Consistent Weighted Sampling

Mark Manasse
Microsoft Research, SVC
manasse@microsoft.com

Frank McSherry
Microsoft Research, SVC
mcsherry@microsoft.com

Kunal Talwar
Microsoft Research, SVC
kunal@microsoft.com

July 2, 2008

Abstract

We describe an efficient procedure for sampling representatives from a weighted set such that for any weightings S and T, the probability that the two choose the same sample is equal to the Jaccard similarity between them:

$$\mathbf{Pr}[sample(S) = sample(T)] = \frac{\sum_x \min(S(x), T(x))}{\sum_x \max(S(x), T(x))}$$

where $sample(S)$ is a pair (x, y) with $0 < y \leq S(x)$. The sampling process takes expected computation linear in the number of non-zero weights in S, independent of the weights themselves.

Sampling computations of this form are commonly limited mainly by the required (pseudo) randomness, which must be carefully maintained and reproduced to provide the consistency properties. Whereas previous approaches require randomness dependent on the sizes of the weights, we use an expected number of bits per weight independent of the values of the weights themselves.

Furthermore, we discuss and develop the implementation of our sampling schemes, reducing the requisite computation and randomness substantially in practice.

1 Introduction

"Consistent sampling" refers to sampling processes with the property that, while producing an appropriate distribution (perhaps uniform) over samples from an input set, the samples produced from differing input sets are *coupled*, so that samples from similar sets are likely to be identical. The standard approach for defining such a process, from [3], is to apply a random hash function to each element in a set and retain the pre-image of the greatest resulting hash.[1] The resulting sample from any set is uniformly distributed over elements of that set, and the probability that two samples drawn from different sets S and T are equal is their Jaccard similarity $|S \cap T|/|S \cup T|$. A typical application of consistent sampling is to efficiently estimate the similarity of web pages (when viewed as sets of words, or possibly k-word phrases): the probability of identical samples from two documents equals their Jaccard similarity as sets, and multiple independent trials can be used to accurately estimate this quantity.

In many contexts we do not want to sample uniformly from the set, but rather in proportion to a set of predetermined weights. In the example of web documents, we might consider applying some version of term frequency normalization (eg: TF-IDF), where we weight terms by their frequency in the document, and discount terms by their frequency in the corpus of all relevant documents. By selecting samples using TF-IDF weights, we are more likely to select informative terms, arriving at a more accurate estimate for semantic similarity when comparing the vectors of samples.

Traditional consistent sampling can be applied to non-uniform distributions by replicating each item a number of times proportional to its weight, and tagging each replica with its ordinal position, so item x with weight $S(x)$ is replaced by $S(x)$ items $(x, 1), (x, 2), \ldots, (x, S(x))$. This introduces substantial additional hash evaluations, linear in the weights, and constrains the weights to be integers (or, by scaling, integer multiples of some smallest non-zero weight). When sampling uniformly and consistently from the set of such pairs, the probability that two sets yield identical samples (both element and index) becomes the following natural generalization of the Jaccard similarity:

$$Pr[sample(S) = sample(T)] \quad = \quad \frac{\sum_x \min(S(x), T(x))}{\sum_x \max(S(x), T(x))} \, .$$

Recently, [6] showed that the computational burden can be substantially eased, giving a sampling algorithm that takes time proportional to the sum of the logarithms of the weights, assuming all weights are at least one. Their approach is motivated by the understanding that for any x, relatively few of the pairs (x, y) will be of interest: only those pairs whose hash value exceeds all preceding hash values, we will call them "active pairs", can ever possibly be returned as samples. Rather than iterate through all pairs (x, i), they generate the sequence of active pairs: from one active pair (x, i_j) with hash value h_j, the number of pairs until the hash value is improved follows a geometric distribution with mean $1/(1 - h_j)$. The value of i_{j+1} can be sampled from this distribution, without considering the intervening hash values, and the value of h_{j+1} is uniformly drawn from $(h_j, 1]$. As the i_j values grow exponentially in j, we arrive at the last element for which $i_j \leq S(x)$ when $j = O(\log(S(x)))$.

While this represents a substantial improvement over simple replication, the techniques of [6] are still constrained by the required normalization (to a minimum of 1) and a logarithmic dependency on the normalized weights. In essence, it can require per-candidate time proportional to the logarithm of the largest ratio of weights. As weightings can go both up and down, sometimes substantially so, these average values need not be negligible, as we describe later.

[1] It is perhaps more common to retain the pre-image of the least hash. We use the greatest hash because the mathematics simplify somewhat, but the two are equivalent.

We present a scheme for consistent weighted sampling taking expected constant time per non-zero weight. The weights themselves can be arbitrary non-negative real values. As in [6], our approach emulates the behavior of the replication approach, including the generalization to the continuous case. Unlike [6], our process allows "random access" to the sequence of indices i_j whose corresponding hash values exceed those at all lesser indices. For an arbitrary weight $S(x)$, we can leap directly to the relevant region of the sequence and produce the appropriate representative index. Also, in contrast to [6], our scaling doesn't depend on a smallest weight, but instead is capable of producing a doubly-infinite sequence of indices i_j, heading to zero or positive infinity as j heads to negative or positive infinity, respectively.

1.1 Applications

Consistent sampling raises many interesting theoretical and algorithmic questions, but also bears practical relevance to many problems. We take a moment to survey several different settings, ranging from deeply applied web processing to more theoretical contexts.

1.1.1 Similarity / Duplicate Detection

Many large corpora, noted first in web documents [3] and file-systems [12], are filled with content that may be replicated multiple times, in whole or in part. As direct hashing of content will capture only exact duplication, a common approach is to consistently sample from the set of n-grams in the document. Locality-sensitive hashing [8, 4] generalizes the sampling techniques of the earlier papers, presenting a general theory for estimating proximity.

Weighting features may be important if given a deeper understanding of the objects in the set. For example, in web pages, a match in the title may be viewed as more significant than other matches. Words or phrases might be sampled with probability that decreases with frequency in the corpus, as commonplace terms or phrases may represent idiomatic or spurious duplication. Confidence scores may exist for parts of the document that were unsuccessfully extracted, and may also suggest lower probability of selection. Each of these reasons can skew the distribution from uniform, resulting in a modified Jaccard similarity score that more accurately corresponds to duplication.

1.1.2 Mechanism Design

Mechanism design refers to the design of algorithms whose intended input is supplied by strategic players: consider an auction, whose inputs are bids from interested parties. The players are assumed intelligent, and willing to manipulate their inputs if they believe that doing so will lead to more appealing outcomes.

In recent work, [5] shows that mechanisms satisfying *differential privacy*, the property that no outputs become substantially more or less likely as a result of any one player's input, can be transformed to mechanisms with the additional guarantee that with high probability, the output of the mechanism simply does not change as a result of changes to any one player's input.

Their transformation results from the application of consistent sampling to the distribution over outputs. Since the change in distribution caused by a single participant is relatively small, the similarity of the distributions is nearly one, and consequently a consistent sampling scheme will, with high probability, return identical answers. The implementation they propose is, unfortunately, limited by the fact that in these constructions the probability densities vary exponentially, making the direct replication approach infeasible, and the more recent logarithmic ascent approach of [6] super-linear.

1.1.3 Embeddings

As noted by Charikar [4], consistent sampling is closely related to the problem of rounding of the earthmover linear program for the metric labeling problem [10]. In fact, one way to look at the Kleinberg-Tardos

rounding for the metric labeling problem on uniform metrics is to note that the Jaccard distance, taken to be one minus the Jaccard similarity, is a 2-approximation to the ℓ_1 distance between norm 1 vectors, and to observe that min-wise hashing gives an unbiased estimator for the Jaccard distance; this is exactly the Kleinberg-Tardos algorithm. The rounding on earthmover linear programs is also used for approximating other problems such as 0-extension [1] and the multiway uncut problem [11]. Moreover, Charikar [4] uses consistent sampling to design fast sketching/streaming algorithms for similarity estimation. Our work now gives an efficient implementation of the rounding algorithm.

Searching for nearest neighbors in the earthmover metric is an important problem and has been used for image retrieval(see e.g. [9] and the references therein). Since consistent sampling gives an approximate estimator for the earthmover distance between distributions, this leads to a fast approximate locality sensitive hash function. This naturally leads to an efficient near neighbor search algorithm for such spaces.

1.2 Previous Work

The earliest applications of consistent sampling techniques were in examining file-systems, looking for repeated segments of data, as described in [12, 7]. Independently, and roughly contemporaneously, Broder *et al.* [3, 2] applied similarity techniques to a collection of web pages. All of these develop techniques for selecting samples from files or documents in such a way that small edits are likely to result in small changes to the set of samples. All of these papers cast sample selection as selection of hash values of features of the underlying objects, but viewing this as selection of the pre-image of the chosen hash value leads to samples which are elements of the objects in question.

In support of related goals, and coming from a background of dimension reduction, Indyk and Motwani [8] investigated techniques looking not at sampling, but rather at vectors in high-dimensional spaces with small angles between them. This line of research views the sampling techniques as a restricted class of locality-sensitive hashing, a broader class in which the goal is to produce a family of hash functions such that, averaged over the family, the similarity of two objects can be estimated by looking at the average number of matching hash values. While this is also true of the sampling techniques, the hashes considered in locality sensitive hashing often consider the vector as a whole, looking at hyperplanes to produce hash bits by determining whether a vector and the normal vector to a hyperplane have positive inner product.

Note that the latter technique is very compatible with weighted inputs; the magnitude of the vector in a given dimension can be the weight of that dimension, and smaller values will be less influential in the inner product computation. The sample-based techniques also accomodated weighting, but at higher cost (linear in the sum of the weights), and supporting only integer weights. This last restriction can be, to some extent, bypassed by scaling, but this adds a multiplicative factor to the computational effort; in this scaling, we replace feature f with weight w by w features $f_1, f_2, ..., f_w$.

Recently, [6] improved on this by presenting an iterative algorithm for producing a sample, running in time proportional to $\sum 1 + \log(w_f)$. It still requires the weights to be larger than some predetermined minimum, most conveniently taken to be one. Scaling the weights in a set make it possible to choose weights larger than one, but the samples chosen depend on the scaling. Thus, if the universe of weighted sets is unknown, new sets may appear at a later time with weights smaller than the previously chosen scaling factor. In this case, the samples chosen for the new weighting will not be comparable with previously chosen samples. Our new algorithm, in contrast, avoids the logarithmic cost of the weights, and allows consistent selection of samples even in the presence of arbitrarily small or large weights.

2 Main Result

The sampling algorithm we develop will take in a non-empty weighted set S, where a weight $S(x)$ is associated with every x, and produce a representative sample of the form (x, y) where x is the selected object, and y is a positive weight value that is at most $S(x)$. The distribution over (x, y) will be uniform over the pairs satisfying $0 < y \leq S(x)$. That is, x is chosen with probability proportional to $S(x)$, and y uniformly between 0 and $S(x)$. Note that elements in the universe but not in the set (*i.e.* elements with weight 0) will never be chosen as samples. The sampling algorithm is also consistent, producing the same sample from different sets with probability equal to the Jaccard similarity. Finally, the sampling algorithm does so in expected time linear in the number of positive weights, whose only dependence on the values of the weights comes from the assumption that operations such as `log` take unit time.

Before presenting the specifics of the algorithm, we establish two necessary and sufficient conditions for a sampling process to yield the Jaccard similarity as the probability of sample collision:

1. **Uniformity**: The sample (x, y) is distributed uniformly over the pairs satisfying $y \leq S(x)$.

2. **Consistency**: If $T(w) \leq S(w)$ for all w, then whenever (x, y) satisfying $y < T(x)$ is sampled from S, (x, y) is also sampled from T.

Lemma 1 *For any uniform and consistent sampling scheme,*

$$Pr[sample(S) = sample(T)] \quad = \quad \frac{\sum_x \min(S(x), T(x))}{\sum_x \max(S(x), T(x))} \ .$$

Proof: Consider weights $R(x) = \max(S(x), T(x))$. By consistency, samples from S and T coincide when the sample (x, y) from R satisfies $y \leq \min(S(x), T(x))$. By uniformity, this happens with the stated probability. ∎

2.1 Algorithm Definition

Our sampling algorithm is composed of two steps: for each element x with non-zero weight, we choose a representative (x, y) satisfying $y \leq S(x)$. Secondly, for each representative (x, y) we produce a hash value $h(x, y)$, and report the pair with the greatest hash value.

2.1.1 Index Production

As done in [6], we will produce the sequence of "active" indices, those whose hash values exceed the hash values of any lesser index. The chosen y value for x will then be the greatest active index below $S(x)$. Unlike [6], our sequence will extend infinitely in both directions, towards 0 as well as ∞, and will allow effective random access to any region of the sequence.

The main property we leverage is that the distribution over active indices in any interval is independent from the distribution over active indices in a disjoint interval. We will decompose $(0, \infty)$ into intervals of the form $(2^{k-1}, 2^k]$, and determine which indices are active in each interval independently, using pseudo-randomness whose seed depends in part on k. Using such a scheme, we can leap directly to any such interval and explore the active indices therein. In the following algorithm, $salt$ is a string used as part of the seed of the random number generator.

The following code fragment outputs the active indices in the interval $(2^{k-1}, 2^k]$:

GenerateSamples(x, k, salt)

1. random.seed(x, k, salt);
2. sample = 2^k * random.uniform(0.0, 1.0);
3. while (sample $> 2^{k-1}$)
   (a) record(sample);
   (b) sample = sample * random.uniform(0.0, 1.0);

The collection of these descending sequences will conceptually form the full sequence, though we have effective random access to any point in the sequence. We can use **GenerateSamples** to determine the values immediately above or below any weight $S(x)$ by computing $\lceil \log(S(x)) \rceil$ and exploring the subsequences in intervals up or down until we find a suitable element. Importantly, when determining the active indices in another interval we do *not* fall through from the computation of a prior interval, but always restart the computation using a new invocation of **GenerateSamples** with the appropriate new value of k. This ensures that all computations have a consistent view of the active indices in each interval, independent of their history and prior consumption of randomness.

As each interval is non-empty with probability exactly $1/2$ and of constant expected length, we can determine the greatest value $y \leq S(x)$ and the least value $z > S(x)$ in expected constant time. We will refer to this process (not formally stated) as **ActiveIndices**$(x, S(x), salt)$.

2.1.2 Hash Production

The hash value we produce should be independent of y, but nonetheless consistent. An excellent source of randomness will be the least active value $z > S(x)$, which is consistent (any weight $S(x)$ immediately above y is necessarily immediately below z) and independent from y, once conditioned on $S(x)$. From z we define and sample from the following carefully chosen cumulative density function over hash values $a \in [0, 1]$:

$$cdf_z(a) \quad = \quad a^z + a^z z \ln(1/a) \ . \tag{1}$$

To produce a hash value from cdf_z, we choose a value β_x uniformly at random from $[0, 1]$, and set $h(x, y) = cdf_z^{-1}(\beta_x)$, which we can find using binary search as cdf_z is monotone. Importantly, the value β_x is chosen from a pseudo-random source whose seed depends on x, but not $S(x)$, y, or z.

Relying on the density derivation proved below in Section **??**, the distribution (1) is used because it has the property that for any $S(x)$, when we integrate cdf_z over the possible values $z > S(x)$ using the density $S(x)/z^2$, derived below, we arrive at the following cumulative density function over hash values $a \in [0, 1]$:

$$S(x) \int_{S(x)}^{\infty} \left(a^z/z^2 + a^z \ln(1/a)/z \right) dz \ = \ a^{S(x)} \ . \tag{2}$$

This is appealing, as $a^{S(x)}$ is the cumulative density function of the maximum of $S(x)$ independent random values between 0 and 1. Drawing hash values from this distribution for each x and then selecting the x with the largest hash properly emulates the replication process.

We write $F_{S(x)}(a) = a^{S(x)}$ for the cumulative density function over hash values described in (2), and write $f_{S(x)}(a) = S(x) a^{S(x)-1}$ for the probability density function associated with $F_{S(x)}$.

2.1.3 Algorithm

Based on the previously described structure (and again using $salt$ to distinguish different members of the family of sampling functions and different uses of randomness), the pseudo-code for our sampling algorithm looks as follows:

ConsistentSample(weights S, salt)

1. hmax = 0; xmax = null; ymax = 0;
2. foreach (x : nonzero S(x))
   (a) random.seed(x, salt);
   (b) β = random.uniform(0.0, 1.0);
   (c) (y, z) = **ActiveIndices**(x, S(x), salt);
   (d) Compute $h = cdf_z^{-1}(\beta)$ via binary search;
   (e) If ($h >$ hmax)
   $\{$hmax = h; xmax = x; ymax = y;$\}$
3. Return the sample (xmax, ymax)

As mentioned previously, **ActiveIndices** uses **GenerateSamples** to produce the greatest active index $y \leq S(x)$ and least active index $z > S(x)$. **GenerateSamples** is used to investigate the active indices in the power-of-two interval containing $S(x)$, as well as adjacent intervals as needed.

2.2 Proof of Correctness

As noted previously, the two important features we need to establish are uniformity and consistency.

Uniformity is established in two parts: by proving that y is uniform in $[0, S(x)]$ and that x is selected with probability proportional to $S(x)$. Uniformity of y follows from our construction of the sequence of active values; each active index is uniformly distributed between zero and the next active index. For any weight $S(x)$, the sampled index y is uniformly distributed over $[0, z]$, and, conditioned on being at most $S(x)$, uniformly distributed over $[0, S(x)]$.

The specific non-uniformity of x follows from the independence of the hash values and nature of their distributions. Formally, the probability that x is selected is the integral over possible values a from 0 to 1 of the density $f_{S(x)}(a)$ that a is chosen as the hash, times the probability that every other term x' chooses a lesser hash, $\prod_{x' \neq x} F_{S(x')}(a)$:

$$\mathbf{Pr}[hash(x) > \max_{x' \neq x}(hash(x'))] \quad = \quad \int_0^1 \left(f_{S(x)}(a) \prod_{x' \neq x} F_{S(x')}(a) \right) da \; . \tag{3}$$

Substituting the definition of $F_{S(x')}(a)$ and $f_{S(x)}(a)$, multiplying out, and intergrating, we get

$$\int_0^1 \left(S(x) a^{S(x)-1} \prod_{x' \neq x} a^{S(x')} \right) da \quad = \quad S(x) \int_0^1 \left(a^{\sum_{x'} S(x') - 1} \right) da \quad = \quad S(x) / \sum_{x'} S(x') \; . \tag{4}$$

This final probability is exactly what we seek: x selected with probability proportional to $S(x)$.

Consistency is established as follows: For any S and T with $S(w) \geq T(w)$ for all w, if (x, y) is selected by S and $T(x) > y$, then (x, y) will be considered by T, with the same hash value, against other representative elements whose hash values are at most those of the other elements considered by S, all of which are less than $h(x, y)$. This last point is subtle, and only holds because of the consistent use of β_x for each x, and the fact that $cdf_z^{-1}(\beta_x)$ strictly decreases with increasing z.

2.3 Integral Counts and Representatives

In the case of integral counts, the mathematics and their derivations are perhaps a bit more clear. We take a moment to describe the derivations in the integral case, and show that the limit of the integral case achieves the continuous mathematics.

First, the density at any integral value z is exactly $S(x)/(z(z-1))$. This follows from direct computation: an index z' is active with probability $1/z'$, and for an index z to be the least active index above $S(x)$, it must be active, and each element in between must be inactive. These two events lead to the probability

$$1/z \times \prod_{S(x)<z'<z} (1 - 1/z') \quad = \quad S(x)/(z(z-1)) \tag{5}$$

Consequently, we would like to use a cumulative density function that when integrated using these probabilities, over z from $S(x)$ to infinity, lead to the cumulative density function $a^{S(x)}$. The appropriate cdf_z function turns out to be:

$$cdf_z(a) \quad = \quad za^{z-1} - (z-1)a^z \ . \tag{6}$$

Remark: We should point out that the continuous case can easily emulate the integral case, simply taking any output samples of the form (x, y) and rounding y upwards to the nearest integer. Assuming $S(x)$ is integral, the rounded y values are uniformly distributed and the representatives x chosen with the correct probability owing to the correctness of the continuous approach. Likewise, we could use the integral mathematics to derive the continuous form, by discritizing the real line, and taking the limit as the the interval widths go to zero.

3 Implementation Enhancements

Algorithms for sampling are aimed at accelerating interaction with sets, and it is natural to try to squeeze as much performance out of a sampling algorithm as is possible. A direct implementation of the algorithm as described consumes a fixed, but substantial, amount of computational resources. Notably, the production of random bits is expensive, and the inversion of cdf_z seems to require a numerical, rather than analytic, approach.

Consequently, we now describe several implementation details that can be used to accelerate the production of samples, to use randomness more carefully, to avoid unnecessary computation, and to amortize certain resources across parallel sampling instances. We measure the impact of these decisions on synthetic sets of varying length with uniform weights. In all cases, non-uniform weights improve performance when the representation is first sorted in decreasing order of weights.

3.1 Deferred Evaluation of y

Determining which sample x is to be returned does not actually require the determination of which value y will accompany it, but rather only requires the value z. Consequently, we can defer the determination of y until we have determined x and are ready to return.

3.2 Avoiding Numerical Inversion

We define the hash value $h(x, y)$ as the pre-image of a uniformly random value β under the somewhat messy density function cdf_z. We are not aware of a way to easily invert the density function, and instead exploit the monotonicity of the cumulative density function to perform binary search over the hash values. Searching to a sufficient degree of accuracy is expensive, and as we do it for each term it quickly becomes the computational bottleneck.

However, we do not actually need to compute a hash value for every x, we only need to determine which x has the best value. Rather than compute and compare $cdf_z^{-1}(\beta)$ to the best hash h seen so far, we can exploit the monotonicity of cdf_z, which implies

$$cdf_z^{-1}(\beta) > h \quad \text{iff} \quad \beta > cdf_z(h) . \tag{7}$$

Using this test, we can efficiently determine *if* we need to compute $cdf_z^{-1}(\beta)$ before actually doing so. This reduces the number of numerical inversions to the number of times the best sample changes in the course of processing the document.

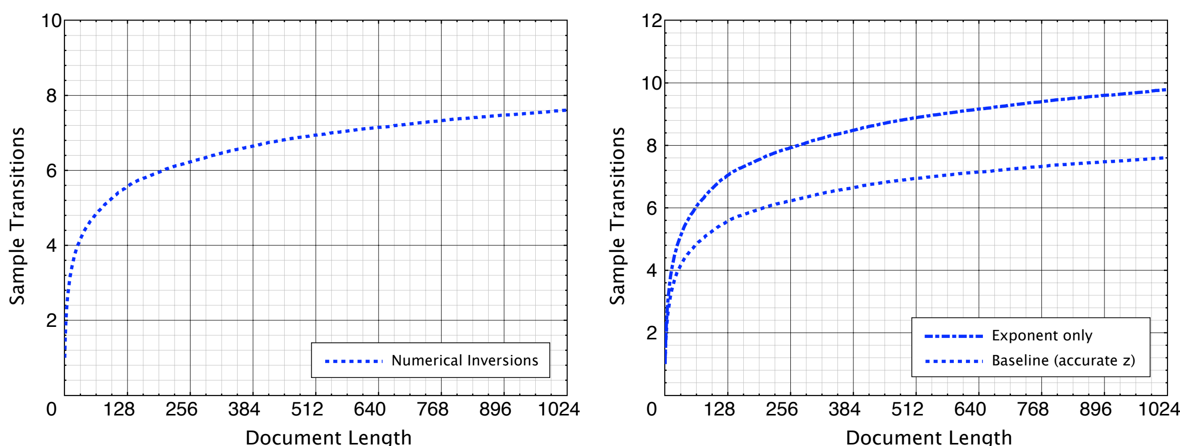

Figure 1: Number of Numerical Inversions and Approximate z Schemes

We see in Figure 1 the number of numerical inversions we conduct as a function of set size. The experiments confirm mathematical intuition that the number of inversions should be logarithmic in the length: with uniform weights, the probability that term i needs to be inverted is $1/i$, and the sum of these terms forms the harmonic sequence, approximately $\ln(n)$ for length n documents. For sets with non-uniform weights, we will see fewer inversions if the weights are processed in decreasing order, since heavier set elements are more likely to be selected and to have large synthesized hash values.

3.3 Use of Randomness

As an alternate approach to explicitly investigating each interval $(2^{k-1}, 2^k]$ to see if it is non-empty, which happens with probability exactly $1/2$, we could produce a bit vector whose kth bit indicates non-emptiness of the sequence in that interval. By examining this bit vector, we could determine which values of k merit a call to **GenerateSamples**, saving ourselves the trouble of exploring empty intervals. The **GenerateSamples** method will of course need modification to ensure that whenever it is invoked it produces at least one sample, as promised, which we can do by setting the first bit of the first uniform random number.

Even more cleverly, we can steal this first bit, capturing it rather than setting it, and using it for the first bit in the next sample, warning us that the coming value will fall below 2^{k-1} before we even produce the value. Doing likewise with each subsequent sample, we avoid producing many of the random variables that we do not need.

3.4 Partial Evaluation of z

In subsection 3.3 we described the use of a bit vector, from which we can determine the highest interval $(2^{k-1}, 2^k]$ in which z could lie, followed by a call to **GenerateSamples** to determine where in the interval it would land. We can view these two steps as producing the exponent and mantissa of z separately. In light of subsection 3.2, and by virtue of the monotonicity of cdf_z with respect to z, we can avoid computation of the mantissa if we can determine that even the largest possible value of z would not compare favorably to our current best hash h. We simply compare β against $cdf_{2^k}(h)$, and only evaluate the mantissa if β is the larger.

Figure 1 plots the number of times that we need to fully evaluate the mantissa of z given only β and the exponent field of z. Knowing z up to a power of two is sufficient to rule out most of the candidates without computing the mantissa, thereby saving a substantial amount of computation.

Remark: It is important that the upper bound we use for z is determined by the first active bit in the exponent bitvector that is *strictly* beyond the one associated with $S(x)$. Even though z may live in this interval when we investigate it, we can not rule out the possibility that all the elements in the interval are less than $S(x)$.

3.5 Parallel Sampling

When we are trying to produce many samples in parallel, we require many β values and many z values. We do not necessarily need to evaluate each β and z fully, and rather than generating a large number of fully formed β and z values, we can carefully parcel out the randomness in small amounts to each of the parallel instances, providing each β and z with enough randomness to quickly rule out those x that will not lead to viable samples. We can produce additional randomness for those β and z values that may lead to viable samples, which we hope happens infrequently.

Having discussed an approximation for z in Section 3.4, we now discuss two examples for approximations to β. For the first scheme, we chose to take 8 bits of randomness for each sample, with 128 bits leading to 16 parallel samples. The second, adaptive scheme takes the same 128 bits and reads out β values by proceeding through the bits, and emitting a number when it first sees a zero. That is, it takes the sequence 10110100 and outputs 10, 110, 10, and 0. This has the advantage of terminating early on samples that are unlikely to be close to one, and continuing deeply into those that are.

We now compare these two schemes, measuring the number of times that we need to produce additional randomness as a function of the document length. We use synthetic sets which have weight one for each "term" in the document. Non-uniformity in the weights, if the document is sorted by decreasing weights, would only help, as subsequent terms are even less likely to be viable.

In Figure 2 we measure (from bottom to top) the number of times the maximum hash value changes, as a lower limit on the times we must fully evaluate β and z, the number of times that a fully formed β would be viable, given that we are not fully computing z, as a lower limit on the possible culling that approximate β values might produce, followed by the adaptive and oblivious schemes.

Notice that initially, the oblivious scheme outperforms the adaptive scheme, due to its larger number of bits. However, as the document sizes increase, the adaptive scheme begins to gain ground, due to its ability to investigate beyond the first 8 if needed. The oblivious scheme can only cull at most $255/256$ of the elements, as a value of 11111111 will always be viable. Looking farther out, we see quite clearly that

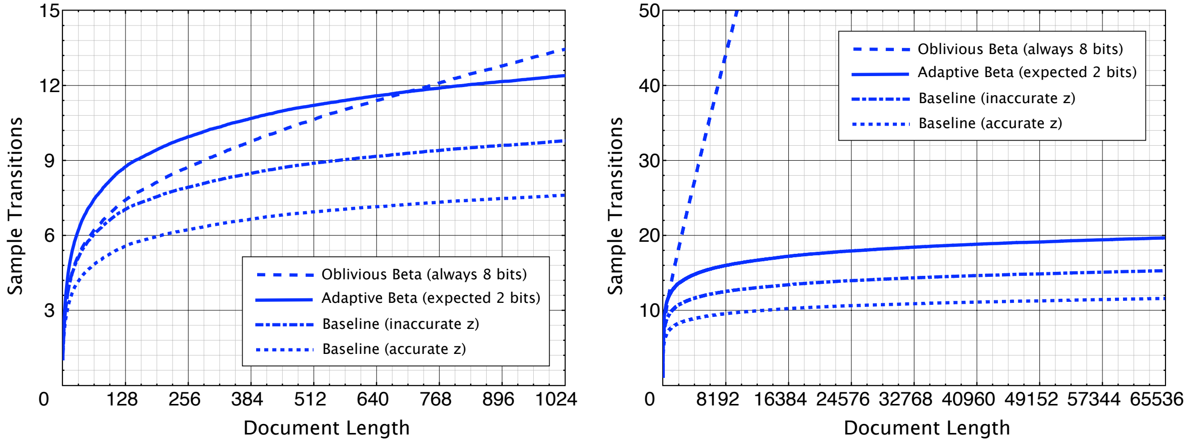

Figure 2: Approximate β Schemes

the oblivious scheme loses out to the adaptive scheme.

3.6 Numerical Accuracy

The skeptical reader may have already noted that the cdf_z function we use tends to produce terribly small values, raising its input a to very large powers. For large z, we must take h very nearly one to prevent $cdf_z(h)$ from vanishing, roughly $1 - 1/z$, but floating point numbers do not excel at maintaining high precision away from zero. While not a problem for most z, we do expect to see rather large values infrequently, as z has infinite expectation.

Fortunately, these problems can be ameliorated with careful mathematical manipulation. Rather than solving for $h = cdf_z^{-1}(\beta)$, with its numerical imprecision, we can solve for h^z using its density function:

$$cdf(a^z) \quad = \quad a^z(1 - \ln(a^z)) \; . \tag{8}$$

Having computed $h^z = cdf^{-1}(\beta)$, we could either raise it to the power $1/z$, which returns us to the realm of numerical inaccuracy, or simply compute $\log(h^z)/z$, which accurately represents $\log(h)$, and which as a monotone function of h is sufficient for comparisons between two hash values.

Going to an extreme, in applications like mechanism design [5, 13], where the weights are exponentially sized and only stored as their logarithms, we would like to avoid even representing z, producing and computing with $\log(z)$. This is not hard to do as z is produced in an exponent and mantissa form, corresponding to the integral and fractional components of its logarithm. Rather than compare $\ln(h^z)/z$ values, we take another logarithm, computing and comparing values of

$$\log(\log(1/h^z)) - \log(z) \quad = \quad \log(\log(1/h)) \; , \tag{9}$$

which, by monotonicity, lets us compare h values indirectly.

4 Concluding Remarks

We described an algorithm for consistent sampling that takes a constant amount of computation and randomness for each non-zero weight that it sees. This improves on previous work, whose running time depended on the values of the weights, and for which a lower bound is assumed.

Additionally, we have developed and enhanced the implementation of the algorithm, reducing the number of numerical inversions to logarithmic in document length, and the amount of randomness per weight in expectation, in the limit. These optimizations are supported by empirical measurements of the efficacy.

References

[1] A. Archer, J. Fakcharoenphol, C. Harrelson, R. Krauthgamer, K. Talwar, and É. Tardos. Approximate classification via earthmover metrics. In J. I. Munro, editor, *SODA*, pages 1079–1087. SIAM, 2004.

[2] A. Z. Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[3] A. Broder, S. Glassman, M. Manasse Syntactic clustering of the web. In *Sixth International World Wide Web Conference*, 1997.

[4] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, New York, NY, USA, 2002. ACM Press.

[5] K. Chaudhuri, S. Kale, F. McSherry, and K. Talwar. From differential privacy to privacy with high probability, 2007. Manuscript.

[6] S. Gollapudi and R. Panigrahy. Exploiting asymmetry in hierarchical topic extraction. In *Proceedings of ACM Fifteenth Comnference on Knowledge Management*, 2006.

[7] N. Heintze. Scalable document fingerprinting. In *1996 USENIX Workshop on Electronic Commerce*, November 1996.

[8] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC*, pages 604–613, 1998.

[9] P. Indyk and N. Thaper. Fast color image retrieval via embeddings. In *Workshop on Statistical and Computational Theories of Vision*, 2003.

[10] J. Kleinberg and É. Tardos. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and markov random fields. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 14, Washington, DC, USA, 1999. IEEE Computer Society.

[11] M. Langberg, Y. Rabani, and C. Swamy. Approximation algorithms for graph homomorphism problems. In *Proceedings of Approximation Algorithms for Combinatorial Optimization (APPROX)*, pages 176–187, 2006.

[12] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Francisco, CA, USA, 17–21 1994.

[13] F. McSherry, and K. Talwar Mechanism design via differential privacy. To appear in FOCS 2007.