Don't Let The Negatives Bring You Down: Sampling from Streams of Signed Updates

Edith Cohen
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932, USA
edith@cohenwang.com

Graham Cormode
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932, USA
graham@research.att.com

Nick Duffield
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932, USA
duffield@research.att.com

ABSTRACT

Random sampling has been proven time and time again to be a powerful tool for working with large data. Queries over the full dataset are replaced by approximate queries over the smaller (and hence easier to store and manipulate) sample. The sample constitutes a flexible summary that supports a wide class of queries. But in many applications, datasets are modified with time, and it is desirable to update samples without requiring access to the full underlying datasets. In this paper, we introduce and analyze novel techniques for sampling over dynamic data, modeled as a stream of modifications to weights associated with each key.

While sampling schemes designed for stream applications can often readily accommodate positive updates to the dataset, much less is known for the case of negative updates, where weights are reduced or items deleted altogether. We primarily consider the turnstile model of streams, and extend classic schemes to incorporate negative updates. Perhaps surprisingly, the modifications to handle negative updates turn out to be natural and seamless extensions of the well-known positive update-only algorithms. We show that they produce unbiased estimators, and we relate their performance to the behavior of corresponding algorithms on insert-only streams with different parameters. A careful analysis is necessitated, in order to account for the fact that sampling choices for one key now depend on the choices made for other keys.

In practice, our solutions turn out to be efficient and accurate. Compared to recent algorithms for L_p sampling which can be applied to this problem, they are significantly more reliable, and dramatically faster.

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Probabilistic Algorithms

Keywords

sampling, deletions, data streams, updates

1. INTRODUCTION

Random sampling has repeatedly been proven to be a powerful and effective technique for dealing with massive data. The case for working with samples is compelling: in many cases, to estimate a function over the full data, it suffices to evaluate the same function over the much smaller sample. Consequently, sampling techniques have found applications in areas such as network monitoring, database management, and web data analysis. Complementing this, there is a rich theory of sampling, based on demonstrating that different techniques produce unbiased estimators with low or optimal variance.

Initial work on sampling from large datasets focused on a random access model: the data is static, and disk resident, and we want to build an effective sample with a minimal number of probes to the data. However, in modern applications we do not think of the data as static, but rather as constantly changing. In this setting, we can conceive of the data as being defined by a stream of transactions, and we get to see each transaction that modifies the current state. For concreteness we describe some motivating scenarios:

- First, we consider the case when the stream is a sequence of financial transactions, each of which updates an account balance. It is important to be able to maintain a sample over current balances, which describes the overall state of the system, and to provide a snapshot of the system against which to quickly test for anomalies without having to traverse the entire account database.
- Internet Service Providers (ISPs) witness streams in the form of network activities. These can set up or tear down (stateful) network connections. It is not practical for the ISP to centrally keep a complete list of all current and active connections. It is nevertheless helpful to draw a sample of these connections, so that the ISP can keep statistics on quality of service, round-trip delay, nature of traffic in its network and so on. Such statistics are needed to show that its agreements with customers are being met, and for traffic shaping and planning purposes.
- Updates to a database generate a stream of transactions, to insert or delete records to tables. The database management system needs to keep statistics on each attribute within a table, to determine what indices to keep, and how to optimize query processing. Currently, deployed systems track only simple aggregates online (e.g. number of records in a table), and

must periodically rebuild more complex statistics with a complete scan, which makes this approach unsuitable for (near) real-time systems.

In all these examples, and others like them, we can extract a central problem. We are seeing a stream of weighted updates to a set of keys, and we wish to maintain a sample that reflects the current weights of the keyset. Specifically, we aim to support queries over the total weight associated with a subset of keys (subset-sum queries), which are the basis of most complex queries [6]. If the set of keys ever seen is tiny, then we could just retain the weights of these keys, but in general there are many active keys at any one time, and so we want to maintain just a small sample. From this sample, we should be able to accurately address the problems in the above examples, such as fraud detection, finding anomalies, and reporting behavior trends.

The core problem of sampling from a stream of distinct, unweighted keys is one of the earliest examples of what we now think of as streaming algorithms [22]. In this paper, we tackle a much more general version of the problem: each key may appear multiple times in the stream. Each occurrence is associated with a weight, and the total weight of a key is the sum of its associated weights. Moreover, the updates to the weights are allowed to be *negative*. This models various queuing and arrival/departure processes which can arise in the above applications. Despite the naturalness and generality of this problem, there has been minimal study of sampling when the updates may be negative. Through our analysis, we are able to show that there are quite intuitive algorithms to maintain a sample under such updates, that offer unbiased estimators, and whose performance can be bounded in terms of the performance of a comparable algorithm on an update stream with positive updates only. We next make precise the general model of streams we adopt in this paper, and go on to describe prior work in this area and why it does not provide a satisfactory solution. At the end of this section, we summarize our contributions in this work, and outline our approach.

Streaming Model. We primarily consider the *turnstile stream* model, where each entry is a (positive or negative) update to the value of a key i. Here, the data is a stream of updates of the form (i, Δ), where i is a key and $\Delta \in \mathbb{R}$. The value v_i of key i is initially 0 and is modified with every update, but we do not allow the aggregate value to become negative. Formally, the value of key i is initially $v_i = 0$ and after update (i, Δ),

$$v_i \leftarrow \max\{0, v_i + \Delta\} \ . \tag{1}$$

1.1 Prior Work on Stream Sampling

The simplest form of sampling is Bernoulli sampling, where each occurrence of an (unweighted) key is included in the sample with probability q. More generally, Poisson sampling includes key i in the sample with probability q_i (depending on i but independent of other keys). The expected sample size is $\sum_i q_i$. However, typically we wish to fix a desired sample size k, and draw a sample of exactly this size.

To draw a uniform sample of size k from a stream of (distinct) keys, the so-called reservoir sampling algorithm chooses to sample the ith key with probability $1/i$, and overwrite one of the existing k keys uniformly [18]. This algorithm, attributed to Waterman, was refined and extended by Vitter [22].

Gibbons and Matias proposed two generalizations in the form of counting samples and concise samples for when the stream can contain multiple occurrences of the same (unit weight) key [14]. Concise samples are Bernoulli samples where multiple occurrences of the same key are replaced in the sample with one occurrence and a count, obtaining a larger effective sample. This approach is used in networking as the basis of the "sampled netflow" format (`http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/12s_sanf.html`). In the case of counting samples, however, the sampling scheme is altered to count all subsequent occurrences of a sampled key. The same idea was referred to as Sample and Hold (SH) by Estan and Varghese, who applied it to network data with integral weights, and provided unbiased estimators [10].

In detail, the SH algorithm maintains a *cache* S of keys and a counter c_i for each cached key $i \in S$. The (true) value v_i of a key i is initially zero and each stream occurrence increments v_i. When an increment of key i is processed, then if $i \in S$ (the key is already cached), its counter is incremented $c_i \leftarrow c_i + 1$. Otherwise, i is inserted into the cache with (fixed) probability q and c_i is initialized to 0. Clearly, the probability that a key is not cached is $(1-q)^{v_i}$ and when cached the distribution of $v_i - c_i$ is geometric with parameter q. SH is particularly effective when multiple occurrences are likely. Because once a key is cached, all subsequent occurrences are counted, the resulting sample is more informative than "packet" sampling, where only a q fraction of the updates are sampled and the count of each key is the number of sampled updates for this key. It therefore offers lower variance than a concise sample for the same cache size k.

A drawback of SH is the fixed sampling rate q, which means that it is not possible to exactly control cache size. An adaptive version of sample and hold (aSH) was proposed in [14, 10] where subsampling is used to adaptively decrease q so that the number of cached keys does not exceed some fixed limit k. Subsampling is applied to the current cache content and the result mimics an application of SH with respect to the new lower $q' < q$. A very similar idea with geometric step sizes was proposed by Manku and Motwani in the form of "sticky sampling" [19].

Estan and Varghese [10] showed that with SH, $\hat{v}_i = 1/q + c_i - 1$ if $i \in S$ and 0 otherwise is an unbiased estimate of v_i. Thus, an unbiased estimate on the total value of selected keys (based on a selection predicate P) is the sum of $\hat{v}_i$ over cached (sampled) keys that satisfy P. With aSH, however, the analysis is complicated by dependence of the rate adjustments in the randomization. Only in subsequent work were unbiased estimators for aSH presented, as well as estimators over SH and aSH counts for other queries including "flow size" distribution and secondary weights [6, 5]. The technique used to get around the dependence was to consider each key after "fixing" the randomization used for other keys. This could then be used to establish that the dependence actually works in the method's favor, as correlations are zero or non-positive.

We review (increment-only) aSH as presented in [6, 5]: new keys are cached until the cache contains $k+1$ keys. The sampling rate is then "smoothly" decreased, such that the cache content simulates the lower rate, until one key is ejected. Subsequent processing of the stream is subjected to the new rate. When a new key is cached, the sampling rate is again decreased until one key is ejected. The sampling rate of this process is monotone non-increasing. The set of keys cached by aSH is a PPSWR sample (sampling probability proportional to size without replacement), also known as bottom-k (order) sampling with exponentially distributed ranks [21, 7, 8]. Here, keys are successively sampled with probability proportional to their value and without replacement until k keys are selected. Since (when there are multiple updates per key) the exact values v_i of sampled keys are not readily available, we need to apply different (weaker) estimators than PPSWR estimators.

All the stream sampling techniques described so far, however, rely on the fact that all weights are positive: they do not allow the negative weights that arise in the motivating examples. In fact, there has been only very limited prior work on sampling in the turnstile model, also known as sampling with deletions. Gemulla *et al.* gave a generalization of reservoir sampling for the unweighted, distinct key case [12]. They subsequently studied maintaining a SH-style sample under *signed* unit updates [13], and proposed an algorithm that resembles SH and adds support for unit decrements by maintaining an additional "tracking counter" for each cached key. Our results are more general, and apply to arbitrary updates as well as to the much harder adaptive Sample and Hold case.

Other models. Under the regime of "distinct" sampling (also known as L_0 sampling), the aim is to draw a sample uniformly over the set of keys with non-zero counts [15]. This can be achieved under a model allowing increments and decrements of weights by maintaining data structures based on hashing, but this requires a considerable overhead, with factors logarithmic in the size of the domain from which keys are drawn [9, 11].

More recently, the notions of "L_p sampling" and "precision sampling" have been proposed, which allow each key to be sampled with probability proportional to the pth power of its weight ($0 < p < 2$) [20, 1, 17]. These techniques rely on sketch data structures to recover keys and can tolerate arbitrary weight fluctuations. Our problem can be seen as related to the $p = 1$ case. In our setting, we do not allow the aggregate weight of a key to fall below 0. In contrast, the sketch-based techniques for L_p sampling can sample a key whose aggregate weight is negative, with probability proportional to the pth power of the absolute value of the aggregate weight.

The sketch-based sampling techniques have limitations even when the aggregate weights do not fall below zero: They incur space factors logarithmic in the size of the *domain* of keys, which is not favorable for applications with structured key domains (IP addresses, flow keys) that are much larger than the number of active keys. Extracting the sample from the sketches is a very costly process, since it requires enumerating the entire domain for each sketch. Our methods operate in a comparison model, and so can work on keys drawn from arbitrary domains, such as strings or real values (although they do still need to store the sampled keys) and the space usage is proportional to the sample size. The space factors in the sketch-based methods also grow polynomially with the inverse of the bias, whereas our samples are *unbiased*, and thus allow the relative error to diminish with aggregation. Lastly, the weighted sampling performed is "with replacement," which suffers when the weights are highly skewed compared with our "without replacement" sampling. Consequently, sketch-based methods are very slow to work with, and require a lot of space to provide an accurate sample, as we see in our later experimental study.

A different model of deletions arises from the "sliding window" model. In the time-based case, keys above a fixed age are considered deleted, while in the sequence-based case, only the W most recent keys are considered active [2, 3]. However, results in this model are not comparable to the model we study, which has explicit modifications to key weights.

1.2 Our Results

We present a turnstile stream sampling algorithm that efficiently handles signed weighted updates, where the number of cached keys is at most k. The presence of negative updates, the efficient handling of weighted (versus unit) updates, and an adaptive implementation which allows for full utilization of bounded storage, pose several challenges and we therefore develop our algorithm in two stages:

- **SH with signed weighted updates.** (Section 2) We first provide a generalization of Sample and Hold which allows arbitrary updates. When working with weighted updates, we find it convenient to work with a parameter we call the *sampling threshold*, which corresponds to the inverse of the sampling rate when updates are one unit ($\tau \equiv 1/q$). The sampling threshold is an unbiased estimator on the portion of the total value that is not accounted for in c_i.

  We quantify the impact of negative updates on performance, and specifically on the storage used. Although the final sample only depends on v_i, the occupancy of the sample at intermediate points may be much larger, due to keys which are subject to many decrements. We relate the probability that a key is cached at some point but not at the end to the aggregate sum of positive updates.

  When restricted to signed unit weight updates, our algorithm maintains a single counter c_i for each cached key $i \in S$ and the distribution of the counter c_i is exactly the same as (increment-only) SH with value v_i. This gives the benefit of using "off the shelf" SH estimators, and simplifies and improves over the previous efforts of Gemulla *et al.* [13].

- **aSH with signed weighted updates.** In Section 3 we present aSH with signed weighted updates. Our algorithm generalizes reservoir sampling [22] (single unit update for each key), PPSWR sampling (single weighted positive update for each key) [21, 7, 8] and aSH [14, 10, 6, 5] (multiple unit positive updates for each key).

  We work with a bounded size cache which can store at most k keys. At the same time, we want to ensure that we obtain maximum benefit of this cache by keeping it as full as possible. Hence, we allow the "effective" sampling threshold for entering the cache to vary: it increases after processing positive updates but can also decrease after processing negative updates. At the extreme, when negative updates cause ejections so there are fewer than k cached keys, the "effective" sampling threshold is zero.

As a precursor to presenting the adaptive algorithm, we study sampling changes that are randomization-independent. Specifically, we show how to efficiently subsample weighted data, that is, how to modify counter values of cached keys (possibly ejecting some keys) so that we mimic an increase in the sampling threshold from (fixed) τ to a (fixed) $\tau' > \tau$.

The analysis of our aSH with signed weighted updates is delicate and complicated by the dependence of the "effective" sampling threshold on the randomization, on other keys, and on the implementation of "smooth" adaptation of sampling threshold to facilitate ejecting exactly one key when the cache is full. Nevertheless, we show that it provides unbiased estimates with bounded variance that is a function of the effective sampling rate. Along the way, we show how to handle a varying sampling threshold that can decrease as well as increase. When the sampling threshold decreases, we cannot "recover" information about keys which have gone by, but we can "remember" the effective sampling threshold for the keys which are stored in the cache. Then for subsequent keys, a lower threshold makes it is easier for them to enter the cache.

To better understand the behavior of these algorithms, we perform an experimental study in Section 4. We compare the more general solution, aSH with signed weighted updates, to the only comparable technique, L_1 sampling, over data drawn from a networking application. We show that, given the space budget, aSH is more accurate, and faster to process streams and extract samples.

Lastly, in Section 5, we conclude by briefly remarking upon alternate models of stream updates, where instead of incrementing values, a reappearance of a key "overwrites" the previous value. In contrast to many other problems in this streaming model, we note some positive results for sampling.

2. SH WITH SIGNED WEIGHTED UPDATES

We present an extension of the SH procedure to the case when the stream consists of signed, weighted updates. The algorithm maintains a cache S of keys and associated counters c_i, based on a sampling rate $q \equiv 1/\tau$. Increments to key i are handled in a similar way as arrivals in the original (unweighted) SH setting: if i is already in S, then its counter c_i is incremented. Otherwise, i is included in S if Δ exceeds a draw from an exponential distribution with parameter $1/\tau$.

Processing a decrement of key i can be seen as undoing prior increments. If i is not in the cache S, no action is needed. But if i is in the cache, we decrement its associated counter c_i. If this counter becomes negative, we remove i from S. The estimation procedure is unchanged from the increment-only case. This procedure is formalized in Algorithm 1.

To simplify exposition, every key i is associated with a count c_i: keys that are not cached ($i \notin S$) have $c_i = 0$. To show the unbiasedness of this procedure, we derive the distribution of c_i and show that it depends only on v_i, and not any other function of the pattern of updates.

THEOREM 2.1. *The distribution of c_i for a key i with value v_i is*

$$[v_i - \textsc{Exp}_\tau]^+ \tag{2}$$

Algorithm 1 Sample and Hold with signed weighted updates. Sampling rate $q \equiv 1/\tau$

```
1: procedure UPDATE($i, \Delta$)                    ▷ $v_i \leftarrow [v_i + \Delta]^+$
2:     if $i \in S$ then
3:         $c_i \leftarrow c_i + \Delta$
4:         if $c_i \leq 0$ then
5:             $S \leftarrow S \setminus \{i\}$                ▷ eject $i$ from cache
6:     else                                          ▷ case $i \notin S$
7:         $r \leftarrow \textsc{Exp}_\tau$
                      ▷ Exponential distribution with mean $\tau$
8:         if $r < \Delta$ then
9:             $S \leftarrow S \cup \{i\}$, $c_i \leftarrow \Delta - r$

10: procedure SUBSETSUMEST($P$)
           ▷ Estimate $\sum_{i:P(i)} v_i$ for a selection predicate $P$
11:     $s \leftarrow 0$
12:     for $i \in S$ do
13:         if $P(i)$ then
14:             $s \leftarrow s + \tau + c_i$
15: return $s$
```

where $\textsc{Exp}_\tau$ *is exponentially distributed with mean τ. We make use of the notation $[x]^+$ to indicate the function* $\max\{x, 0\}$.

PROOF. We establish the result by computing the changes in c_i in response to the updates of v_i. We consider a fixed key i, whose label we henceforth omit, and let $\{\Delta_{(n)} : n = 1, 2, \ldots\}$ be its updates, yielding corresponding values $v_{(0)} = 0$ and $v_{(n+1)} = [v_{(n)} + \Delta_{(n)}]^+$. The value of v_i at a given time is the cumulative result of the updates that have occurred up to that time.

According to Algorithm 1, the corresponding counter values are $c_{(0)} = 0$ with

$$c_{(n+1)} = \mathrm{I}(c_{(n)} > 0)[c_{(n)} + \Delta_{(n)}]^+ + \mathrm{I}(c_{(n)} = 0)[\Delta_{(n)} - \textsc{Exp}_{(n)}]^+ \tag{3}$$

where the $\{\textsc{Exp}_{(n)} : n = 0, 1, 2, \ldots\}$ are i.i.d. exponential random variables of mean τ, and I is the indicator function. Recall that a counter value $c = 0$ of zero corresponds to the counter not being maintained for the key in question. In particular, $c_{(n)} = 0$ corresponds to no counter being kept prior to the update $\Delta_{(n)}$, while taking the positive part $[c_{(n)} + \Delta_{(n)}]^+$ encodes that when the update due to $\Delta_{(n)}$ yields a non-positive counter value, that counter is deleted from storage.

We show that for each $n = 0, 1, \ldots$,

$$c_{(n)} =^d [v_{(n)} - \textsc{Exp}]^+ \tag{4}$$

where $=^d$ denotes equality in distribution, and $\textsc{Exp}$ is another exponential random variable of mean τ independent of $\{\textsc{Exp}_{(n')} : n' \geq n\}$. (4) is trivially true for $n = 0$. We establish the general case by induction. Assuming (4), then by (3)

$$\begin{aligned} c_{(n+1)} = \tilde{c}_{(n+1)} := &\mathrm{I}(v_{(n)} > \textsc{Exp})[v_{(n)} - \textsc{Exp} + \Delta_{(n)}]^+ \\ &+ \mathrm{I}(v_{(n)} \leq \textsc{Exp})[\Delta_{(n)} - \textsc{Exp}_{(n)}]^+ \end{aligned} \tag{5}$$

To complete the proof, we need to show that

$$\tilde{c}_{(n+1)} =^d c'_{(n+1)} =^d [[v_{(n)} + \Delta_{(n)}]^+ - \textsc{Exp}']^+$$

where $\textsc{Exp}'$ is an independent copy of $\textsc{Exp}$. When $v_{(n)} + \Delta_{(n)} \leq 0$ then $c'_{(n+1)} = \tilde{c}_{(n+1)} = 0$. When $v_{(n)} + \Delta_{(n)} > 0$, the complementary cumulative distribution function (CCDF) of $c'_{(n+1)}$ is

$$\begin{aligned}\Pr[c'_{(n+1)} > z] &= \Pr[\textsc{Exp} < [v_{(n)} + \Delta_{(n)} - z]^+]) \\ &= 1 - e^{-[v_{(n)}+\Delta_{(n)}-z]^+/\tau}, \text{ for } z \geq 0.\end{aligned}$$

The CCDF of $\tilde{c}_{(n+1)}$ can be derived from (5) as

$$\begin{aligned}\Pr[\tilde{c}_{(n+1)} > z] = &\Pr[\textsc{Exp} < \min\{v_{(n)}, [v_{(n)} + \Delta_{(n)} - z]^+\}] + \\ &\Pr[v_{(n)} \leq \textsc{Exp}] \Pr[\textsc{Exp}_{(n)} < [\Delta_{(n)} - z]^+]\end{aligned} \quad (6)$$

When $\Delta_{(n)} < z$, the first term in (6) is

$$\Pr[\textsc{Exp} < [v_{(n)} + \Delta_{(n)} - z]^+] = \Pr[c'_{(n+1)} > z]$$

and the second is zero. When $\Delta_{(n)} \geq z$, then

$$\begin{aligned}&\Pr[\tilde{c}_{(n+1)} > z] \\ &= \Pr[\textsc{Exp} < v_{(n)}] + \Pr[\textsc{Exp} \geq v_{(n)}) \Pr[\textsc{Exp}_{(n)} < \Delta_{(n)} - z] \\ &= (1 - e^{-v_{(n)}/\tau}) + e^{-v_{(n)}/\tau}(1 - e^{-(\Delta_{(n)}-z)/\tau}) \\ &= 1 - e^{-(v_{(n)}+\Delta_{(n)}-z)/\tau} \\ &= 1 - e^{-[v_{(n)}+\Delta_{(n)}-z]^+)/\tau} \\ &= \Pr[c'_{(n+1)} > z]\end{aligned}$$

□

Theorem 2.1 shows that the distribution of the counter c_i depends only on the actual sum v_i, and is exactly captured by a truncated exponential distribution. This is a powerful result, since it means that the sampling procedure is identical in distribution to one where each key is unique, and occurs only once with weight v_i. From this, we can provide unbiased estimators for subset sums and bound the variance. In SubsetSumEst (Algorithm 1), each key is assigned an adjusted weight $\hat{v}_i$, which is $\tau + c_i$ if $i \in S$, and 0 otherwise. Using the convention $c_i = 0$ for deleted counters, this can be expressed succinctly as

$$\hat{v}_i = \mathrm{I}(c_i > 0)(c_i + \tau)$$

Given a selection predicate P, we estimate the subset sum $\sum_{i:P(i)} v_i$ with $\sum_{i:P(i)} \hat{v}_i$, the sum of adjusted weights of keys in the cache which satisfy P. It suffices to show $\hat{v}_i$ is an unbiased estimate of v_i.

Lemma 2.2. *$\hat{v}_i$ is an unbiased estimate of v_i.*

Lemma 2.2 turns out to be a special case of the more general Theorem 3.1 that we state and prove later in context.

2.1 Special case: unit updates

We briefly consider the special case of unit updates, i.e. where each $\Delta \in \{-1, +1\}$. In this case, we can adopt a simplified variant: note that the test of Δ against an exponential distribution succeeds only when $\Delta = +1$, and does so with probability $q = 1/\tau$. When this occurs, we choose to initialize $c_i = 0$. Here, c_i is integral and can be initialized with value ≥ 0 or uninitialized (when i is not cached). It is easy to see the correctness of this procedure: observe that if an i is sampled, then it will only remain in the cache if there is no corresponding decrement later in the stream. Thus we can "pair off" increments which are erased by decrements, and leave only v_i "unpaired" increments. The key i only remains in the cache if it is sampled during one of these unpaired increments, and therefore has count c_i with probability $q(1-q)^{v_i - c_i}$, and is not sampled with probability $(1-q)^{v_i}$. That is, it exactly corresponds to a SH procedure on an increment-only input. Correctness, and unbiasedness of the SubsetSumEst procedure follows immediately as a result. This improves over the result of [13], by removing the need to keep additional tracking counters for cached keys.

2.2 Cache occupancy

From our analysis, it is clear that the distribution of the final count c_i depends only on v_i. In particular, the probability that the key is cached at termination is $1 - \exp(-v_i/\tau)$. With the presence of negative updates in the update stream, however, a key may be cached at some point during the execution and then ejected. The key i is never cached if and only if the independent exponential random variables r generated at line 7 of Algorithm 1 all exceed their corresponding update $\Delta_{(n)}$. This observation immediately establishes:

Lemma 2.3. *The probability that key i is cached at some point during the execution is $1 - \exp(-\Sigma\Delta^+(i)/\tau)$, where $\Sigma\Delta^+(i) \equiv \sum_t \max\{0, \Delta_{(n)}\}$ is the sum of positive updates for key i.*

When $\Sigma\Delta^+(i) \ll \tau$, the probability that a key ever gets cached is small, and is approximately $\Sigma\Delta^+(i)/\tau$. The probability that it is cached at termination is $\approx v_i/\tau$. Summing over all keys, the worst-case cache utilization (which is observed after all negative updates occur at the end), is the ratio of the sum of positive updates to the sum of values, $\sum \Delta^+(i) : v_i$.

3. ASH WITH SIGNED WEIGHTED UPDATES

In this section, we describe an adaptive form of Sample and Hold (aSH), generalized to allow updates that are both weighted and signed, while ensuring that the total number of keys cached does not exceed a specified bound k.

3.1 Increasing the sampling threshold

In order to bound the number of keys cached, we need a mechanism to eject keys, by effectively increasing the sampling threshold τ. We show how to efficiently increase from τ_0 to $\tau_1 > \tau_0$ so that we achieve the same output distribution as if the sampling threshold had been τ_1 throughout. Algorithm 2 shows the procedure for a single key $i \in S$. With probability τ_0/τ_1, no changes are made; otherwise c_i is reduced by a value drawn from an exponential distribution with mean τ_1, and ejected if this is now below 0. We can formalize this as follows. Let $c \geq 0$, $0 \leq \tau_0, \tau_1$, u be a random variable uniformly distributed in $(0, 1]$, and $\textsc{Exp}_{\tau_1}$ be an exponential random variable of mean τ_1 independent of u. Then define the random variable

$$\Theta(c, \tau_0, \tau_1) = \mathrm{I}(u\tau_1 > \tau_0)[c - \textsc{Exp}_{\tau_1}]^+ + \mathrm{I}(u\tau_1 \leq \tau_0)c$$

Note $\Theta(c, \tau_0, \tau_1) = c$ when $\tau_1 \leq \tau_0$. We give an algorithmic formulation of Θ in Algorithm 2 as SampThreshInc, which replaces c_i by the value $\Theta(c_i, \tau_0, \tau_1)$ if this value is positive, and otherwise deletes the key i.

We now show that Θ preserves unbiasedness, and in particular that the distribution of an updated count c_i under Θ is that of a fixed-rate SH procedure with threshold τ_1:

Theorem 3.1. *Let $0 \leq \tau_0 < \tau_1$ and $\tilde{c} = \Theta(c, \tau_0, \tau_1)$.*

Algorithm 2 Adjust sampling threshold from τ_0 to τ_1 for key i

```
Require: $\tau_1 > \tau_0$, $i \in S$
1: procedure SampThreshInc($i, c_i, \tau_0, \tau_1$)
2:     $u \leftarrow$ rand()
3:     if $\tau_1 > \frac{\tau_0}{u}$ then
4:         $z \leftarrow$ rand()
5:         $r \leftarrow (-\ln(z))\tau_1$                    ▷ $r \leftarrow \textsc{Exp}_{\tau_1}$
6:         $c_i \leftarrow c_i - r$
7:         if $c_i \leq 0$ then $S \leftarrow S \setminus \{i\}$
```

(i) $\mathsf{E}[\mathrm{I}(\tilde{c} > 0)(\tilde{c} + \tau_1)|c] = c + \tau_0$

(ii) If $c =^d [v - \textsc{Exp}_{\tau_0}]^+$, *then* $\tilde{c} =^d [v - \textsc{Exp}_{\tau_1}]^+$.

Proof. (i)

$$\begin{aligned}
&\mathsf{E}[\mathrm{I}(\tilde{c} > 0)(\tilde{c} + \tau_1)|c] \\
&= (1 - \tau_0/\tau_1) \int_0^c e^{-x/\tau_1}(c - x + \tau_1)\, dx \; + (\tau_0/\tau_1)(c + \tau_1) \\
&= (1 - \tau_0/\tau_1)c + (\tau_0/\tau_1)(c + \tau_1) = c + \tau_0
\end{aligned}$$

(ii) Let $c =^d [v - \textsc{Exp}_{\tau_0}]^+$. Then $\Theta(c, \tau_0, \tau_1)$ can be rewritten as $[v - W]^+$ where

$$W = \mathrm{I}(\tau_1 u > \tau_0)(\textsc{Exp}_{\tau_0} + \textsc{Exp}_{\tau_1}) + \mathrm{I}(\tau_1 u \leq \tau_0)\textsc{Exp}_{\tau_1}$$

A direct computation of convolution of distributions shows that $\textsc{Exp}_{\tau_0} + \textsc{Exp}_{\tau_1}$ has distribution function

$$x \mapsto (\tau_1 \textsc{Exp}_{\tau_1}(x) - \tau_0 \textsc{Exp}_{\tau_0}(x))/(\tau_1 - \tau_0).$$

Hence, using $\Pr[\tau_1 u > \tau_0] = 1 - \tau_0/\tau_1$, one computes that W has distribution function

$$\begin{aligned}
W(x) &= \Pr[\tau_1 u > \tau_0](\textsc{Exp}_{\tau_0} + \textsc{Exp}_{\tau_1})(x) + \\
&\quad + \Pr[\tau_1 u \leq \tau_0]\textsc{Exp}_{\tau_1}(x) \\
&= \textsc{Exp}_{\tau_1}(x)
\end{aligned}$$

□

Proof of Lemma 2.2.
From Theorem 2.1, $c_i =^d [v_i - \textsc{Exp}_\tau]^+ = \Theta(v_i, 0, \tau)$.
Hence $\mathsf{E}[\hat{v}_i] = \mathsf{E}[\mathrm{I}(c_i > 0)(c_i + \tau)] = v_i$ as a special case of Theorem 3.1(i). □

3.2 Maintaining a fixed size cache

We are now ready to present Algorithm 3, which maintains at most k cached keys at any given time. For each cached key i, the algorithm stores a count c_i and the key's effective sampling threshold τ_i. This τ_i is set to the sampling threshold in force during the ejection following the most recent entry of that key into the cache. It may also be adjusted upward due to the ejection of another key, a process we explain below.

When the cache is not full, a new key i is admitted with sampling threshold $\tau_i = 0$. When the cache is full, in the sense that it contains k keys, a new key is provisionally admitted, then one of the $k + 1$ keys is selected for ejection. The procedure EjectOne adjusts the minimum sampling threshold τ^* to the lowest value for which one key will be ejected. This is implemented by considering the potential action of SampThreshInc on all counts c_i simultaneously, fixing a independent randomization of the variates u_i, z_i for each i, and finding the smallest threshold τ^* for which $\tilde{c}_i = \Theta(c_i, \tau_i, \tau^*) = 0$ for some i. This key is ejected, while the thresholds of surviving keys are adjusted as $(c_i, \tau_i) \leftarrow (\tilde{c}_i, \tau^*)$ if $\tau_i \leq \tau^*$, and are otherwise left unchanged.

Algorithm 3 aSH with signed weighted updates

```
1: procedure Update($i, \Delta$)
                                        ▷ Update weight of key $i$ by $\Delta$
2:     if $i \in S$ then
3:         $c_i \leftarrow c_i + \Delta$
4:         if $c_i \leq 0$ then
5:             $S \leftarrow S \setminus \{i\}$          ▷ $i$ ejected from cache.
6:     else if $\Delta > 0$ then                       ▷ $i \notin S$
7:         $S \leftarrow S \cup \{i\}$, $\tau_i \leftarrow 0$, $c_i \leftarrow \Delta$
                          ▷ Insert $i$ with sampling threshold 0
8:         if $|S| = k + 1$ then
9:             EjectOne($S$)

10: procedure SubsetSumEst($P$)    ▷ Estimate $\sum_{i:P(i)} v_i$
11:     $s \leftarrow 0$
12:     for $i \in S$ do
13:         if $P(i)$ then
14:             $s \leftarrow s + \tau_i + c_i$                ▷ estimate
    return $s$

15: procedure EjectOne($S$)
    ▷ Subroutine to increase sampling threshold so one key is ejected
16:     for $i \in S$ do
17:         $u_i \leftarrow$ rand(), $z_i \leftarrow$ rand()
18:         $T_i \leftarrow \max\{\frac{\tau_i}{u_i}, \frac{c_i}{-\log(z_i)}\}$
               ▷ $T_i$ is the sampling threshold that would eject $i$
19:     $\tau^* \leftarrow \min_{i \in S} T_i$
               ▷ $\tau^*$ is the new minimum sampling threshold
20:     for $i \in S$ do
21:         if $T_i = \tau^*$ then $S \leftarrow S \setminus \{i\}$
22:         else
23:             if $\tau_i \leq \tau^*$ then
24:                 if $\tau^* u_i > \tau_i$ then
25:                     $c_i \leftarrow c_i + \tau^* \log z_i$
        ▷ Given $T_i > \tau^*$ and $\tau^* u_i > \tau_i$, then $c_i + \tau^* \log z_i > 0$
        ▷ Given $T_i > \tau^*$ and $\tau^* u_i \leq \tau_i$, then $c_i$ is unchanged.
26:                 $\tau_i \leftarrow \tau^*$
```

Lemma 3.2. *$\hat{v}_i$ retains its expectation under the action of* EjectOne.

Proof. We choose a particular key i, and fix all c_j, and also fix for $j \neq i$ the random variates u_j, z_j from line 17 of Algorithm 3. This fixes the effect of random past selections and updates. Let $T_j = \max\{\tau_j/u_j, c_j/(-\log z_j)\}$ and $\tau' = \min_{j \neq i} T_j$, which are hence also treated as fixed. Then the update to c_i is $\tilde{c}_i = \Theta(c_i, \tau_i, \tau')$; we will establish that $\tilde{v}_i = \mathrm{I}(\tilde{c}_i > 0)(\tilde{c}_i + \tau')$ is a (conditionally) unbiased estimator of v_i **for any fixed** τ', and hence unbiased.

We first check $\tilde{c}_i$ corresponds to the action on c_i described in Algorithm 3. For clarity we state

$$\tilde{c}_i = \mathrm{I}(\tau' u_i > \tau_i)[c_i - \textsc{Exp}_{\tau'}]^+ + \mathrm{I}(\tau' u_i \leq \tau_i)c_i$$

and observe that $-\tau' \log z_i$ is the random variable $\textsc{Exp}_{\tau'}$ employed. When $T_i < \tau'$, i is selected for deletion. This corresponds to the case $\{\tau' > \tau_i/u_i\} \cap \{\tau' > c_i/(-\log z_i)\}$.

The condition $\tau' > \tau_i/u_i$ selects the first term in the expression for $\tilde{c}_i$, while the condition $\tau' > c_i/(-\log z_i)$ makes that term zero, i.e. $\tilde{c}_i = 0$ and the key is deleted. Otherwise, if $T_i \geq \tau'$, the first or second term in $\tilde{c}_i$ may be selected, but both cannot be zero.

Let $\tilde{v}_i$ denote the estimate of v_i based on $\tilde{c}_i = \Theta(c_i, \tau_i, \tau')$. Then from Theorem 3.1(i)

$$\mathsf{E}[\tilde{v}_i | \tau', c_i > 0] = \mathsf{E}[\mathrm{I}(\tilde{c}_i > 0)(\tilde{c}_i + \tau') | \tau', c_i > 0] = c_i + \tau_i$$

independent of τ'. Hence

$$\mathsf{E}[\tilde{v}_i] = \mathsf{E}[\mathrm{I}(c_i > 0)(c_i + \tau_i)] = v_i.$$

□

Theorem 3.3. *$\hat{v}_i$ is an unbiased estimator of v_i*

Proof. Initially, the cache is empty and both $v_i = 0$ and $\hat{v}_i = 0$. We need to show that the estimate remains unbiased after an update operation. The first part of a positive update $\Delta > 0$ clearly preserves the expectation: τ_i is not modified (or initialized to 0 if i was not cached) and c is increased by Δ. Hence $\hat{v}_i$ increases by exactly Δ. Unbiasedness under the second part, performing EjectOne if the cache is full, follows from Lemma 3.2.

To conclude the proof of the unbiasedness of the estimator, we invoke Lemma 3.6(i), which is stated and proved below. This Lemma uses a 'pairing' argument to pair off the negative update with a prior positive update, and so reduces to a stream with only positive updates, which yields the same $\hat{v}_i$. Therefore, the result follows from the above argument on positive updates. □

3.3 Estimation Variance

It is important to be able to establish likely errors on the unbiased estimators $\hat{v}_i$ resulting from Algorithm 3. A straightforward calculation shows that when $c_i =^d [v_i - \mathrm{Exp}_\tau]^+$, the unbiased estimate $\hat{v}_i = \mathrm{I}(c_i > 0)(c_i + \tau_i)$ has variance

$$\mathsf{Var}[\hat{v}_i] = \tau_i^2 (1 - e^{-v_i/\tau_i})$$

Moreover, $\mathsf{Var}[\hat{v}_i]$ itself has an unbiased estimator s_i^2 that does not depend explicitly on the value v_i:

$$s_i^2 = \mathrm{I}(c_i > 0)\tau_i^2.$$

The intuition behind s_i^2 is clear: for a key in S, the uncertainty concerning v_i is determined by the estimated unsampled increments, which are exponentially distributed with mean τ_i and variance τ_i^2. Observe that both $\mathsf{Var}[\hat{v}_i]$ and s_i^2 are increasing functions of τ_i.

Note that the estimated variance associated with a given key is non-decreasing while the key is stored in the cache, then drops to zero when the key is ejected, possibly growing again after further updates for that key. Since s_i^2 is increasing in τ_i, a simple upper bound on the variance is obtained using the maximum threshold encountered over all instances of EjectOne. This is because one of these instances must have given rise to the largest threshold associated with each particular key.

Lemma 3.4. *For any two keys $i \neq j$, $\mathsf{E}[\hat{v}_i \hat{v}_j]$ is invariant under* EjectOne.

Proof. Reusing notation from the proof of Lemma 3.2, let $T_j = \max\{\tau_j/u_j, c_j/(-\log z_j)\}$. We fix c_j and $\tau' = \min_{h \neq i,j} T_h$. The update to c_h ($h \in \{i,j\}$) is $\tilde{c}_h = \Theta(c_h, \tau_h, \tau')$, and (when $\tilde{c}_h > 0$) the update to τ_h is $\tilde{\tau}_h = \min\{\tau', T_i, T_j\}$. It suffices to show that

$$\begin{aligned}&\mathsf{E}[\mathrm{I}(\tilde{c}_i > 0, \tilde{c}_j > 0)(\tilde{c}_i + \tilde{\tau}_i)(\tilde{c}_j + \tilde{\tau}_j)] \\ &\quad = \mathrm{I}(c_i > 0, c_j > 0)(c_i + \tau_i)(c_j + \tau_j)] \ .\end{aligned}$$

We observe that

$$\begin{aligned}&\mathsf{E}[\mathrm{I}(\tilde{c}_i > 0, \tilde{c}_j > 0)(\tilde{c}_i + \tilde{\tau}_i)(\tilde{c}_j + \tilde{\tau}_j)] \\ &\quad = \mathsf{E}[\mathrm{I}(\tilde{c}_i > 0, \tilde{c}_j > 0)(\tilde{c}_i + \tau')(\tilde{c}_j + \tau')],\end{aligned}$$

which holds because there can be a positive contribution to the expectation only when $T_i, T_j > \tau'$. Under this conditioning, however, τ' essentially functions as a fixed threshold. Thus, we can apply Theorem 3.1 (i) and obtain

$$\begin{aligned}&\mathsf{E}[\mathrm{I}(\tilde{c}_i > 0, \tilde{c}_j > 0)(\tilde{c}_i + \tau')(\tilde{c}_j + \tau')] \\ &\quad = \mathrm{I}(c_i > 0, c_j > 0)(c_i + \tau_i)(c_j + \tau_j).\end{aligned}$$

□

Consequently, we are able to show that the covariance between the estimates of any pair of distinct keys is zero.

Theorem 3.5. *For any two keys $i \neq j$, $\mathsf{Cov}[\hat{v}_i, \hat{v}_j] = 0$.*

Proof. We show inductively on the actions of Update in Algorithm 3 that $\mathsf{E}[\hat{v}_i \hat{v}_j] = v_i v_j$.

For a positive update (i, Δ), the claim is clear following $c \leftarrow c + \Delta$ (line 3 in Algorithm 3), when i is in the cache. Both v_i and $\hat{v}_i = I(c_i > 0)\tau_i + c_i$ increase by Δ and both v_j and $\hat{v}_j$ are unchanged. Hence, if prior to the update we had $\mathsf{E}[\hat{v}_i \hat{v}_j] = v_i v_j$ then this holds after the update. A positive update may also result in EjectOne and we apply Lemma 3.4 to show that $\mathsf{E}[\hat{v}_i \hat{v}_j]$ is unchanged.

To complete the proof, we also need to handle negative updates. We argue that the expectation $\mathsf{E}[\hat{v}_i \hat{v}_j]$ is the same as if we remove the negative update and a prior matching positive update, leaving a stream of positive updates only. The details of this argument are provided in Lemma 3.6(ii). □

A consequence of this property is that the variance on subset sum estimate is the sum of single-key variances, just as with independent sampling: $\mathsf{Var}[\sum_{i:P(i)} \hat{v}_i] = \sum_{i:P(i)} \mathsf{Var}[\hat{v}_i]$. In particular, $\sum_{i:P(i)} \mathrm{I}(c_i > 0)\tau_i^2 = \sum_{i \in S:P(i)} \tau_i^2$ is an unbiased estimate on $\sum_{i:P(i)} v_i$.

3.4 Analysis of negative updates under aSH

Our analysis in Theorems 3.3 and 3.5 relies on replacing a stream with a mixture of positive and negative weight updates with one which has positive weight updates only, and then arguing that certain properties of this positive stream match those of the original input. This generalizes the "pairing" argument outlined in Section 2.1 for the special case of unit weighted updates. In this section, we provide more details of this pairing transformation, and prove that this preserves the required properties of the estimates.

Pairing Process. In our pairing process, each negative update is placed in correspondence ("paired") with preceding, previously unpaired, positive updates. We then go on to relate the state to that on a stream with both paired updates omitted. Figure 1 gives a schematic view of the pairing process for a key, which we describe in more detail below.

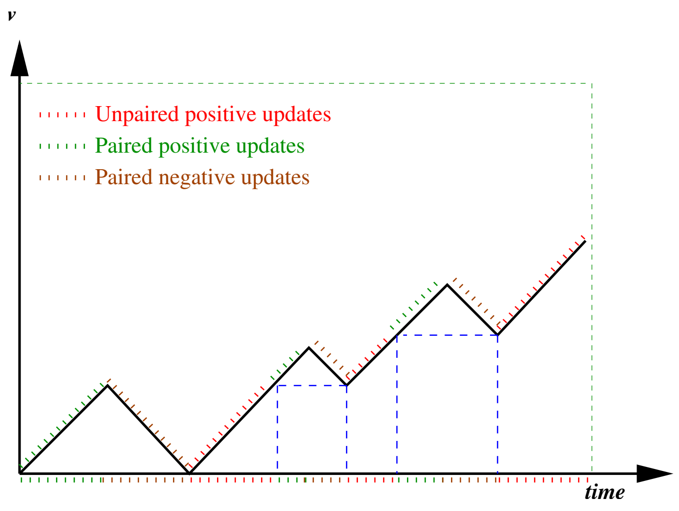


Figure 1: Value of key with time. Unpaired and paired positive update and paired negative updates are marked.

Specifically, a negative update of key i is paired with the most recent unpaired positive update of i. If there is no such update, the negative update is unpaired. This occurs when the aggregate weight before the update is zero, in which case according to (1), v_i remains zero, and we can ignore this update.

Since updates are weighted, a negative update $(i, -\Delta)$ may be paired with multiple positive updates, or with a fraction of a positive update. Without loss of generality, each negative update will be paired with a "suffix" of a positive update, followed by zero or more "complete" positive updates. If $v_i < \Delta$, the negative update only has its v_i "prefix" paired and $v_i - \Delta$ "suffix" unpaired.

To aid analysis, we observe that we can "split" updates without altering the state distribution. More precisely, let σ be an update stream terminating with an update (i, Δ). Let σ' be the same stream with the last update replaced by two consecutive updates (i, Δ_1) and (i, Δ_2) such that $\Delta_1 + \Delta_2 = \Delta$, and $\text{sign}(\Delta_1) = \text{sign}(\Delta_2) = \text{sign}(\Delta)$. Then the distribution of the state of the algorithm after processing σ or σ', conditioned on the initial state, is the same. This means that by splitting updates, we can replace the original sequence by another sequence (where each original update is now a subsequence of updates), on which the algorithm has the same final state but all updates are fully paired or fully unpaired.

We next prove the lemma we require to replace the original stream with a positive-only stream.

LEMMA 3.6. *Consider an update stream where the ℓ'th update is a negative update $(i, -\Delta)$ and a modified stream with the negative update, and all positive update(s) paired with it (if any) omitted. Then*

(i) $\mathsf{E}[\hat{v}_i]$ and

(ii) for $j \neq i$, $\mathsf{E}[\hat{v}_i \hat{v}_j]$

are the same for both streams.

PROOF. We observe that a negative update can be unpaired if and only if $v_i = 0$ just prior to the current negative update. Since the counter $c_i \leq v_i$, this means that i is not cached and thus the processing of the unpaired negative update has no effect on the state of the algorithm.

For paired negative updates, let σ_0 be the prefix of the stream up to the corresponding paired positive update, σ be the suffix including and after the positive update, and $\sigma' \subset \sigma$ be an alternative suffix with the negative update and its paired positive update all omitted. Fix the execution of the algorithm on σ_0. To complete the proof, we show that

(i) The distributions of (τ_i, c_i) for key i after processing σ and σ' are the same.

(ii) For a key $j \neq i$, if $c_i > 0$, then the distribution of (τ_j, c_j) conditioned on the final (τ_i, c_i) when $c_i > 0$ is the same: $(\tau_j, c_j)|\sigma, \tau_i, c_i =^d (\tau_j, c_j)|\sigma', \tau_i, c_i$

Since the actual splitting point of updates does not affect the state distribution, we can treat v_i as defined by the update stream projected on i as a contiguous function of "time". Initially, before any updates of i are processed, the time is 0. If the time is t_0 prior to processing an update (i, Δ), then for $t \in [t_0, t_0 + |\Delta|]$, we interpolate the value v_i as a function of time as $v_i(t) = [v_i(t_0) + t - t_0]^+$. After processing the update, the time is $t_0 + |\Delta|$. Any point in time now has a gradient, depending on whether v_i is increasing or decreasing at that time. Pairing is between intervals of equal size and opposite gradient. Figure 1 shows the value v_i as a function of time and distinguishes between updates that are paired and unpaired with respect to the final state.

Tracking the state of key i during execution, we map an interval $[0, c_i]$ to positive update intervals in a length-preserving manner. The mapping corresponds to the positive updates which contributed to the count c_i, that is, the most recent unpaired positive update intervals of (total) length c_i. We refer to the point that 0 is mapped to as the (most recent) *entry point* of key i into the cache. Upon a positive update, the counter is incremented by $\Delta > 0$. The mapping of $[0, c_i]$ is unchanged and the interval $[c_i, c_i + \Delta]$ maps to the interval $[t_0, t_0 + \Delta]$ of the current update on the time axis. An EJECTONE operation which reduces the counter c_i by r removes an r "prefix" of the counter interval: a positive update time previously paired with $y \in [0, c]$ such that $y > r$ is paired with the point $y - r$ in the updated interval. A negative update of Δ results in $c \leftarrow [c - \Delta]^+$. A Δ "suffix" of $[0, c]$ is removed (reflecting that the positive update points this suffix was mapped to are now paired) and the mapping of the remaining $[0, [c - \Delta]^+]$ prefix is unchanged. Note that the positive update interval removed is the one that is paired with the negative update just processed.

We now consider a negative update of Δ, and compare the execution of the algorithm on suffixes σ and σ'.

- If i is not cached after σ_0, then it would not be cached after any execution of either σ or σ', since all positive updates of i are paired. If i was cached at the end of σ_0 and ejected during σ, we fix the randomization in each EJECTONE performed when processing σ. We then look at executions of σ' that follow that randomization until (and if) the set of cached keys diverges. Under σ', i has a lower c value (by Δ) than under σ. Thus, when i is removed by an EJECTONE at σ, it would also be removed under σ', unless it was removed already under a prior EjectOne. As all positive updates of i in σ and σ' are paired, once i is removed, then even if it reenters the cache it will not be cached at termination for either σ' or σ. Thus, the outcome for i under σ and σ' is the same.

- If i is cached after σ, its most recent entry point must be in σ_0. We again fix the execution of the algorithm on σ, fixing all random draws when performing EjectOne procedures. We will show that if the execution of σ' follows these random draws, then i will not be ejected and will have the same τ_i, c_i values at termination of σ'. Consider the execution of the algorithm on σ' and let τ'_i, c'_i be the parameters for i. Initially, $c'_i = c_i - \Delta$. Clearly, any EjectOne where c is not changed, retains i under both σ and σ'. If c is reduced under σ, it still must be such that $c_i > \Delta$ for i to be cached at the end, since all subsequent positive updates are paired, and the last update is of $-\Delta$. Thus, it would also remain cached under the corresponding EjectOne in σ'. Moreover, all other keys cached under σ would also be cached (and have the same (τ_j, c_j) values) under σ'.

Lastly, we consider (τ_j, c_j) for $j \neq i$ conditioned on (τ_i, c_i) when $c_i > 0$. If i is not cached after σ, it is also not cached under σ' and thus $c_i = 0$. If i is cached after σ, it has the same state (τ_i, c_i) as under σ' (fixing randomization). Furthermore, the state (τ_j, c_j) for any key and $j \neq i$ is the same.

From these observations, we can conclude that the expectation of the estimate for any key, and for any pair of distinct keys (and likewise for higher moments), are the same for both streams. $\square$

Note that once we have the above lemma, we can apply it repeatedly to remove all negative updates, and result in a "positive-only" stream. The distributions of the state of algorithm on these paired streams are identical.

4. EXPERIMENTS

In this section we exhibit the action of aSH on a stream of signed updates, and compare its performance against a comparable state-of-the-art method, namely L_p sampling.

4.1 Implementation Issues.

Our reference comparison was with L_p sampling. The L_p sampling procedure builds on the idea of sketch data structures, which summarize a d-entry vector x. A sketch built with parameter ε can provide an estimate of any x_i with error $\varepsilon\|x\|_2$, using space $O(\varepsilon^{-2} \log d)$ [4]. In outline, for L_1 sampling, we build a sketch of the vector $x_i = v_i/u_i$, where v_i are the aggregate weights for key i, and each u_i is a fixed uniform random variable in the range $[0, 1]$ [17]. To draw a sample, we find the entry of this vector which is estimated as having largest x_i value by the sketch. To obtain the correct sampling distribution, we have to reject samples whose x_i value does not exceed a fixed threshold as a function of $\|v\|_1$, which in our case is just the sum of all update weights. As a result, the procedure requires a lot of space per sample: $O(\log^2 d)$, even using a large constant value of ε. Further, the time to extract a sample can be high, since we have to enumerate all keys in the domain. The search time could be reduced using standard tricks, but this would increase the space by further factors of $\log d$. We implemented both L_1 sampling and our adaptive Sample and Hold algorithm in interpreted languages (Perl, Python); we expect optimized implementations to have the same relative performance, but improved scalability. To ensure comparability, we fix the space used by aSH to draw a sample of size k (i.e. k tuples of $(i, c_i, \tau_i, u_i, z_i)$), and allocate this much space to L_p sampling to be used for sketch data structures. Based on our parameter settings, the sketch used by L_p sampling to extract a single sample equates to the space for 10 samples under aSH.

4.2 Data Description

We tested our methods on data corresponding to the ISP network monitoring scenario described in the Introduction. The update stream was derived from arrival and departures at a queue. The queue was driven by a synthetic arrival process, comprising keyed items with sizes drawn independently from a Pareto(1.1) distribution. The arrival times and key values were derived from an IP network trace of 10^6 packets. We simulated a FIFO queue in which each item remains at the head of the queue for a time equal to its size, and converted the trace into a stream of 2×10^6 updates (i, s), where i is a key and s is positive for an arriving item and negative of the same magnitude when that item has completed service for a departure, the stream items having the corresponding event order as for the queue. Thus the values v_i correspond to the total size of items of key i in the queue at a given time. We evaluate the methods based on their ability to accurately summarize the queue occupancy distribution of different subsets of keys.

4.3 Parametrization

We parametrized different runs of the simulation as follows.

- *Key Granularity:* The trace contained about 110,000 unique keys. These were mapped to smaller keys sets as $i \mapsto \beta = h(i) \mod b$ with h a hash function and b taking values $1,000$, $10,000$ and $100,000$. This is the granularity of keys in the sample set. Note that the time for our aSH algorithm does not depend explicitly on the key granularity, but L_p sampling incurs a cost linear in b; hence, we focus on moderate values of b to bound the experimental duration.

- *Query Granularity:* For determination of estimation accuracy, we divide the keyspace into a different classes, and use the recovered sample to estimate the weight distribution across these classes. Specifically, we assigning each key β to a bin by $\beta \mapsto \alpha = \beta \mod a$. (Note that the sampling methods are not aware of this fixed mapping, and so are unable to take advantage of it.) We used values $10, 100$, and $1,000$ for a.

- *Sample Size:* we selected roughly geometric sample sizes k from $30, 100, 300$ and $1,000$ for aSH, and allocated the equivalent space to L_p sampling, as described above.

Our subsequent experiments compare the performance in time and accuracy across the different combinations of these settings of a, b, and sample size k.

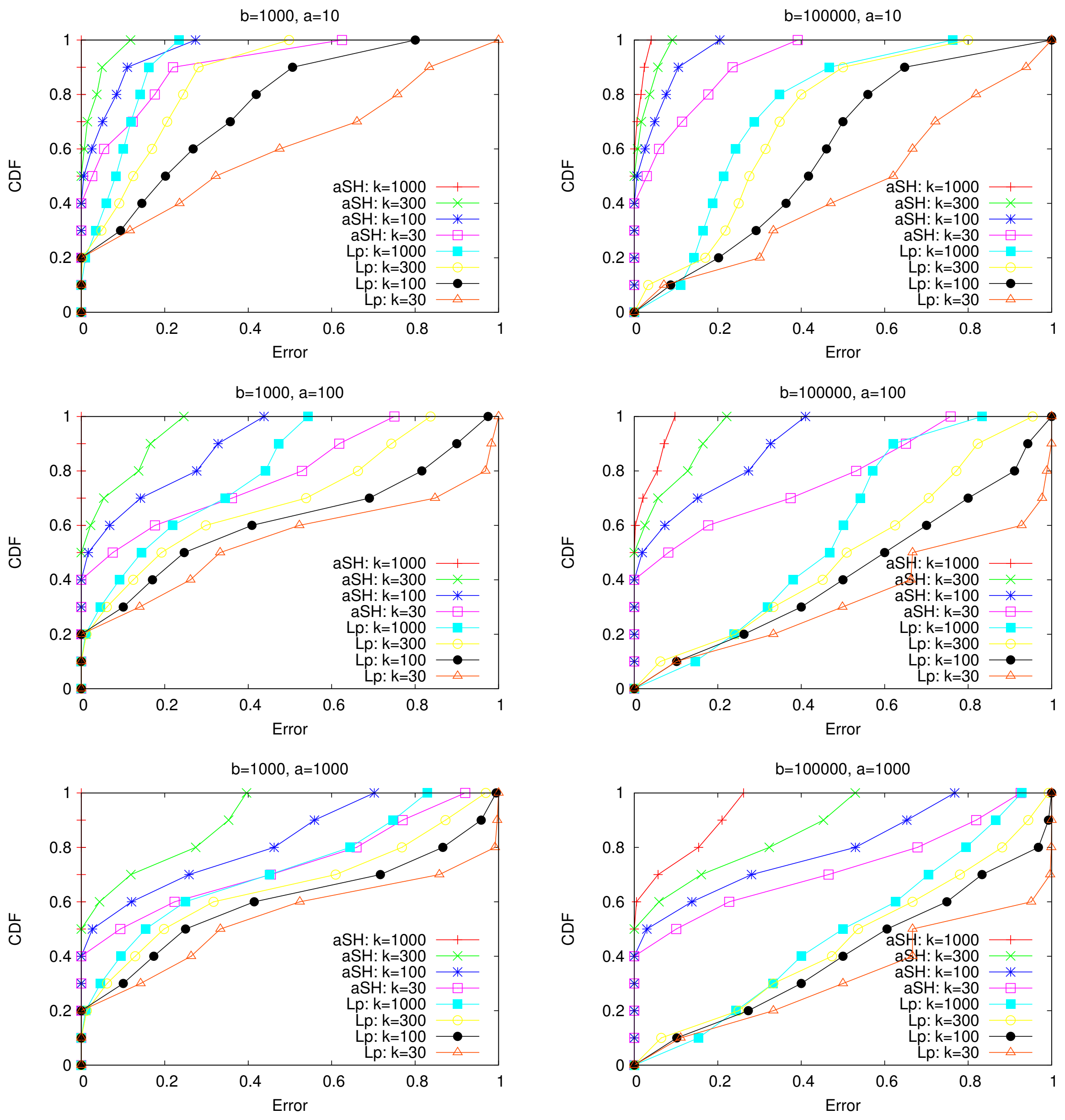


Figure 2: CDF of Estimation Error as a function of sample size k for aSH and L_p; key granularity $b = 1,000$ and $100,000$. Query granularity $a = 10, 100$ and $1,000$. aSH is consistently more accurate than L_p.

4.4 Results

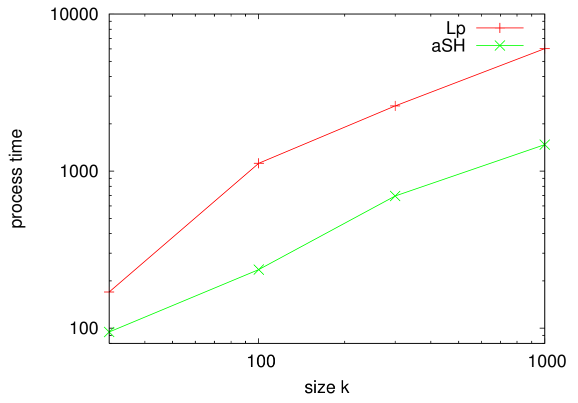
Query Accuracy. Each time we probe the samples, we obtain the estimated weights of the a different bins. We measure the accuracy by comparing the induced frequency distribution to the true distribution computed offline. That is, let $w(\alpha)$ and $\hat{w}(\alpha)$ the total and estimated weighted in each query aggregate labeled by $\alpha = 1, 2 \ldots a$. Then accuracy is measured by the distribution error

$$\frac{1}{2} \sum_{\alpha} \left| \frac{w(\alpha)}{W} - \frac{\hat{w}(\alpha)}{\hat{W}} \right| \in [0, 1] \tag{7}$$

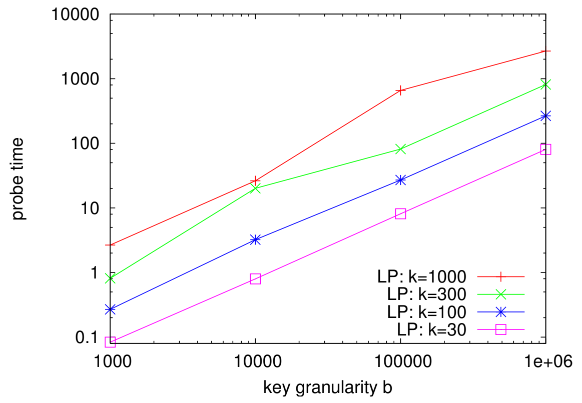
where $W = \sum_\alpha w(\alpha)$ and $\hat{W} = \sum_\alpha \hat{w}(\alpha)$.

This distribution error was computed for each positive update for aSH and every 1000 updates in L_p sampling. Figure 2 shows the empirical cumulative distributions of the errors computed for each method, using key granularity $b = 1,000$ and $100,000$ and query granularity $a = 10, 100$ and $1,000$.

On these plots, an ideal summary reaches the top left hand corner (i.e. if all queries have zero error). For aSH with k=1000, this is indeed achieved in the case $b = 1000$. That is, when there are only 1000 distinct keys, the sample of size 1000 is able to represent this distribution exactly, and hence can answer any query perfectly. In more realistic situations, k is less than b (often, much less than b). However, we see even for $k = 100$, corresponding to 10% of the active keys, we see that aSH does a good job of answering the query. This holds even for $b = 100,000$, meaning the sample can capture



(a) Update Processing Time (in seconds) as a function of sample size k for aSH and L_p.



(b) Query Probe Time (in seconds) as a function of key granularity b for L_p.

Figure 3: Timing results for stream processing and querying

only 0.1% of the keyspace. We note the accuracy tends to improve as a decreases, meaning that there are fewer bins for the query, and hence more keys in each bin. In this case, almost every query is answered with high accuracy.

In contrast, the accuracy from L_p sampling is quite low. The main reason is the much higher space overhead of this method. As noted above, even with space corresponding to $k = 1000$, L_p sampling only has room to recover up to 100 samples. Moreover, each attempt to recover a sample from the sketch fails with constant probability (this is an integral part of the method, which is needed to achieve the correct sample distribution). In practice, we were only able to recover a small number of samples, often only in the single figures. As a result, we observe that L_p sampling with space for $k = 1000$ is often outperformed by aSH with $k = 30$, i.e. aSH is more accurate even with a much smaller summary size.

Update Processing Time. Both aSH and L_p sampling were run using interpreted languages on the same system. We measured the total processing time associated with processing the update stream. The update processing times is largely independent of key granularity, and was observed to grow roughly linearly as a function of the sample size k; see Figure 3(a) which shows the cost on logarithmic axes. Under this implementation, L_p sampling is roughly half an order magnitude (i.e. five times) slower than aSH.

Asymptotically, L_p sampling is truly linear in k: it updates $O(k)$ different sketches, taking constant time for each. Hence, we do not expect to be able to improve substantially on this. The time cost for aSH includes the time to perform EjectOne for every update. We note that the update time could be improved for aSH, at the expense of slightly increased space, by deferring this step. That is, suppose we buffer c arrivals, then perform c steps of EjectOne, one after another. This can be shown to be equivalent to determining a new threshold τ_c^* which is sufficient to eject c keys. Then we can perform a single pass over the $c + k$ keys and track the c'th largest threshold in a heap. Thus, setting $c = O(k)$ means that the update time could be reduced to $O(\log k)$, allowing larger samples to be drawn efficiently.

Query Answering Time. For the aSH summary, answering queries is trivial: we just have to read the explicitly maintained counts and thresholds, and use these within the SubsetSumEst routine. In our experiments, this time was essentially nil. However, this is not the case for the L_p sampling case, due to the need to test the sketch-estimated count each potential key value. Figure 3(b) displays the time to answer perform this extraction as a function of key granularity b for different buffer sizes k. The results are in accordance with the asymptotic cost, growing as $O(bk)$. At the extreme end, the cost is of the order of 45 minutes to extract a sample over a domain of $b = 10^6$, making real time analysis infeasible. We would argue that this is not a particularly large domain (e.g. IP addresses span a domain 3 orders of magnitude larger). From this, we conclude that the time cost of working with L_p sampling is too high for anything above a moderate sized domain. In contrast, the time cost for working with aSH sampling is independent of b.

5. CONCLUDING REMARKS

In many situations, data is presented in the form of streams of updates which can include negative weights. When these streams become large, it is important to be able to accurately maintain a sample over them. We have introduced an aSH sampling technique for such streams of weighted, signed updates, which is unbiased and has zero co-variance between estimated keys. The procedure is easy to implement. In our experimental study, we observed that its performs substantially better than the L_p sampling technique based on sketches: given the same amount of space, aSH is more accurate, faster to process the stream, and dramatically faster to answer queries.

Looking beyond this work, we note that other models of streaming arrivals are also of interest, but have similarly received relatively little attention. One important case is that of the *overwrite streams* model, where an update of the form (i, v) means that the weight associated with key i is updated to v. The update $(i, 0)$ corresponds to deletion of the key. This model captures a simple notion of "updating" information about a key. It is also of interest since many natural streaming problems are hard in this setting, meaning that they require space linear in the size of the input; see e.g. [16].

In contrast, a sampling algorithm for constant rate $q = 1/\tau$ is straightforward: if a key is observed which is already stored in the cache, eject it. Then independently decide whether to retain the current key, based on its weight v and τ. Correctness—i.e., that this is distributed as a Poisson sample on the final values for each key—is immediate. It is more challenging to manage a fixed-sized cache S. Here, again, overwrites deplete the cache, but arrivals can cause the cache size to be exceeded. In this case, we must choose a key to eject, in a similar way to EjectOne, based on a local τ_i for each. We leave further study of this model to future work.

6. REFERENCES

[1] A. Andoni, R. Krauthgamer, and K. Onak. Streaming algorithms from precision sampling. Technical Report 1011.1263, arXiv, 2010.

[2] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 633–634, 2002.

[3] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. In *ACM Principles of Database Systems*, 2009.

[4] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

[5] E. Cohen, N. Duffield, H. Kaplan, C. Lund, and M. Thorup. Algorithms and estimators for accurate summarization of Internet traffic. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (IMC)*, 2007.

[6] E. Cohen, N. Duffield, H. Kaplan, C. Lund, and M. Thorup. Sketching unaggregated data streams for subpopulation-size queries. In *Proc. of the 2007 ACM Symp. on Principles of Database Systems (PODS 2007)*. ACM, 2007.

[7] E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *Proceedings of the ACM PODC'07 Conference*, 2007.

[8] E. Cohen and H. Kaplan. Tighter estimation using bottom-k sketches. In *Proceedings of the 34th VLDB Conference*, 2008.

[9] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *International Conference on Very Large Data Bases*, 2005.

[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM'02 Conference*. ACM, 2002.

[11] G. Frahling, P. Indyk, and C. Sohler. Sampling in dynamic data streams and applications. In *Symposium on Computational Geometry*, June 2005.

[12] R. Gemulla, W. Lehner, and P. J. Haas. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *VLDB*, pages 595–606, 2006.

[13] R. Gemulla, W. Lehner, and P. J. Haas. Maintaining bernoulli samples over evolving multisets. In *PODS 2007*, pages 93–102. ACM, 2007.

[14] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*. ACM, 1998.

[15] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 2001.

[16] M. Hoffmann, S. Muthukrishnan, and R. Raman. Streaming algorithms for data in motion. In *ESCAPE*, pages 294–304, 2007.

[17] H. Jowhari, M. Saglam, and G. Tardos. Tight bounds for Lp samplers, finding duplicates in streams, and related problems. In *PODS*, pages 49–58, 2011.

[18] D. E. Knuth. *The Art of Computer Programming, Vol 2, Seminumerical Algorithms*. Addison-Wesley, 2nd edition, 1998.

[19] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *International Conference on Very Large Databases (VLDB)*, pages 346–357, 2002.

[20] M. Monemizadeh and D. P. Woodruff. 1-pass relative-error l_p-sampling with applications. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 2010.

[21] B. Rosén. Asymptotic theory for successive sampling with varying probabilities without replacement, I. *The Annals of Mathematical Statistics*, 43(2):373–397, 1972.

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