The Case for Sampling on Very Large File Systems

George Goldberg, Danny Harnik, Dmitry Sotnikov
IBM Research–Haifa, {georgeg, dannyh,dmitrys}@il.ibm.com

***Abstract*—Sampling has long been a prominent tool in statistics and analytics, first and foremost when very large amounts of data are involved. In the realm of very large file systems (and hierarchical data stores in general), however, sampling has mostly been ignored and for several good reasons. Mainly, running sampling in such an environment introduces technical challenges that make the entire sampling process non-beneficial. In this work we demonstrate that there are cases for which sampling is very worthwhile in very large file systems. We address this topic in two aspect: (a) the technical side where we design and implement solutions to efficient weighted sampling that is also distributed, one-pass and addresses multiple efficiency aspects; and (b) the usability aspect in which we demonstrate several use-cases in which weighted sampling over large file systems is extremely beneficial. In particular, we show use-cases regarding estimation of compression ratios, testing and auditing and offline collection of statistics on very large data stores.**

I. Introduction

The use of sampling to obtain meaningful and accurate statistics is abundant in many aspects of our lives and has been methodically used for at least two centuries. Polls over large populations, scientific experiments and physical measurements all build open the power of random sampling to estimate figures in cases where going over the entire space at hand is simply too large or too expensive to be practical. In the realm of storage, sampling has typically been used in data bases as a means to expedite some of the analytics at hand.

In file systems, however, sampling has largely been avoided. This is true also for extremely large instances where the shear size of the file system begs to considering means for speeding up the process of acquiring analytics on the data. It also holds for other hierarchical heterogenous data stores, such as object storage. This avoidance is not by chance as there are multiple reasons and challenges that make sampling in this setting both harder and less beneficial. In particular, the following issues arise when attempting sampling based estimations:

- While sampling a random subset out of a flat space of elements is straightforward (e.g. sampling a block out of a volume), a file system does not allow true random access to the data since the files lie in a complex hierarchical structure and the access to the data is via a directory tree.
- The hierarchical structure of file systems or object storage and the high variance in sizes of files, objects and directories, can cause a situation in which the bulk amount of data in the system resides only in a small portion of the files. As a result, one has to scan the metadata of the whole repository in order to account for all entirety of the data. In a file system this amounts to a costly full directory tree traversal.
- The cost of a full directory traversal on a large file system is prohibitively high and in addition such a traverse allows to collect accurate statistics on metadata, therefore, leaving hardly any benefit in doing an estimation using sampling.

The aforementioned issues create a situation by which it is unclear that there is any point in using random sampling for analytics in the setting of very large file systems.

A. Our Contributions

In this work we argue the case for using sampling as a mean for achieving meaningful analytics on very large hierarchical data stores and file systems in particular. Our contributions address two main concerns in doing so.

1) The first is the technical aspect of sampling. Formalizing the correct sampling distribution in such systems, devising and implementing algorithms that actually perform this sampling with minimal overhead. We produce a general interface that can be called during any traversal of the metadata and produce a short list of sampled files according to the desired distribution. Our design addresses the challenges of running in a single pass and in a multi-threaded distributed environment. These are crucial features since the performance of the traverse is highly influenced by running in a distributed manner.
2) The second concern is in identifying use-cases for which sampling is actually beneficial. Our use-cases include processes that require deep data inspection (such as estimation of compression ratios), testing and auditing and offline collection of statistics on very large file systems.

We note that our framework is general and its usability transcends far beyond the realm of file systems and is relevant to almost any storage form, such as object stores, archives and others. Yet, in this paper, we mostly focus on our implementation for the important use-case of very large scale file systems (such as Network Attached Storage or a clustered file system).

B. Use Cases

a) Scope of the analytics:: The fact that we do sampling limits the types of analytics that we can successfully support. Still there is a large and useful scope of analysis that can be achieved based on sampling. In general, we support any measure that can be described as an average or sum over *local*

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.

978-1-4799-5671-5/14/$31.00

tests, where local means a function that can be computed at a relatively small granularity of the data set, such as on single files or on parts of files. Our guarantees are statistical and assure that any phenomena that is noticeable in the file system will be detected on the sample with very high probability (the exact parameters are tied to the sample size). We describe these concepts more formally in Appendix A.

b) The benefits of sampling: Random sampling is beneficial in that it allows one to inspect far less data during the analysis. That being said, there are settings where the actual benefits are not always clear. As mentioned above, the sampling process likely entails iterating all elements in the data set (full directory tree traversal), and during such a pass many of the statistics can be calculated with negligible overhead. So, for example, in order to answer a query like "what is the total size of *jpeg* images in the file system" a full traversal can simply keep a counter and answer this query accurately. On the other hand, there are other cases that make sampling very attractive and these are the focus of our work. For example, if the query involved requires to actually read the data in the files from the disk, then collecting this data is far more strenuous than running a traversal and may turn a feasible task into an infeasible full data scan.

We describe several specific examples of the benefits of our framework. These are just the tip of the iceberg in terms of applications:

1) Estimating compression ratios: Compression for file systems is gaining popularity and there are several such offerings today of built-in compression for large scale file systems (e.g. [22], [20], [15]). A tool for predicting the benefit of using such compression on a (currently) uncompressed file system would be highly beneficial. Since migration of a very large file system into compressed form is prohibitively lengthy time wise and taxing in terms of CPU usage (both during compression and during de-compression), it is paramount to understand how much there is to gain before deciding to go forward with the migration. Moreover, accurate estimation of this compression ratio can prove valuable financially through tighter capacity planning of a compressed system. Compression estimation via sampling was recently studied in [11], but this work fell short of providing a practical implementation for large scale file systems (but rather focused on block interface or a given list of objects). In our work we implement an efficient distributed *traversal with sampling* tool that culminates by running a compression estimation based on the chosen sample. The main appeal of this method is the minimal amount of actual data that needs to be read from disk for the estimation, yet it achieves sound accuracy guarantees. The performance is directly tied to the time of the traversal with a very small overhead for sampling and a constant overhead of less than a minute required to read the bits of the chosen sample files from disk and evaluate compression on them. In Section IV we present accuracy and performance tests of our implementation. For example, we accurately estimate the compression ratio on 1.8TB by reading only 320 MB of data while requiring only 3% the running time of highly distributed exhaustive evaluation. Our solution is currently being integrated as a customer evaluation and sizing tool for a prominent compression product, and in fact this application was the original motivation for this study.

2) Testing and auditing mechanisms: In the previous example the bottleneck was mainly reading data from the disk (and also compressing it). In other scenarios the amount of data might not be excessive, yet the processing that it has to undergo is very heavy. Examples are various analysis and learning algorithms such as video processing, or clustering algorithms. In such an environment minimizing the set of files that require processing is a valuable optimization.

One example is for auditing that requires heavy computation. Sampling has always played a key role in auditing, and for example, checking that TV content of certain types (e.g., advertisements) does not exceed a certain percentage of the airing time may be difficult to achieve exhaustively and can benefit greatly from sampling. Another area of relevance is in testing the success of algorithms. For example, one use case that we have explored is testing of speech to text algorithms. Given several algorithms and a large data set, the choice of the best algorithm is not clear and typically the only way to verify the result of such a translation algorithm is for a human being to listen and translate it. So the bottleneck here is not the text to speech algorithm but rather the amount of data that a human reviews. In this use case the files are weighted according to their recording length (counted either by time or by the number of recorded words) and the sample allows a human reviewer to review a fixed number of recorded time units and produce an estimation on the success rate of a speech to text algorithm. Note that for the specific task of testing success rate of algorithms, our estimation turn out to be especially tight for highly successful algorithms. This is since the variance of a very successful algorithm is very small (see discussion in Section B).

3) Offline analysis of file system distributions:: The use cases thus far consider situations in which the full directory traversal was not the bottleneck in the system but rather it is the data reading and processing which is the heaviest task. In this use case we consider the benefits in situations where the traversal is the heaviest link in the chain. There are several methods to deal with analytics on such large directories, for example by distribution of the traverse [17], or by doing only a partial traverse [13] (an approach that may jeopardize accuracy but works well in many cases). The shortcoming of these approaches is that the queries at hand needs to be defined before the traverse is run in order to collect the right statistics during the traverse. Alternatively one can collect all of the metadata for each of the millions of records and query this metadata offline (e.g. by putting all of the metadata in a separate database), but for very large systems this may turn out to be an excessive overhead. Instead our mechanism complements any fast traverse approach by creating a short "sketch" of the data set, from which one can deduce numerous statistics and queries after the traverse has finished. A traverse of a very large system can take many hours and maybe days

and by analyzing it one can learn various properties such as popular file types, libraries or file properties. In many cases, based on this first analysis one would like to refine the query, but this would require a new traversal to collect new statistics. Using weighted sampling, one can collect a relatively short list of sample files, that are weighted according to key parameters. For example, according to file length or any other relevant measure. Now statistics and measurements can be run on the short list, providing provably accurate results, but without requiring a further traversal of the data set.

We demonstrate this methodology by scanning a real life data set of 17.5 million files, with 7.25 TB of data and analyzing some of its properties offline, finally comparing these results to the accurate full scan numbers. The appeal is that maintaining even a relatively large list (say of 100,000 samples) is still orders of magnitude smaller than managing all of the file system's metadata, yet provides extremely accurate estimations.

C. Related Work

Sampling has long been a central tool in statistics (see, for example [14]). In the storage wold it has mostly been employed in data bases. There are many works that attempt to estimate query sizes using sampling in order to optimize data bases operations. Some examples that consider uniform sampling over data base elements are [9], [10], [8], [5], [3]. In general, the interface provided by data base is much friendlier to sampling distributions than that of file systems. It should be noted though, that also in data bases sampling uniformly can cause difficulties, as pointed out in [4], mainly because the elements their are too fine grained and don't correlate well with the way data is stored in the back end.

In the context of file systems and hierarchical data structures in general, sampling, and analytics in general become harder to achieve. In fact, there is a trend to modify design of file systems to support either a different structure to their metadata handling [16],[19] (and thus achieve faster analytics on metadata) or a different structure altogether [21].

Weighted random sampling was studied in [7] (see also references within). While the one of the sampling forms used here fits our framework, the underlying algorithms differ from ours, and in particular use much more invocations of the randomness function than our technique (see discussion in Section III).

Analytics on very large file systems where studied in [13] and in [17]. Both these works consider only analytics on metadata of the files (as opposed to content, such as the compression example). The first work tackles the long time to do full directory traversal by doing a partial random traverse on the tree (and thus cannot have accuracy guarantees on the statistics). The second work improves the traversal speed by smarter distribution. Note that our work is complementary to both techniques, and can be applied on top of these faster traverse.

Finally, compression estimation was studied in [6] and [11]. The first paper sampled parts of each file which amounts to worst performance than our method and without accuracy guarantees. The second paper lays the foundation for our accuracy guarantees for compression, but do not give suitable interface for working efficiently with file system.

II. Estimation via Sampling - Preliminaries

The main objective of this work is to support estimation via sampling over a large data set. As described in the use-cases section, there is ample motivation to reduce the size of the data at hand which can reduce disk IOs, CPU and time to obtain statistics (to name a few resources). However, randomly sampling files in a file system carries an inherent difficulty because data is typically accessed through a directory tree (some file systems have other means of access to the data, such as i-node traversal, but these too are subject to some of the difficulties that will be described in the following section). The problem with directory trees is that the majority of the files can possibly reside in a single sub-directory, and finding this sub-directory can require a full scan of the directory tree. Even more so, a key observation is that naïve random sampling of files may end up in giving grossly inaccurate approximation. This is due to the fact that file sizes can be very diverse. Research (e.g. [6], [24]) has shown that in many large scale file systems, small files account for a large number of the files but only a small portion of total capacity (in some cases 99% of the files accounted for less then 10% of the capacity). So sampling randomly from the entire list of files may yield a sample set with a disproportional number of small files that actually have very little effect on the overall capacity. It is therefore clear that files should be sampled according to their capacity to ensure sound analytics. The natural approach in this case is to sample files with probability according to their length. Namely, a file of length 2 MB is twice as likely to be chosen than a file of length 1 MB. However, there is a subtlety here - this linearity in probability is for each choice of a file in the sample set (and not for a single choice).

In the following we describe the distribution that we aim for in our sampling - a distribution that allows us to prove statements on the accuracy of the estimation. We complement it with a model for generalizing the type of statistic for which we can achieve accuracy guarantees.

A. Basic notation

Throughout the paper we consider a file system with N files (without loss of generality, this could be elements in any type of data set, such as objects in an object store, or columns in a data base). Each file $i \in \{1, ..., N\}$ is assigned a weight w_i and denote by

$$W = \sum_{i=1}^{N} w_i$$

the total weight in the system. It is instructive to think of the weight as the file size, but this can be generalized to numerous different measures.

B. The sampling distribution

Our goal is to **sample** M **files** in the system with the property that each such sample is taken uniformly at random from all the files when adjusted according to the file's weight w_i. For example if $w_1 = 1000$ and $w_2 = 200$ then each sample is five times more likely to be the first file than second file. Note, however, that this does not mean that the first file is five times more likely to be part of the sample set, since this set contains M points and the ratio of five only holds for a single point out of the M. More precisely, the probability that a file i appears in the sample set has distribution very close to the binomial distribution with M trials and success probability $\frac{w_i}{W}$. Namely, file i gets value $k_i \sim B(M, \frac{w_i}{W})$ (to be exact, our algorithm outputs a multinomial distribution with M trials and N categories with probabilities $\frac{w_1}{W}, \cdots, \frac{w_N}{W}$, a distribution which tends to the binomial distribution collection as N and M grow). Note that the value k_i is an integer (not only 0 and 1) so the i^{th} file is assigned a random variable k_i so that the file i is *not* in the sample if $k_i = 0$ and otherwise appears in the sample k_i times. As a result a single file can appear more than once in the sample and this is very intuitive since in case that a single file is so large that it's capacity is equal, say, $\frac{1}{2}$ of the total capacity, then we would like approximately half of the sample points to belong to this huge file. We refer to the distribution we are sampling as the **weighted multinomial distribution**.

in Appendix A we give a more rigorous account of the statistics that can be obtained via weighted sampling and what are some guarantees that can be obtained.

III. The Sampling Method

So far we have discussed the general framework, its applicability and defined the distribution on files that we aim to sample. This section is devoted to the algorithms and methods that we developed in order to actually carry out the sampling part over a file system (or other data set). Since the distribution is well defined and simple enough, selecting a random subset of files from a given collection is a pretty straightforward programming exercise. However, once we factor in the scale of the data set and the interface to the files, then things become less obvious and need to be designed more carefully. In this respect our design and implementation attempt to optimize all the factors involved in the process, although in some use-cases this may be an overshoot since some of the resources that we optimize for could be abundant and do not pose any issue. Still, there are other cases, for example when the directory tree is in the cache or metadata is managed in fast SSDs in which the performance of the sampling tools may become a bottleneck if not designed carefully (see Section IV-B for such an example). Since our work aims to address as wide as possible scope, we addressed all resources to the best of our ability. In the following section we describe the interface we provide along with a list of key requirements and considerations of the sampling process.

A. The Interface and Key Considerations

The interface of the sampling process should interleave with a full traversal of the data-set at hand (in our examples this is a full directory tree traverse. We implement three processes that can be called by any traverse of iterator over files:

1) **Init sample:** Initialize data structures (called once at the beginning of the traverse).
2) **Update sample:** Called for each file during the traverse, takes as input the file's weight its id and any additional metadata that is available. This process is *thread safe* and can be called by multiple processes in parallel.
3) **Post process:** Finalize and output sample list.

In general, it is expected that the time to run the traversal will dominate the time of the sampling process. However, we attempt to make the overhead of the sampling process as minimal as possible. The philosophy applies for the main resources of memory and CPU utilization. Other note worthy considerations include:

- Randomness - choosing a random distribution naturally requires the use of random generation. While this is not a heavy resource, we observe that it is a limiting one when multiple processes are run simultaneously (see Section III-B). Thus we attempt to use only as much randomness as actually required.
- Multiprocess locks and communication - multiprocessing is crucial enabler for improved directory traversals (see [17])) and therefore supporting multiple threads is a necessity. We want our support of multi-processing to have as little impact as possible and avoid a noticeable slow down due to synchronization of multiple processes.
- Cache awareness - another critical factor in the speed of a traversal is the contents of its cache. As seen in Section IV if the directory tree or parts of it are in cache this enables tremendous speed-up. It is crucial that the sampling (or analytics) component will refrain from extensive memory usage to avoid evicting such useful data from cache.

B. The Core Technique

Since the sampling process is part of an iteration over file and accept one file at a time, the first natural approach is to make an independent decision on a per file basis of whether the file should or should not be included in the sample (and if yes, then how many times should it be in the sample). The problem is that in order to generate a binomial distribution per each file, one needs to flip a biased coin several times per file (assuming the correct bias is known in advance). There are several ways to approach this, but the bottom line is that the number of coin flips (calls to the randomness function) needs to be at least the total weight W of the file system. If the weight is the length of a file, the best optimization would be to count the length in chunks of size 4KB (typical page size of modern file systems). So the number of randomness calls ends up at around the number of 4KB chunks in the entire file system. Such a large number of randomness calls may be very taxing on the system (see Figure 2 in Section IV).

Instead we take a different approach, that will require exactly M calls to the randomness function (recall that M is the number of samples). We describe this core technique now, under the assumption that W is known before hand. In Section III-C we expand our implementation to the case that W is not known in advance.

Our technique first does a pre-processing step that picks the chosen random "positions" for the sample before hand. During the traverse, we map our chosen points onto actual files, as we encounter them. In a nutshell, the pre-processing considers the entire file system as one flat sequential space spanning the range between 1 and W and chooses M random locations inside this range (simply by calling the randomness function M times). When the actual traverse is run, a counter is maintained of how much capacity has been encountered so far (by all files seen up to this point of the traverse). When a new file arrives, the counter is advanced according to the file size, say, for example from point D to point $D+w$. Now if any of the M pre-chosen sample points happened to reside in the area spanned by this file, then this file is added to the sample. If there was more than one sample point between D and $D+w$ then the file is added several times (according to the number of such hits). Below is a pseudocode that implements the technique.

CoreSample(M, W)**:**
Setup:

1) Randomly pick M numbers between 1 to W. Put them into an array `Sample_Index`
2) Sort the M numbers in `Sample_Index`
3) Maintain a counter D of data bytes scanned so far. Initialize to 0 (should eventually reach W).
4) Maintain a counter k of sample points handled so far. Initialize to 0 (should eventually reach M)

Traverse: goes over all files in the system (any list or order suffices). For each file i in the traverse:

1) Increment D by the w_i (the size of the file)
2) If $D >$ `Sample_Index`$[k]$ then
   a. Add the file to the sample list
   b. k++
   c. If $k = M$ then finish
   d. Else goto 2
3) Else continue to the next file and goto 1

It should be stressed that while the random choices are made before the traverse, the files are determined only during the traversal, and indeed a different order of the traversal will yield a different list of files.

C. Handling Unknown File System Size - One Pass Sampling

Our core technique (as well as the other approaches we mentioned in Section III-B) assumes the knowledge of the total size W of the file system before the process begins. This is a reasonable assumption in many settings, since file systems typically maintain counter of their current size (and number of files) and this counter can be queried directly (for example with the `df` command). However, this counters are kept only for the root of the file system and count all files in the system, while our analytics often would like to focus on a specific sub-tree or family of files (e.g. files from a partial list of the file types). In such cases, the only way to compute W is to actually run the full traverse on the directory at hand, and overhead which is unacceptable (since typically the traverse is the heaviest part of the entire process).

In this section we describe our solution to running with unknown size. It uses the CoreSample technique from Section III-B as its basic building block and employs some conventional methods from the realm of streaming algorithms. The general process is outlined in the following pseudo-code:

Sample(M)

- Let S_1 be an initial segment size with the guarantee that $S_1 < W$
- Let $Samp$ be a list of files in the current sample

First iteration:

1) Call CoreSample$(M, S_1) \rightarrow Samp$
2) Set $S = S_1$

Main while loop (until end of traverse reached):

1) Call CoreSample$(M, S) \rightarrow Samp$ (merge with existing list in $Samp$)
2) If reached traverse end then go to End
3) Else set $S = 2 \cdot S$
4) Randomly dilute $Samp$ by leave M random points (out of $2M$)
5) Goto 1

End:
Dilute $Samp$ to M sample points.

The basic idea is to start with a preliminary size S which should be no larger than the eventual size (one can start with a small capacity, say, even 1 MB). One can run the core technique with parameters S rather than W and receive M sample points from the first S bytes that where encountered during the traverse. One can now continue in the same manner but set the bound S to higher than before. However, there are now more than M sample points collected, but worst yet, the first M points were chosen more densely than the rest (assuming that the range did indeed grow). So before continuing to gather sample points, one needs to dilute the set of points chosen so far. It is possible to run this process of diluting for every new chosen point in the set. However, for simpler implementation, we do it in chunks that we call segments. At each step, the new segment is equal in size to the total of all segments seen thus far (to allow exponential growth). Once a segment is finished, its sample points are first merged with the previous set and then randomly diluted by half.

Note that the above pseudocode is high level and in real implementation there are additional details that need to be handled carefully. One such central point is that it is unlikely

that files will fit exactly into segments and so whenever a file fills up a segment it is split and its remainder would spill over to the next segment. A split file may be chosen to the sample set from two consecutive segments (or from none).

The dilution process takes a number of samples larger than M and picks M random files to keep in the list. Notice that we are careful in the algorithm to merge only lists that have the same density to the samples, and thus simply picking a random subset suffices to give a new sample set with the correct distribution but with lower density. In practice we use the "Reservoir sampling" technique [23] (a technique to pick M samples in the same data structure without having to copy all of the M chosen files to a new data structure).

Since the segment size doubles in every iteration, the segments grow exponentially and the number of iterations remains low. To be exact, our method will perform exactly $\left\lceil \log_2 \frac{W}{S_1} \right\rceil + 1$ calls to the CoreSample procedure. When including the dilution processes, the amount of calls to the randomness function will be no more than $2M \cdot (\left\lceil \log_2 \frac{W}{S_1} \right\rceil + 1)$. Our tests show that running our one-pass algorithm without prior knowledge of the overall size has very small overhead when compared to running the core technique with known size (see Section IV for evaluation).

D. Distributed Sampling - Running with Multiple Processes

Supporting multiple processes was done by means of standard locking mechanisms. To the CoreSample we added a Mutex on the counters D and k. Since the operation at hand is a few simple compare and advance operations, the lock is very quickly released and has little effect on performance. A supposedly more efficient design can hand out different subsegments to different processes and thus reduce the number of mutex calls, but since the underlying operation is extremely fast the simpler designed proved sufficient. The only heavier operation is once segments are exhausted and a new segment is generated (via a new call to CoreSample). In this case the process that exhausted the segment is responsible for generating the new segment and new `Sample_Index` list and does not release the mutex until this operation is completed.

1) The Option for Parallel Execution: The above approach for distribution proves adequate in many cases, but falls short in some scenarios. One is when the data set being studied lies on several mount points that are not accessible from a single server. Another is when the systems scale-up to high-performance computing scales and the distribution can benefit greatly for multiple nodes running the traverse (as opposed to multiple-processes on the same computer). A recent study [17] on distributed traversal claims that there is a major slowdown caused by the communication overhead for synchronizing between the processes in a traverse running on very high scale (on the order of 0.5 PB and higher). Instead they devise a traverse that is mostly run in parallel with much lower communication and synchronization requirements. When using such a traverse, our sampling mechanism becomes problematic since it is centralized and relies on communication and locks.

To support such traverses and parallel sampling, we suggest the following mechanism. Each node participating in the traverse will run it separate sampling process starting with a relatively small segment size (the run on each node can be either multi-process or single thread). At the end of the respective traverses, each node supplies a list of M sample files for its respective files. Finally, as a post processing step, all the lists are merged into a single representative sample list (of length M). The problem is that unlike the merge and dilute operations discussed in Section III-C, the merge here takes lists that can account for very different total capacities and hence the sample points for different nodes where sampled with different densities.

More formally, our aim is to merge ℓ lists $L_1, \cdots, L_\ell$ of M points each representing different total weights $W_1, \cdots, W_\ell$. Our solution is to run a small process that picks randomly how many of the M points will be taken out of each list (according to a multinomial distribution on the lists with respective probabilities $\frac{W_1}{W}, \cdots, \frac{W_\ell}{W}$) and then picking the required number of points uniformly at random from within each of the lists. This process results in a distribution which is identical to that of the basic sampling algorithm.

IV. Evaluation

A. The Test Environment and Data

We implemented our sampling all of the versions listed in Section III in C and we hope to make this code public in the near future. In addition, in order to support our evaluation we implemented a distributed directory traversal (along the lines of [2] with some memory optimizations for handling large scale traverses) and a compression estimation tool (see Section IV-C).

We ran our tests on a number of file systems listed below:

- **Impressions FS:** This synthetic file system consists of approximately 4.9 million files and 1.86 TB of data. We created it using the *Impressions* tools [1] with the augmentation that we replaced dummy files with actual content matching the file type (according to extension). This was done in order to make our compression estimation tests meaningful. The file system resides on an enterprise mid range storage controller connected to the test machine by fiber channel.
- **Project Repository:** This file system consists of real life data from a shared projects repository of a large R&D unit. Due to access control issues, we had access to 17.6 Million files constituting 7.8 TBs of data (out of a total of 30TB in the entire repository). The data resides on a clustered file system and is accessed via NFS. It is made up of 3 different mount points of varying sizes.
- **Compression Collection:** This is a collection of 430 GBs of data (in 21,500 files) from various data types used to benchmark compression related products. Resides on a clustered file system and accessed via NFS.
- **Bloated Repository:** In order to test our code on a much larger scale we simulated a large file system by including

snapshots in the directory tree of part of the project repository. This amounted to traversal of a file system with 220 Million files and 93 TBs of data.

The performance measurements were carried out on a server with a 4 core Intel Xeon CPU X5570 2.93GHz with 8GB of RAM.

Till the end of this section we describe our evaluation of the sampling method itself, and of two of the use-cases (compression estimation and offline analysis).

B. Evaluating the Sampling Process

Our first test is to see the performance of our traverse and sampling on the Impressions FS. We run it with increasing number of threads and see a steady improvement in running time that evens out at around 32 threads. Figure 1 shows the results and emphasizes the dramatic improvement that the traverse has when employing multi-threading.

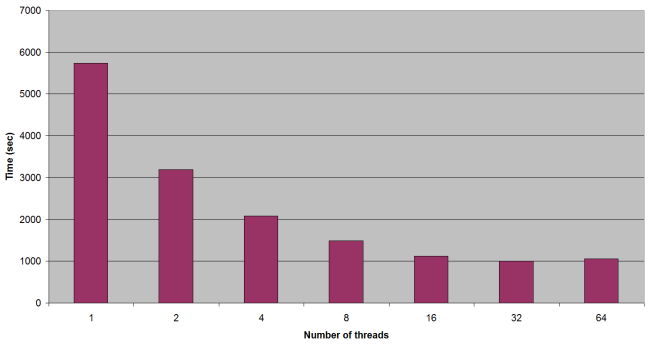

Fig. 1. Running time of a traverse and sample on the Impressions FS as a function of the number of threads.

Cache vs. no cache The running time in the previous test was with a clean cache (we dropped the cache before each execution). This is crucial for fair evaluation since we observed that a far more dramatic improvement in the running time occurs when the directory tree is in the machines cache. Running times can drop to well below one minute as opposed to 15 minutes at best with extensive multi-threading. While this is out of our control when running a traverse at a third party's machine, it gives us an understanding of the fluctuating results that we see in running times.

Running with the directory tree in cache gives very high performance to the traverse and is an ideal setting to test the overhead of our sampling tools on top of the traverse. In Figure 2 we see the running times of the traverse as a stand alone, versus the same traverse with our sampling mechanism on top. We see that the overhead is very small and for almost all points adds up to 10 second slow down. To cap this, we also ran a test that simply adds a calls to the randomness function for every 4 KB block in the file - a test that mimics a simpler approach than ours but requires a linear amount of randomness calls (see discussion in Section III-B). This graph justifies our insistence of minimizing the randomness calls as indeed, we witness a bottleneck in the ability to run intensive multiple invocations of the randomness function.

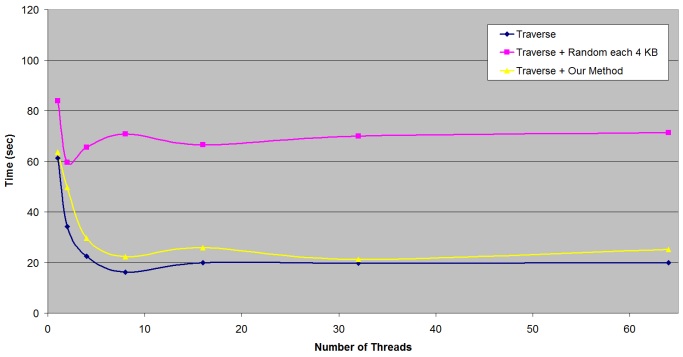

Fig. 2. Comparing the running time of the traverse only vs. the traverse with our sampling and vs. a randomness heavy sampling method. Ran on the Impressions file system with directory tree in cache.

The effect of many segments. We next study the effect of running with an unknown total capacity W (see Section III-C). The effect was unnoticeable in most of the tests that we ran (especially if the cache was turned off). We only managed to see an effect when pushing the process to an extreme - we ran a very fast test (on the Compression Collection, with cache) and took a relatively large number of samples (1,000,000 sample points). Figure 3 depicts the overhead in doing several segment crosses throughout the traverse. As seen the effect is measured in single seconds which become negligible in larger traverses.

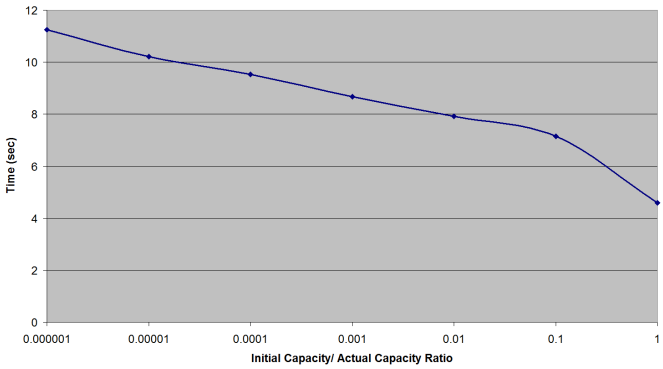

Fig. 3. Running time on the Compression Set with cache enabled as a function of the initial segment size with respect to the total size (namely $\frac{S_1}{W}$ with 1 meaning that the total capacity was known in advance).

C. The Compression Estimation Evaluation

A central use-case for this study was compression ratio estimation. For this we implemented a process that takes as input a file sample list, and for each file occurrence in the list picks a random compression chunk, reads and compresses it to evaluate the chunk's compressibility (if a file appearing more than once in the list, then an additional chunk is chosen and compressed for each occurrence). At the end the process returns the average compression ratio on all of the chunks that it compressed.

In Table I we see the clear benefits of running the estimation process over a full scan of the data. The sampling based estimation reads a mere 320MB of data from disk (irrespective

Data Set	Total Capacity (GB)	Capacity Read for Estimation	Exhaustive Time (sec)	Sampling + Estimation Time (sec)
Impressions	1905 GB	320 MB	49248	1560
Compression	428 GB	320 MB	6032	42

TABLE I
COMPARISON OF EXHAUSTIVE RUN VS SAMPLING FOR COMPRESSION ESTIMATION. BOTH RAN WITH 32 THREADS.

of the entire file system size) and returns an estimation with sound guarantees.

The running times in the table include both the time of the traverse with sampling and the time to read and compress that data in the samples. In Fig 4 we see just the time of the compression estimation parts, without the traversal. This part remains relatively constant, ranging from 4 minutes to less than a minute (depending on the number of threads). In case of a large traverse this is overhead is negligible, for a quick traverse it may become the heavier part of the estimation. Other than multi-threading, the best way to shorten this part is to reduce the number of samples (at the cost of reduced accuracy).

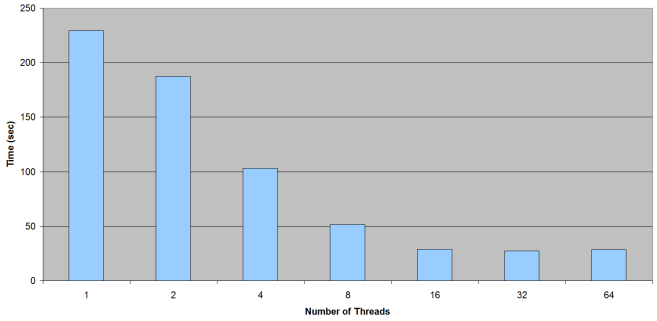

Fig. 4. Running time of the compression estimation part of the process (without the traverse) with 10,000 samples.

We next evaluate the accuracy of the estimation. For this we ran the estimation on the Impressions FS for 300 independent times (with fresh random choices). The results (in Figure 5) are very well centered around the actual compression ratio in the form of a normal distribution. We see that the maximum skew on 300 runs was 1.1% and the overwhelming majority of the runs returned a far better estimation. In terms of applicability, this is an excellent estimation, well enough to make all necessary compression related decisions.

D. Evaluating the Offline Analysis Use-case

In this section we demonstrate some of the applicability of our sampling mechanism as a tool for low memory and resources offline analytics tool to understanding trends and distributions within the file system. To this end, we ran our traverse and sampling over the *Project Repository* and the *Bloated Repository*, collecting a list of 100,000 samples from each of them. Note that this is well below the overall number of files in the repository. The traverse actually returned two lists - one weighted according to file length (for capacity based analytics) and the other with each file getting weight 1 (for analytics on the number of files). Note that we collected the

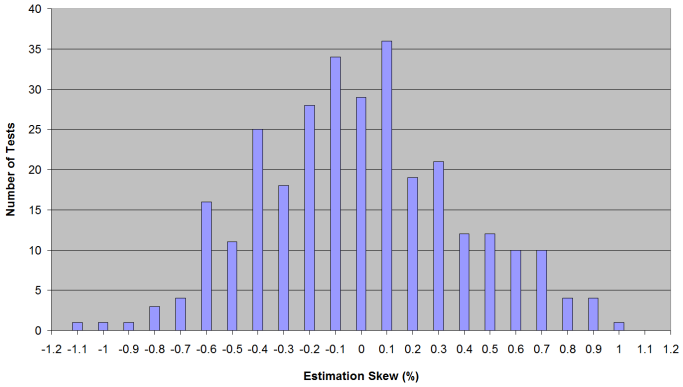

Fig. 5. The histogram depicts how many tests out of 300 independent runs fell in each skew range.

samples separately on each of the three mount points of this file system and merged them using the method described in Section III-D1.

A first test was to identify the popular file extensions according to the capacity sample, and compare their estimated capacities to ones that where exhaustively collected. The results, are depicted in Figure 6.

Another test was to learn the compression ratios of each of these file types separately. As long as a file type was popular enough, we have enough sample points to provide guarantees on the accuracy of the compression figure that we attach to it. Examples of some of the results are in Figure 7.

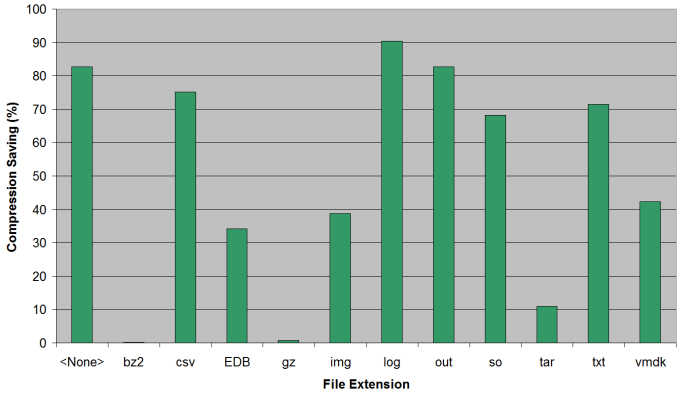

Fig. 7. Compression ratios for various extension types in the Projects Repository.

Figure 8 depicts the capacities of files for different depths in the directory tree (this was run on one of the 3 mount points of the Projects Repository file system). We see that the estimation managed to get extremely accurate results with 100,000 sample points. We ran the same experiment also with 10,000 and 1000 sample points. Naturally, the skew in this cases grew, but interestingly even for 1000 sample points all

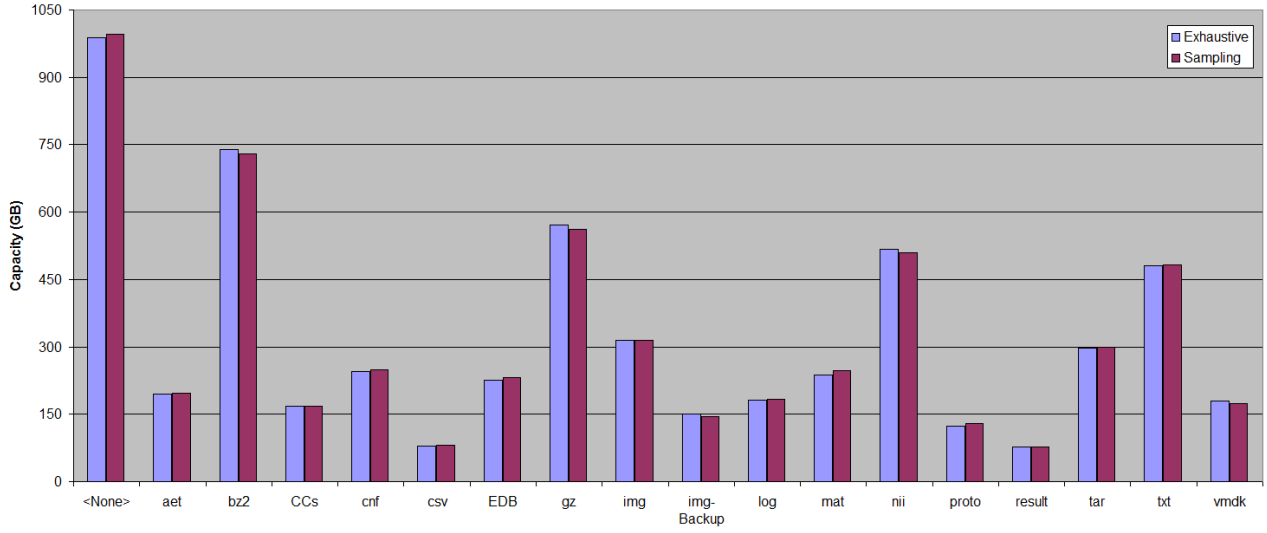

Fig. 6. Capacities of popular file types in the Projects Repository according to the estimation and the actual numbers from an exhaustive evaluation.

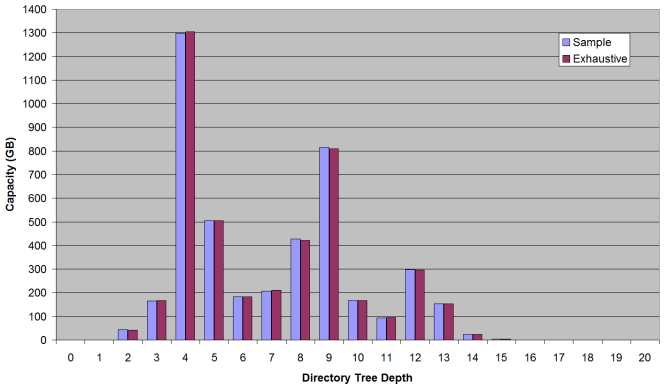

Fig. 8. The distribution of capacity across directory tree depth. Exhaustive vs. offline analysis on 100,000 samples.

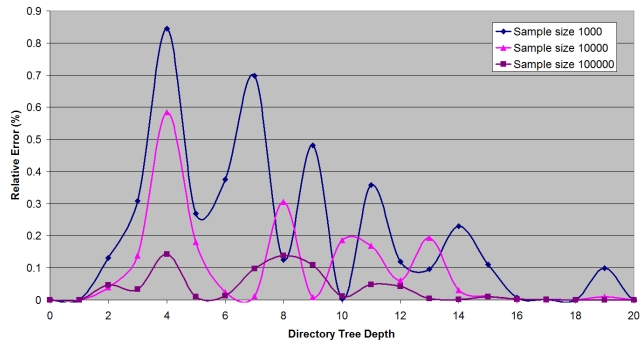

Fig. 9. The skew of estimation of capacities per depth for different sample sizes.

of the estimations where within 1% of the actual value. This is shown in Figure 9

Finally we run a test over the Bloated Projects directory tree (93TB, 220 Million files). The running time of the traverse on this large set was over 70 hours. We managed to test for numerous properties simple by looking at the file sample list. Perhaps the most interesting was looking at snapshots estimated sizes over time. All of the snapshots taken in the past 2 weeks showed consistent size of the project repository during this time. However, 2 snapshots taken over a year ago show that the repository has grown by approximately 46% in this time frame. See Figure 10 for details.

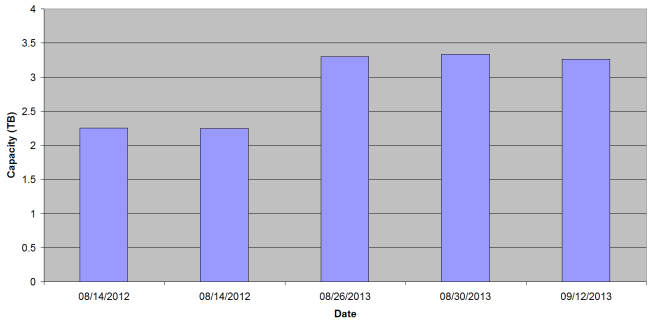

Fig. 10. Snapshots capacity evolution on the Bloated Repository file system.

V. Conclusions

In this work we explored applications of sampling based estimations in the domain of file systems (and hierarchical data stores in general). Employing the sampling methodology in this realm brought new challenges that needed to be addressed in order to make it practical for usage in very large scale file systems. We view our tools as an easy entry point for performing analytics in various settings and believe that the applications presented in the paper are just the tip of the iceberg in terms of usability of the weighted sampling paradigm in hierarchical data stores.

REFERENCES

[1] N. Agrawal, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *7th USENIX Conference on File and Storage Technologies, (FAST '09)*, pages 125–138, 2009.

[2] B. Awerbuch and R. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Trans. Inf. Theor.*, 33(3):315–322, May 1987.

[3] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, (PODS '00)*, pages 268–279, 2000.

[4] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 287–298, 2004.

[5] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 436–447, 1998.

[6] C. Constantinescu and M. Lu. Quick Estimation of Data Compression and De-duplication for Large Storage Systems. In *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing*, pages 98–102. IEEE, 2011.

[7] P. Efraimidis. Weighted random sampling over data streams. *CoRR*, abs/1012.0256, 2010.

[8] P. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 466–475, 1997.

[9] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 341–350, 1992.

[10] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases*, pages 311–322, 1995.

[11] D. Harnik, R. Kat, O. Margalit, D. Sotnikov, and A. Traeger. 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)*, pages 229–241. USENIX Association, 2013.

[12] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 301(58):13–30, 1963.

[13] H. Huang, N. Zhang, W. Wang, G. Das, and A. Szalay. Just-in-time analytics on large file systems. In *9th USENIX Conference on File and Storage Technologies (FAST 2011)*, pages 217–230, 2011.

[14] J. Jiang. *Large Sample Techniques for Statistics*. Springer Texts in Statistics. Springer, 2012.

[15] D. Kay. Oracle Solaris ZFS Storage Management. Technical Report 507914, Oracle Corporation, November 2011.

[16] J. Koren, A. Leung, Y. Zhang, C. Maltzahn, S. Ames, and E. Miller. Searching and navigating petabyte-scale file systems based on facets. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07)*, pages 21–25, 2007.

[17] Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. On distributed file tree walk of parallel file systems. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12*, page 87, 2012.

[18] R.J. Larsen and M.L. Marx. *An Introduction to Mathematical Statistics and Its Applications*. Student solutions manual. Prentice Hall PTR, 2011.

[19] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *7th USENIX Conference on File and Storage Technologies, (FAST '09)*, pages 153–166, 2009.

[20] S. Moulton and C. Alvarez. NetApp Data Compression and Deduplication Deployment and Implementation Guide: Data ONTAP Operating in Cluster-Mode. Technical Report TR-3966, NetApp, June 2012.

[21] Margo I. Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems*, 2009.

[22] R. Tretau, M. Miletic, S. Pemberton, T. Provost, and T. Setiawan. Introduction to IBM Real-time Compression Appliances. Technical Report SG24-7953-01, IBM, January 2012.

[23] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[24] Brent Welch and Geoffrey Noer. Optimizing a hybrid ssd/hdd hpc storage system based on file size distributions. In *IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013*, pages 1–12, 2013.

APPENDIX

A. The statistics of interest

Weighted sampling is described above as our goal, but it is in fact just a crucial stepping stone to achieve the real goal which is analytics on big data. We next define a family of functions on data that are suitable for estimation via sampling.

Definition 1. *Let $f(i,j)$ be a measure defined on a file $i \in \{1,\cdots,N\}$ of weight w_i and position $j \in \{1,\cdots,w_i\}$ and let $W = \sum_{i=1}^{N} w_i$. The* **weighted average** *of f is defined as*

$$F = \frac{1}{W}\sum_{i=1}^{N}\sum_{j=1}^{w_i} f(i,j)$$

Or in words, the statistic F is a sum of a function that can be computed locally on files (of course this can be generalized to a function over sets of files, or over any abstract element in a data set). Some examples of this abstract notion follow:

- Query 1: what is the total capacity of files with type 'jpeg'? The weight w_i is equal to the file length and $f(i,j)$ is set to 1 if the i^{th} file is a jpeg image and set to 0 otherwise (for all j).
- Query 2: What is the fraction of files with write permission? The weight of each file is constant $w_i = 1$ and set $f(i,j) = 1$ if the file is open for writes and 0 otherwise.
- Query 3: If each file is a television program, how much of the time is advertisements? The weight w_i is set to the length of the file in seconds and $f(i,j) = 1$ if the j^{th} second in the i^{th} file is an advertisement.
- Query 4: What is the potential benefit of running a specific compression algorithm on all files in the file system? The weight w_i is equal to the file length. For position j in file i define $f(i,j)$ as the compression ration of the compression "locality" of this location in the file (for example, if the compression is done by independently compressing aligned input chunks, then the locality of the j^{th} position is the aligned chunk that it belongs to, and $f(i,j)$ is the compression ratio of this block. See [11] for an in depth discussion of such formalization).

B. Putting it together

The formalization described above is put into context when using weighted sampling. Define the following method of approximation:

Sampling based estimation:

1) Run the weighted sampling algorithm to obtain a sample set S of M files in $\{1, \cdots, N\}$
2) For each $i \in S$ take choose j_i uniformly at random in $\{1, \cdots, w_i\}$
3) Output the estimation

$$\overline{F} = \frac{1}{M} \sum_{i \in S} f(i, j_i)$$

The estimation technique and method all come together using the following theorem:

Theorem 1. *Let F be a weighted average as in Definition 1 and $\overline{F}$ be the output of the sampling estimation method above using set S taken from a weighted multinomial distribution (see Section II-B). Suppose that there are constants a and b such that $a \leq f(i,j) \leq b$ then for every constant $\varepsilon > 0$ it holds that*

$$Prob[|F - \overline{F}| > \varepsilon] < 2e^{\frac{-2M\varepsilon^2}{(b-a)^2}}$$

The proof is a direct result of the Hoeffding Inequality [12], when considering the *positions* in the files as the basic random variables at hand. The abstraction of looking at $f(i,j)$ as a measure for positions inside a file (rather than considering a measured defined on files) is very helpful in simplifying the analysis (even though for many of the interesting use-cases f is only a function of its the file i and not the position j). The formal proof is omitted.

a) Using Theorem 1: Note that Theorem 1 requires a bound on the range of the function f and indeed if the variance of f is excessive then the accuracy of the estimation cannot be guaranteed. In applications is typically possible to consider measures that are bounded between in the range $[0,1]$ (as seen in all of the examples in Section A). Then the theorem guarantees are of the following type: When taking 5000 samples for a function with $f(i,j) \in [0,1]$ it is guaranteed that the estimation skew will exceed 0.028 with probability at most $\frac{1}{1000}$ and will skew by more than 0.038 only with probability less than 1 in a million. In general the probability that an estimation will skew by more than ε decays exponentially as the number of samples M grows. The fact that ε is squared means that in order to reduce the skew by a factor of k one needs to increase the number of samples by k^2 (e.g., halving the skew requires 4 times as many samples).

Number of Samples	Confidence Level	Hoeffding Skew (ε)	Poisson Skew (ε)
5000	10^{-3}	0.028	0.0048
5000	10^{-6}	0.038	0.0076
10000	10^{-3}	0.019	0.0033
10000	10^{-6}	0.027	0.0052

TABLE II
COMPARISON BETWEEN USING THE HOEFFDING BOUND (THEOREM 1) WITH $b - a = 1$ VS. A POISSON ESTIMATION FOR THE CASE THAT $f(i,j)$ IS A BERNOULI TRIAL WITH PROBABILITY $p = 0.01$. IT IS EVIDENT THAT WHEN IT IS POSSIBLE TO USE THE POISSON ESTIMATION (BERNOULI TRIALS WITH LOW PROBABILITY) THEN THE ACCURACY GUARANTEES ARE MUCH BETTER THAN IN THE GENERAL CASE.

A special interesting case is that all the values of f are 0 or 1, and that this occurs with low probability. For example, the testing use-case (Section I-B2) expects a low probability of errors ($f(i,j) = 1$) or else the method being tested is very bad. In this case using Theorem 1 does not provide tight enough accuracy guarantees (for example, guaranteeing a skew of ± 0.028 for an values inside 0.05 is just not useful. The reason for this shortcoming is that the underlying Hoeffding Inequality ignores the variance of the random variables at hand and uses their range to give a worst case bound on the variance. In this important case it is worthwhile to use an approximation of the weighted average using the Poisson distribution - this approximation is known as "the law of rare events" or "the law of small numbers" (see for example in [18], chapter 4.2).

In Table II we give some numerical example of the accuracy that can be guaranteed when using Theorem 1 and when using a Poisson estimation with probability of error $p = 0.01$. As seen the guarantees provided using this bound are much tighter and make the method appealing in these ranges as well.