compute a DFH $\{y_1, y_2, ...\}$ on the observed sample, but also compute $\{\text{CR}_1, \text{CR}_2, ...\}$ where CR_i is the average compression ratio for all chunks that had reference count i. Denote by $z = \{y_1 \cdot \text{CR}_1, y_2 \cdot \text{CR}_2, ...\}$ the compression weighted DFH of the sample.

2. Compute CR, the average compression ratio on the dataset. This can be done using a very small random sample (which can be part of the already sampled data).
3. Run the Unseen method where the optimization is for $\Delta_p(x,z)$ (rather than $\Delta(x,y)_p$) and under the constraint that $\sum_i x_i \cdot i = \text{CR} \cdot N$.

Figure 8 shows the success of this method on the same dataset and contrasts it to the naive approach of looking at deduplication and compression independently.

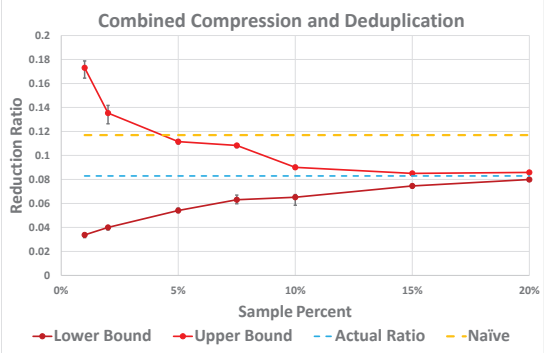


Figure 8: The graph depicts the average upper and lower bounds that are given by the algorithm with compression. Each point is based on 100 different samples. The naive ratio is derived by multiplying independent compression and deduplication ratios.

Note that estimating CR can also be done efficiently and effectively under our two models of low-memory execution: In the base sample method, taking the average compression ratio on the base chunks only is sufficient. So compression needs to be computed only in the initial small base sample phase. In the streaming case, things are a bit more complicated, but in a nutshell, the average CR for the chunks in the fraction of hashes at hand is estimated as the weighted average of the compression ratios of the chunks in the top C hashes, were the weight is their reference counts (a chunk that appeared twice is given double the weight).

3.4 How to Sample in the Real World

Thus far we have avoided the question of how to actually sample chunks from the dataset, yet our sampling has a long list of requirements:

- Sample uniformly at random over the entire (possibly distributed) dataset.
- Sample without repetitions.
- Use low memory for the actual sampling.
- We want the option to do a gradual sample, e.g., first sample a small percent, evaluate it, and then add more samples if needed (without repeating old samples).
- Above all, we need this to be substantially faster than running a full scan (otherwise there is no gain). Recall that the scan time dominates the running time (the time to solve the linear programs is negligible).

The last requirement is the trickiest of them all, especially if the storage medium is based on rotating disks (HDDs). If the data lies on a storage system that supports fast short random reads (flash or solid state drive based systems), then sampling is much faster than a full sequential scan. The problem is that in HDDs there is a massive drop-off from the performance of sequential reads to that of small random reads.

There are some ways to mitigate this drop off: sorting the reads in ascending order is helpful, but mainly reading at larger chunks than 4KB, where 1MB seems the tipping point. In Figure 9 we see measurements on the time it takes to sample a fraction of the data vs. the time a full scan would take. While it is very hard to gain anything by sampling at 4KB chunks, there are significant time savings in sampling at 1MBs, and for instance, sampling a 15% fraction of the data is 3X faster than a full scan (this is assuming sorted 1MB reads).

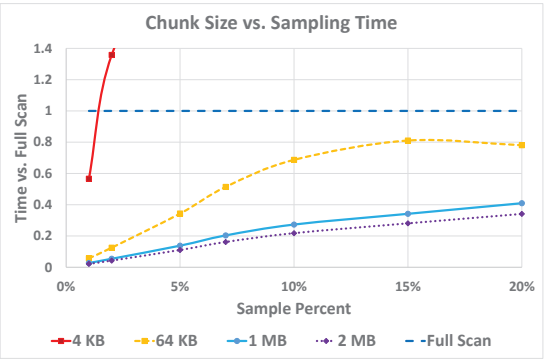


Figure 9: The graph depicts the time it takes to sample different percentages at different chunk sizes with respect to a full scan. We see that sampling at 4K is extremely slow, and while 64K is better, one has to climb up to 1MB to get decent savings. Going higher than 1MB shows little improvement. The tests where run on an Intel Xeon E5440 @ 2.83GHz CPU with a 300GB Hitachi GST Ultrastar 15K rpm HDD.

Accuracy with 1MB reads? The main question is then: does our methodology work with 1MB reads? Clearly, estimating deduplication at a 1MB granularity is not a viable solution since deduplication ratios can change drastically with the chunk size. Instead we read super-chunks of 1MB and break them into 4KB chunks and use these correlated chunks for our sample. The main concern here is that the fact that samples are not independent will form

Name	*Description*	*Size*	*Deduplication Ratio*	*Deduplication + Compression*
VM Repository	A large repository of VMs used by a development unit. This is a VMWare environment with a mixture of Linux and Windows VMs.	8TB	0.3788	0.2134
Linux Hypervisor	Backend store for Linux VMs of a KVM Hypervisor. The VMs belong to a research unit.	370 GB	0.4499	0.08292
Windows Hypervisor	Backend store for Windows VMs of a KVM Hypervisor. The VMs belong to a research unit.	750 GB	0.7761	0.4167
VDI	A VDI benchmark environment containing 50 Windows VMs generated by VMWare's ViewPlanner tool	770 GB	0.029	0.0087
DB	An Oracle Data Base containing data from a TPCC benchmark	1.2TB	0.37884	0.21341
Cloud Store	A research unit's private cloud storage. Includes both user data and VM images.	3.3TB	0.26171	–

Table 1: Data generation approaches for several widely adopted benchmarks in various storage domains.

high correlations between the reference counts of the various chunks in the sample. For example, in many deduplication friendly environments, the repetitions are quite long, and a repetition of a single chunk often entails a repetition of the entire super-chunk (and vice-versa, a non repetition of a single chunk could mean high probability of no repetitions in the super-chunk).

The good news is that due to linearity of expectations, the expected DFH should not change by sampling at a large super-chunk. On the other hand the variance can grow significantly. As before, we control this by increasing the slackness parameter α to allow a larger scope of plausible solutions to be examined. In our tests we raise the slackness from $\alpha = 0.5$ to $\alpha = 2.0$ and if combined with the base sample it is raised to $\alpha = 3.5$. Our tests show that this is sufficient to handle the increased variance, even in workloads where we know that there are extremely high correlations in the repetitions. Figure 10 shows the algorithm result when reading with 1MB super-chunks and with the increased slackness.

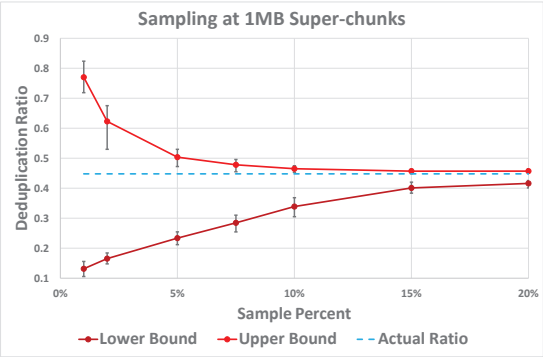


Figure 10: The graph depicts the average upper and lower bounds that are given by the algorithm when reading at 1MB super-chunks. Each point is based on 100 different samples.

How to sample? We next present our sampling strategy that fulfills all of the requirements listed above with the additional requirement to generate *sorted* reads of a configurable chunk size. The process iterates over all chunk IDs in the system and computes a fast hash function on the chunk ID (the ID has to be a unique identifier, e.g. volume name and chunk offset). The hash can be a very efficient function like CRC (we use a simple linear function modulo a large number). This hash value is then used to determine if the chunk is in the current sample or not. The simple algorithm is outlined in Algorithm 3.

Algorithm 3: Simple Sample

Input: Fraction bounds p_0, p_1, Total chunks **M**
Output: Sample $\mathcal{S}_{(p_1-p_0)}$

```
for j ∈ [M] do
    q ⟵ FastHash(j)
    /* FastHash outputs a number in [0, 1) */
    if p_0 ≤ q < p_1 then
        Add j^th chunk to sample
```

This simple technique fulfills all of the requirements that we listed above. It can be easily parallelized and in fact there is no limitation on the enumeration order. However, within each disk, if one iterates in ascending order then the reads will come out sorted, as required. It is nearly stateless, one only need to remember the current index j and the input parameters. In order to run a gradual sample, for example, a first sample of 1% and then add another 5% – run it first with $p_0 = 0, p_1 = 0.01$ and then again with $p_0 = 0.01, p_1 = 0.06$. The fast hash is only required to be sufficiently random (any pairwise independent hash would suffice [3]) and there is no need for a heavy full-fledged cryptographic hash like SHA1. As a result, the main loop can be extremely fast, and our tests show that the overhead of the iteration and hash is negligible (less than 0.5% of the time that it takes to sample a 1% fraction of the dataset). Note that the result is a sample of fraction *approximately* $(p_1 - p_0)$ which is sufficient for all practical means.

4 Evaluation

4.1 Implementation and Test Data

We implemented the core techniques in Matlab and evaluated the implementation on data from a variety of real life datasets that are customary to enterprise storage systems. The different datasets are listed in Table 1 along with their data reduction ratios. It was paramount to tests datasets from a variety of deduplication and compression ratios in order to validate that our techniques are accurate for all ranges. Note that in our datasets we remove all zero chunks since identifying the fraction of zero chunks is an easy task and the main challenge is estimating the deduplication ratio on the rest of the data.

For each of the datasets we generated between 30-100 independent random samples for each sample percentage and for each of the relevant sampling strategy being deployed (e.g., with or without repetitions/ at 1MB super-chunks). These samples serve as the base for verifying our methods and fine tuning the slackness parameter.

4.2 Results

Range sizes as function of dataset. The most glaring phenomena that we observe while testing the technique over multiple datasets is the big discrepancy in the size of the estimation ranges for different datasets. The most defining factor was the data reduction ratio at hand. It turns out that deduplication ratios that are close to $\frac{1}{2}$ are in general harder to approximate accurately and require a larger sample fraction in order to get a tight estimation. Highly dedupable data and data with no duplication, on the other hand tend to converge very quickly and using our method, one can get a very good read within the first 1-2% of sampled data. Figure 12 shows this phenomena clearly.

Note that the addition of compression ratios in the mix has a different effect and it basically reduces the range by a roughly a constant factor that is tied to the compression benefits. For example, for the Windows Hypervisor data the combine deduplication and compression ratio is 0.41, an area where the estimation is hardest. But the convergence seen by the algorithm is significantly better – it is similar to what is seen for deduplication (0.77 ratio) with a roughly constant reduction factor. See example in Figure 13. According to the other dataset we observe that this reduction factor is more significant when the compression ratio is better (as seen in other datasets).

The accumulated effect on estimation tightness. In this work we present two main techniques that are critical enablers for the technology but reduce

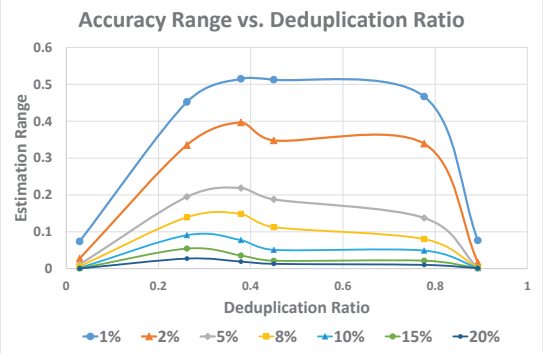


Figure 12: The graph depicts the effect that the deduplication ratio has on the tightness of our estimation method (based on the 6 different datasets and their various deduplication ratios). Each line stands for a specific sample percent and charts the average range size as a function of the deduplication ratio. We see that while the bottom lines of 10% and more are good across all deduplication ratios, the top lines of 1-5% are more like a "Boa digesting an Elephant" – behave very well at the edges but balloon in the middle.

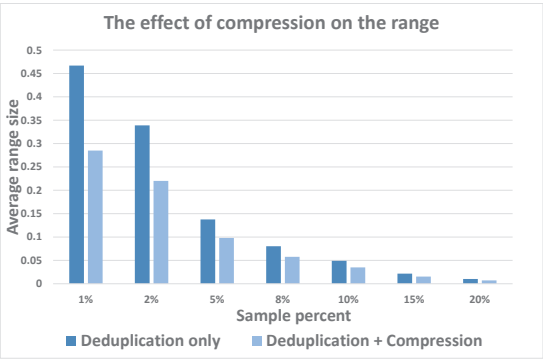


Figure 13: A comparison of the estimation range sizes achieved on the Windows Hypervisor data. We see a constant and significantly tighter estimation when compression is involved. This phenomena holds for all datasets.

the tightness of the initial estimation. The accumulated effect on the tightness of estimation by using the combination of these techniques is shown in Figure 14. It is interesting to note that combining the two techniques has a smaller negative effect on the tightness than the sum of their separate effects.

Putting it all together. Throughout the paper we evaluated the effect of each of our innovations separately and sometimes understanding joint effects. In this section we aim to put all of our techniques together and show a functional result. The final construction runs the *Range Unseen* algorithm, with the *Base Sample* low-memory technique while sampling at *1MB super-chunks*. We test both estimating deduplication only and estimation with compression. Each test consists of a single gradual execu-

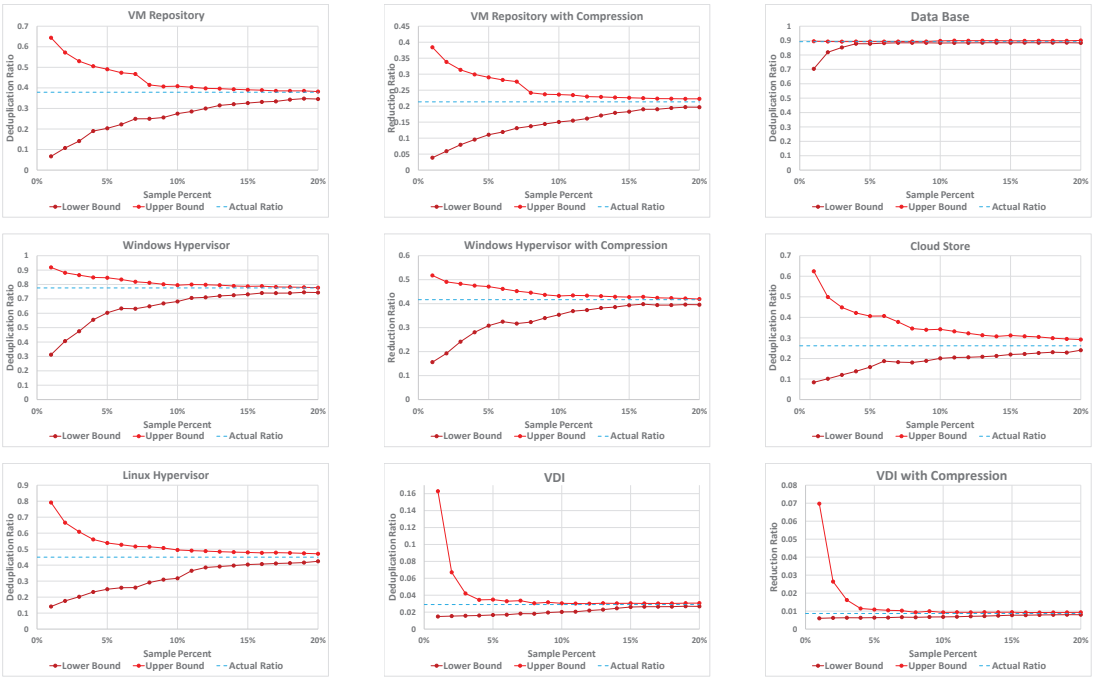
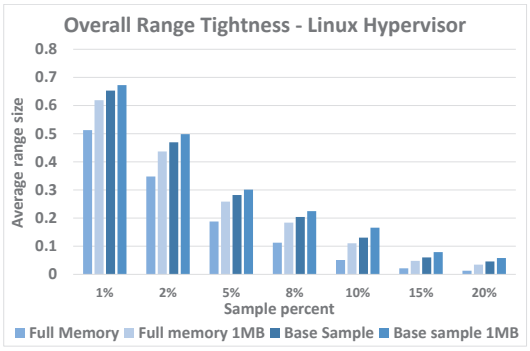


Figure 11: Execution of the full real life setting algorithm on the various datasets. This is a gradual run of the low-memory, 1MB read algorithm, with and without compression. Note that some of the tests reach very tight results with a 3% sample and can thus achieve a much improved running time.



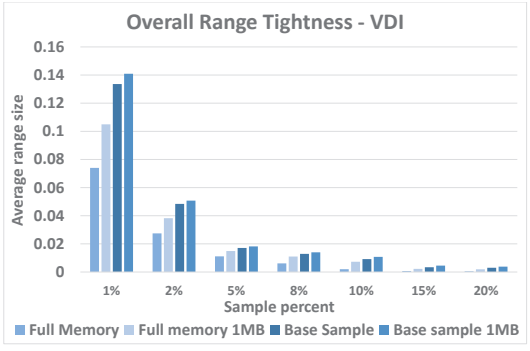


Figure 14: A comparison of the estimation range sizes achieved with various work methods: The basic method, the method with sampling at 1MB, using the base sample low-memory technique and the combination of both the base sample and 1MB sampling. There is a steady decline in tightness when moving from one to the next. This is shown on two different datasets with very different deduplication ratios.

tion starting from 1% all the way through 20% at small intervals of 1%. The results are depicted in Figure 11.

There are two main conclusions from the results in Figure 11. One is that the method actually works and produces accurate and useful results. The second is the great variance between the different runs and the fact that some runs can end well short of a 5% sample. As mentioned in the previous paragraph, this is mostly related to the deduplication and compression ratios involved. But the bottom line is that we can calibrate this into the expected time it takes to run the estimation. In the worst case, one would have to run read at least 15% of the data, which leads to a time improvement of approximately 3X in HDD systems (see Section 3.4). On the other hand, we have tests that can end with a sample of 2-3% and yield a time saving of 15-20X over a full scan. The time improvement can be even more significant in cases where the data resides on SSDs and if the hash computation is a bottle neck in the system.

5 Concluding remarks

Our work introduced new advanced algorithms into the world of deduplication estimation. The main challenges were to make these techniques actually applicable and worthwhile in a real world scenario. We believe we have succeeded in proving the value of this approach, which can be used to replace full scans used today.

Acknowledgements. We thank Evgeny Lipovetsky for his help and thank Oded Margalit and David Chambliss for their insights. Finally, we are grateful to our many colleagues at IBM that made the collections of the various datasets possible. This work was partially funded by the European Community's Seventh Framework Programme (FP7/2007-2013) project ForgetIT under grant No. 600826.

References

[1] XtremeIO case studies. `http://xtremio.com/case-studies`.

[2] BAR-YOSSEF, Z., JAYRAM, T. S., KUMAR, R., SIVAKUMAR, D., AND TREVISAN, L. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002* (2002), pp. 1–10.

[3] CARTER, L., AND WEGMAN, M. N. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2 (1979), 143–154.

[4] CHARIKAR, M., CHAUDHURI, S., MOTWANI, R., AND NARASAYYA, V. R. Towards estimation error guarantees for distinct values. In *Symposium on Principles of Database Systems, PODS 2010* (2000), pp. 268–279.

[5] CONSTANTINESCU, C., GLIDER, J. S., AND CHAMBLISS, D. D. Mixing deduplication and compression on active data sets. In *Data Compression Conference, DCC* (2011), pp. 393–402.

[6] CONSTANTINESCU, C., AND LU, M. Quick Estimation of Data Compression and Deduplication for Large Storage Systems. In *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing* (2011), IEEE, pp. 98–102.

[7] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31*, 2 (1985), 182–209.

[8] GIBBONS, P. B. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Very Large Data Bases VLDB* (2001), pp. 541–550.

[9] HAAS, P. J., NAUGHTON, J. F., SESHADRI, S., AND STOKES, L. Sampling-based estimation of the number of distinct values of an attribute. In *Very Large Data Bases VLDB'95* (1995), pp. 311–322.

[10] HAAS, P. J., AND STOKES, L. Estimating the number of classes in a finite population. *IBM Research Report RJ 10025, IBM Almaden Research 93* (1998), 1475–1487.

[11] HARNIK, D., KAT, R., MARGALIT, O., SOTNIKOV, D., AND TRAEGER, A. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST 2013)* (2013), USENIX Association, pp. 229–241.

[12] HARNIK, D., MARGALIT, O., NAOR, D., SOTNIKOV, D., AND VERNIK, G. Estimation of deduplication ratios in large data sets. In *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012* (2012), pp. 1–11.

[13] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference, SYSTOR 2009* (2009).

[14] KANE, D. M., NELSON, J., AND WOODRUFF, D. P. An optimal algorithm for the distinct elements problem. In *Symposium on Principles of Database Systems, PODS 2010* (2010), pp. 41–52.

[15] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *FAST-Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)* (2011), pp. 1–13.

[16] RASKHODNIKOVA, S., RON, D., SHPILKA, A., AND SMITH, A. Strong lower bounds for approximating distribution support size and the distinct elements problem. *SIAM J. Comput. 39*, 3 (2009), 813–842.

[17] VALIANT, G., AND VALIANT, P. Estimating the unseen: an n/log(n)-sample estimator for entropy and support size, shown optimal via new clts. In *43rd ACM Symposium on Theory of Computing, STOC* (2011), pp. 685–694.

[18] VALIANT, P., AND VALIANT, G. Estimating the unseen: Improved estimators for entropy and other properties. In *Advances in Neural Information Processing Systems 26*. 2013, pp. 2157–2165.

[19] XIE, F., CONDICT, M., AND SHETE, S. Estimating duplication by content-based sampling. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, pp. 181–186.