Continuous Sampling from Distributed Streams*

Graham Cormode
AT&T Labs–Research
graham@research.att.com

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Ke Yi
HKUST
yike@cse.ust.hk

Qin Zhang†
MADALGO‡
Aarhus University
qinzhang@cs.au.dk

October 6, 2011

Abstract

A fundamental problem in data management is to draw and maintain a sample of a large data set, for approximate query answering, selectivity estimation, and query planning. With large, streaming data sets, this problem becomes particularly difficult when the data is shared across multiple distributed sites. The main challenge is to ensure that a sample is drawn uniformly across the union of the data while minimizing the communication needed to run the protocol on the evolving data. At the same time, it is also necessary to make the protocol lightweight, by keeping the space and time costs low for each participant. In this paper, we present communication-efficient protocols for continuously maintaining a sample (both with and without replacement) from k distributed streams. These apply to the case when we want a sample from the full streams, and to the sliding window cases of only the W most recent elements, or arrivals within the last w time units. We show that our protocols are optimal (up to logarithmic factors), not just in terms of the communication used, but also the time and space costs for each participant.

1 Introduction

It is increasingly important for data management systems to cope with large quantities of data that are observed at geographically distributed locations. As data volumes increase (through greater power of measurement in sensor networks, or increased granularity of measurements in network monitoring settings), it is no longer practical to collect all the data together in a single location and perform processing using centralized methods. Further, in many of the motivating settings, various monitoring queries are lodged which must be answered *continuously*, based on the total data that has arrived so far. These additional challenges have led to the formalization of the *continuous distributed streaming model* [8]. In this model, defined more precisely below, a number of distributed peers each observe a high-speed stream of data, and collaborate with a centralized coordinator node to continuously answer queries over the union of the input streams.

*A preliminary version of the paper was presented at the *ACM Symposium on Principles of Database Systems*, June 2010.

†Most of this work was done while Qin Zhang was a Ph.D. student in the Hong Kong University of Science and Technology (HKUST).

‡Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

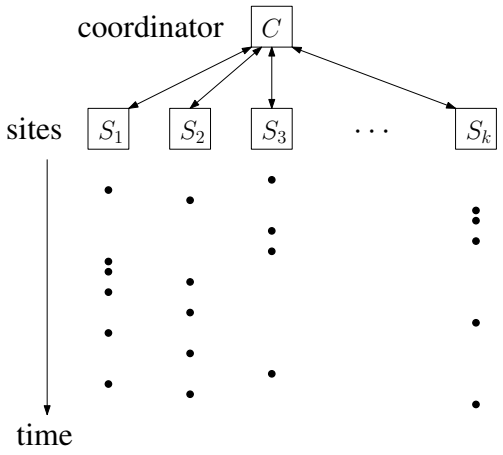

Figure 1: The continuous distributed streaming model.

Protocols have been defined in this model for a number of classes of queries, which aim to minimize the communication, space and time needed by each participant. However, the protocols proposed so far have overlooked the fundamental problem of maintaining a sample drawn from the distributed streams. A sample is a powerful tool, since it can be used to approximately answer many queries. Various statistics over the sample can indicate the current distribution of data, especially if it is maintained to be drawn only from a recent history of data. In this paper, we present techniques for maintaining a sample over distributed data streams either with or without replacement, and show how to adapt them for sliding windows of recent data.

In doing so, we build on the long history of drawing a sample from a single stream of elements. Random sampling, as a fundamental problem and a basic tool for many applications, had been studied in the streaming setting long before formal models of data streams were first introduced. The classical *reservoir sampling* algorithm [23] (attributed to Waterman) maintains a random sample of size s without replacement over a stream. It is initialized with the first s elements; when the i-th element arrives for $i > s$, with probability $1/i$ it adds the new element, replacing an element uniformly chosen from the current sample. It is clear that this algorithm uses optimal $O(s)$ space and $O(1)$ time per element.

There have been various extensions to the basic reservoir sampling algorithm. Using an appropriate distribution, it is possible to determine how many forthcoming elements to "skip" over until the next sample will be drawn [28, 32]. It has been generalized to allow weighted items [12]. Gibbons and Matias introduced concise samples and counting samples, to make better use of the space available [18]. "Distinct samples" aim to draw a sample from the support set of the multi-set of items in a stream, possibly containing deletions [13, 17]. "Priority Sampling" aims to reduce the variance on subset-sum queries [10]. There has also been work on sampling from streams which include deletions [15, 16].

More recently, there has been much interest in understanding how to maintain a uniform sample efficiently over a *sliding window* [2, 4, 14]. There are two models for sliding windows: in *sequence-based windows*, we must maintain a sample over the last W elements in the stream; in *time-based windows*, every element arrives at a particular time (called its *timestamp*), and we want to maintain a sample over the elements that have arrived in the time interval $[t - w, t]$ for a window length w, where t denotes the current time. Time-based windows are usually more useful than sequence-based windows but often require more complex algorithms and more space and time resources to handle.

Continuous distributed streaming. Many streaming applications [26] involve multiple, say k, streams distributed in different locations linked by a network. The goal is to continuously track some function at a designated coordinator over the combined data received from all the streams, as opposed to just one. This is shown schematically in Figure 1. For example, consider a collection of routers in a network, each of which processes a high-speed stream of packets. Maintaining a random sample of the packets from the union of these streams is valuable for many network monitoring tasks where the goal is to detect some global features [21]. Beyond network monitoring, similar problems also arise naturally in applications like distributed databases, telecommunications, web services, sensor networks, and many others.

In this setting, the communication cost is the primary measure of complexity of a tracking algorithm, but its space and time costs are also important to bound. Motivated by the many applications in networking and databases, there has been a lot of work on designing communication-efficient algorithms for tracking certain functions (including frequent items [3, 22, 24, 33], quantiles [7, 33], frequency moments [6, 8], various sketches [7, 9], entropy [1], and other non-linear functions [29, 30]) over distributed streams. But surprisingly, the important and fundamental problem of random sampling has not yet been addressed.

Problem definition. We formally define our problem as follows. Let $A = (a_1, \ldots, a_n)$ be a sequence of elements. The sequence A is observed in an online fashion by k remote *sites* $S_1, \ldots, S_k$ collectively, i.e., a_i is observed by exactly one of the sites at time t_i, where $t_1 < t_2 < \cdots < t_n$. It is assumed that a_i arrives with its timestamp t_i, but the index i, namely its global sequence number, is unknown to the receiving site. Let $A(t)$ be the set of elements received up until the current time t from all sites. There is a designated *coordinator* C who has a two-way communication channel to each of the k sites and needs to maintain a random sample of size s (with or without replacement) of $A(t)$ at all times. We also consider two sliding-window versions of the problem. In a *sequence-based window*, for a given window size W, the coordinator needs to maintain a random sample over the last W elements received across all sites; in a *time-based window*, for a given window duration w, the coordinator maintains a random sample over the set $A(t) \setminus A(t-w)$ at all times t.

In the above setting, our primary concern is the total communication cost between the coordinator and the k sites. There are no other direct communications allowed between sites; but up to a factor of 2 this is not a restriction. We assume that communication is instantaneous. Other than communication we also care about the space/time costs for the coordinator as well as for each site.

Note that the challenge of our problem arises from the combination of distributed data and the requirement that the sample be maintained continuously. Indeed, if either one is missing, the problem becomes trivial. In the centralized setting it is the reservoir sampling problem. For one-time sampling, each site can first report the size of its own data set. Based on these sizes and the size of the combined data set, the coordinator decides (randomly) how many samples are allocated to each of the k sites, and then simply asks them to get the samples. The communication cost is $O(k + s)$. The main obstacle in extending these ideas to the continuous distributed streaming model is that the current value of i (the total number of elements that have arrived) is not known. In fact, tracking i exactly requires every site to notify the coordinator upon the arrival of each element, costing $\Theta(n)$ communication. In the standard streaming model with a single stream, i is trivial to track, and algorithms in this model can rely on this knowledge. For example, the reservoir sampling algorithm samples the i-th element with probability $1/i$. Similarly, the sliding-window algorithms of [2, 4], track exactly how many elements have arrived since a "landmark" time T. One approach would be to use existing methods to track i approximately [8, 22]. But this does not immediately yield efficient algorithms. In addition, any sample maintained will be somewhat approximate in nature: some elements will be more likely to be sampled than others. Such non-uniformity is undesirable, since it is unclear how this error will

window	communication	coordinator space	coordinator total time	site space	site time (per element)
infinite	$k \log_{k/s} n + s \log n$	s	$k \log_{k/s} n + s \log n$	1	1
sequence-based	$ks \log W$	s	$ks \log W$	1	1
time-based	$(k+s) \log W$	$s \log W$	$(k+s) \log W$	$s \log W$	1

Table 1: Our algorithms' asymptotic costs for sampling without replacement over distributed streams. In this paper we define $\log x = \log_2 x$ and $\log_x y = 0$ if $x \leq 1$. The bounds for time-based windows are higher than the lower bounds by a log factor; all the other bounds match the lower bounds.

impact approximations based on the sample, and how this will propagate in various applications.

Our results. In this paper, we present algorithms that maintain a true random sample (i.e., no approximation) over distributed streams, without explicitly tracking i. Our asymptotic bounds are summarized in Table 1 for sampling without replacement under different settings. We measure the communication and space costs in terms of *words*, and assume that each element, as well as any quantity polynomial in k, s and n can be represented in $O(1)$ words. We also require each communicated message to be at least one-word long. For the two sliding-window cases, the bounds are for each window where W denotes the number of elements in the window; note that W may vary from window to window in the time-based case. We do not show the results for sampling-with-replacement, since the bounds are quite similar, often at most a logarithmic factor worse.

Our algorithms for infinite windows (full streams) and sequence-based windows are optimal simultaneously in *all* the five measures in Table 1 (for sufficiently large n and W), while the algorithms for time-based windows are optimal up to a logarithmic factor. Some bounds are clearly optimal, such as a site taking constant time to process each new element and the coordinator requiring $\Omega(s)$ space, while others are less obvious. We prove the lower bounds after presenting the algorithms for each case.

In the centralized setting, time-based windows are usually more difficult to handle than sequence-based windows because the number of active elements can vary dramatically over time. There is a space lower bound of $\Omega(s \log W)$ [14] for time-based windows while sequence-based windows only need space $O(s)$ [2, 4]. One interesting observation from Table 1 is that in the distributed setting, time-based windows turn out to be *easier* than sequence-based windows, and there is a quadratic difference (when $k \approx s$) in terms of communication, while the space bounds match those for the centralized case. The fundamental reason is that it is much harder to determine whether an element a_i has "expired" in the sequence-based case, since we do not have the global sequence number i. Meanwhile, in the time-based window case, expiration can be determined by comparing an element's timestamp to the current time. A formal proof of this hardness is given in Theorem 4.2.

Note that the continuous distributed streaming model degenerates into the standard streaming model if we set $k = 1$ and ignore the communication aspect. When restricted to this case, our algorithms achieve the same bounds as the previous centralized streaming algorithms over infinite-streams (reservoir sampling), as well as the two sliding-window cases [2, 4]. So our algorithms generalize previous techniques with the same space/time bounds while achieving optimal communication. Note however that the space bounds in [4] are worst-case, while ours and those in [2] are probabilistic.

All our algorithms are explained via a simple concept, which we call *binary Bernoulli sampling*. We describe this method in Section 2. Then we present our sampling algorithms (with and without replacement) for the three cases in Table 1 in Section 3, 4, and 5, respectively.

Applications. Our results immediately yield protocols to track a number of interesting functions in the distributed streaming setting. Some of them improve (for certain parameter ranges) previous results while others are new.

Tracking the frequent items (a.k.a. the *heavy hitters*) and quantiles (approximately) over distributed streams has received a lot of attention [3, 7, 22, 24, 33]. There is a deterministic algorithm that costs $\tilde{O}(k/\epsilon)$ communication[1], which is optimal [33], where ϵ is the approximation error. On the other hand, it is well known that a random sample without replacement of size $\tilde{O}(1/\epsilon^2)$ can be used to extract these statistics with high probability. Our result immediately gives a probabilistic algorithm for tracking the heavy hitters and quantiles with communication $\tilde{O}(k + 1/\epsilon^2)$, which breaks the deterministic lower bound when $k > 1/\epsilon$. However, the optimality of the sampling algorithm does not imply that it is optimal for these two problems; in fact it remains an open problem to determine the randomized communication complexity for these two problems.

If the elements in the streams are d-dimensional points, for example IP packets in the source-destination space, then a random sample of size $\tilde{O}(1/\epsilon^2)$ is an *ϵ-approximation* [31] with high probability. Such a sample allows us to approximately count the number of points in any range from a *range space* with bounded VC dimensions, such as rectangles, circles, halfspaces, etc. With our algorithm we can now track all these range-counts with communication $\tilde{O}(k + 1/\epsilon^2)$. If one only needs to determine if a range is large enough, that is, contains at least a ϵ-fraction of all points, then a random sample of size $\tilde{O}(1/\epsilon)$ suffices for this purpose, and is known as an *ϵ-net* [20]. Thus our algorithm tracks an ϵ-net with communication $\tilde{O}(k+1/\epsilon)$. There are numerous other applications of random samples which we will not enumerate here.

2 Binary Bernoulli Sampling

Binary Bernoulli sampling is a way of implementing Bernoulli random sampling which makes the analysis of the cost of the various sampling protocols more convenient. In its simplest form, the method associates each element in the input, e, with a (conceptually unbounded) binary string $b(e)$. The string $b(e)$ is chosen uniformly at random: each bit is independently set to 0 or 1 with probability $\frac{1}{2}$. From this, we can extract a Bernoulli sample of elements each chosen independently with probability $p = 2^{-j}$ for any (integer) j: we simply select all those elements whose binary strings have a prefix of 0^j (i.e., the first j bits are all 0).

A key feature of this method is that we do not need to materialize each bit string $b(e)$ immediately. Instead, it is often sufficient to materialize a prefix of $b(e)$ to determine whether an element in the input passes some initial filter. By the principle of deferred decisions, more bits of $b(e)$ can be generated later, to break ties or to accommodate a smaller p, when needed. In what follows, we treat $b(e)$ as if it is fully defined, with the understanding that if an algorithm accesses $b(e)[i]$ that is not yet fixed, it sets the value of $b(e)[i]$ as needed by generating a random bit.

One straightforward way of using this idea to maintain a random sample of size s without replacement is to keep the s elements with the (lexicographically) smallest $b(e)$'s. Implementing this idea over k distributed streams costs communication $O(ks \log n)$: the coordinator makes sure that all the sites know the global s-th smallest $b(e)$, say τ, and a site sends in a newly arrived element e iff its $b(e)$ is smaller than this threshold; every time τ changes, the coordinator broadcasts the new τ. Standard analysis shows that τ changes $O(s \log n)$ times, hence giving the claimed communication cost. It is also easy to show that the length of each string $|b(e)|$ that we need is $O(\log n)$ with high probability to break all ties, so it fits in $O(1)$ words. However, this simple way of using binary Bernoulli sampling is far from optimal. Below we present protocols that implement this idea in smarter ways so as to achieve optimal communication.

[1]The $\tilde{O}$ notation suppresses log factors.

3 Sampling over an Infinite Window

3.1 Sampling without replacement

We define the protocol ISWoR(s) (for Infinite window Sampling Without Replacement) as follows: The coordinator ensures that all sites are kept up to date with a current sampling probability p, which is a power of two. Initially, p is 1, and periodically the coordinator will broadcast to all sites to reduce p by half. We call the time while $p = 2^{-j}$ the jth round. On receiving an element e, a site tests the first j bits of $b(e)$, and reports this element to the coordinator if they are all zero.

The coordinator maintains a sampled set of size at least s wherein each element is selected with probability p (so this set is initialized with the first s elements from all streams). In fact, the coordinator maintains two subsamples in round j, denoted by T_j and T_{j+1}. On receiving a new element e sent by a site to add to the sample, the coordinator assigns the element to T_j if the $(j+1)$-th bit of $b(e)$ is 1, or to T_{j+1} if it is 0. Note that $T_j \cup T_{j+1}$ is a Bernoulli sample with sampling probability 2^{-j} while each of T_j and T_{j+1} is a Bernoulli sample with sampling probability $2^{-(j+1)}$.

The coordinator proceeds until $|T_{j+1}| = s$. At this time it sends out a broadcast message to halve p, and discards T_j. The coordinator then examines bit $j+2$ of $b(e)$ for each element e in T_{j+1} to determine whether it remains in T_{j+1} (if the bit is 1), or is "promoted" to T_{j+2} (if the bit is 0). Pseudo-code for the protocol as executed by each site and by the coordinator is shown in Algorithm 1 and 2 respectively.

Algorithm 1: ISWoR(s) for site in round j

```
foreach e do
    if the first j bits of b(e) are all zero then send e to Coordinator
```

Algorithm 2: ISWoR(s) for coordinator in round j

```
foreach e received do
    if b(e)[j+1] = 0 then
        T_{j+1} ← T_{j+1} ∪ {e}
    else T_j ← T_j ∪ {e} ;
    if |T_{j+1}| = s then
        foreach e ∈ T_{j+1} do
            if b(e)[j+2] = 0 then
                T_{j+2} ← T_{j+2} ∪ {e} ;
                T_{j+1} ← T_{j+1}\{e} ;
        discard T_j;
        j ← j+1 and signal all sites to advance to the next round;
```

At any moment in round j, a sample without replacement of size s can be derived from the active set of sampled elements via sub-sampling: we take $T_j \cup T_{j+1}$ (note that $|T_j \cup T_{j+1}| \geq s$) and sample s elements from this set without replacement. In fact, we can incrementally maintain the sample as new elements are added to $T_j \cup T_{j+1}$ using the reservoir sampling algorithm.

For the case $k \geq 2s$, we can use biased random bits to further reduce communication: for any element e, each bit of $b(e)$ is set to 0 with probability s/k and 1 otherwise. Note that in this case, $T_j \cup T_{j+1}$ is a

Bernoulli sample with sampling probability $(s/k)^j$; T_{j+1} is a sample with sampling probability $(s/k)^{j+1}$ while T_j is a sample with probability $(s/k)^j(1-s/k)$. The algorithm otherwise remains the same. The next theorem addresses the correctness (i.e. the sample is drawn uniformly) and the costs of the protocol. Correctness follows immediately, but the communication is bound by arguing that the number of rounds is highly unlikely to be much higher than the expected number.

Theorem 3.1 *The protocol* ISWoR(s) *continuously maintains a sample of size s drawn without replacement uniformly from all elements in $A(t)$. The amount of communication is $O(k \log_{k/s} n + s \log n)$; the coordinator needs $O(s)$ space and $O(k \log_{k/s} n + s \log n)$ time; each site needs $O(1)$ space and $O(1)$ time per element. These bounds hold with high probability.*

Proof: We first consider the case $k < 2s$; in this case the communication bound becomes $O(s \log n)$. The correctness of the protocol (that it draws a uniform sample without replacement) follows from the fact that $T_j \cup T_{j+1}$ is a Bernoulli sample with sampling probability 2^{-j}.

Now we analyze the various costs. For the j-th round, the amount of communication can be bounded by $O(s)$ with high probability, simply because each element sent to the coordinator is placed in T_{j+1} with probability $1/2$, therefore by the Chernoff bound, with probability at least $1 - e^{-s/4}$, $|T_{j+1}| \geq s$ after $4s$ elements are received in the coordinator side. The broadcast at the end of the round costs $O(k) = O(s)$ as well.

The total number of rounds as a function of the total number of elements in all streams, n, is bounded similarly. We show that with high probability, after n elements the protocol has reached round $\log(n/s)+1$. First, after the redistribution of elements at the end of round $j-1$ ($j \geq 1$), it is easy to see that $s \leq |T_{j+1}|$ and, by a Chernoff bound, at the beginning of round j, $|T_{j+1}| \leq 5s/8$ with probability at least $1 - e^{-\Omega(s)}$. Note that each element sent by sites in round $\log(n/s)+1$ is sampled with probability $2^{-\log(n/s)-1} = s/2n$, thus included into $T_{\log(n/s)+2}$ with probability $s/4n$, so with probability at least $1 - e^{-\Omega(s)}$, the number of samples being included into $T_{\log(n/s)+2}$ is no more than $3s/8$. Therefore the total number of elements in $T_{\log(n/s)+2}$ is no more than $5s/8 + 3s/8 = s$ before the protocol ends with high probability.

Thus the total communication cost is $O(s \log n)$ with high probability. The coordinator's time cost is asymptotically the same as the communication. The other bounds are immediate.

For the case $k \geq 2s$, the communication bound becomes $O(k \log_{k/s} n)$. The correctness of the protocol holds by noticing that $T_j \cup T_{j+1}$ is a Bernoulli sample with sampling probability $(s/k)^j$. We can bound the communication cost using similar arguments: Since each element sent to the coordinator is placed in T_{j+1} with probability s/k, after $O(k)$ elements, we will have $|T_{j+1}| \geq s$ with high probability. Similarly, the number of rounds is $O(\log_{k/s}(n/s))$ with high probability and in each round. Thus the total communication (as well as the coordinator's time cost) is $O(k \log_{k/s} n)$. The other bounds are mostly straightforward. One difference is that T_j will need to grow much larger before $|T_{j+1}|$ reaches s: in expectation, there must be k elements in T_j before there are s in T_{j+1}. However, we can still bound the space requirement to $O(s)$: by using reservoir sampling to maintain the sample extracted from $T_j \cup T_{j+1}$, T_j need not be materialized in full. □

The coordinator's space and the site's space/time bounds are clearly optimal. We next show that the communication cost is also optimal, by showing a lower bound on the amount of information which must be communicated. Note that the coordinator's time is at least proportional to its communication.

Theorem 3.2 *Any protocol that maintains a sample of size s without replacement over k distributed streams needs communication $\Omega(k \log_{k/s} n + s \log n)$ in expectation.*

Proof: The (expected) number of elements that will ever appear in the sample is $\Theta(s \log n)$. This is because the i-th element should have probability s/i of being sampled (recall that the algorithm has to maintain a uniform sample at each time step), therefore the expected total number of elements that will ever appear the sample is $\sum_{i=1}^{n} s/i = \Theta(s \log n)$. This also gives a lower bound on the communication, since these elements have to be sent to the coordinator.

Next we show that $\Omega(k \log_{k/s} n)$ is also a lower bound (when $k \geq 2s$). Suppose the total number of arrivals so far is m. Consider the next m/s arrivals seen by a site. If this site is the only one receiving elements since the first m elements, there is a constant probability that at least one of these m/s new elements appears in the sample, which should be communicated to the coordinator. If the site is not the only one receiving elements, this knowledge has to be communicated to the site. Either way some communication should occur. Then we can arrange the input so that each site in turn gets m/s arrivals. Therefore over the mk/s arrivals, $\Omega(k)$ messages should be exchanged in expectation. Then we update m and repeat this construction. This gives a lower bound of $\Omega(k \log_{k/s} n)$ over the whole stream. □

3.2 Sampling with replacement

There is a simple reduction from sampling with replacement to sampling without replacement. Suppose we have a sample S of size s without replacement from a population of size n. A sample S' with replacement can be obtained by the following procedure. For $j = 1, \ldots, s$, with probability j'/n we decide that the jth element of S' is a duplicate of an element already in the sample, where j' is the number of distinct elements in S' so far. In this case, we make the jth element of S' the same as an element chosen uniformly at random from the distinct elements of S'. Otherwise, with probability $1 - j'/n$, we decide that the jth element of S' is not a duplicate, and choose it to be the jth element of S. However, to perform this reduction in our setting requires the exact value of n, the population size, in order to pick elements with the right probability. As argued previously, this value is very expensive to track exactly in the distributed streaming model, so we will need separate algorithms for drawing a sample with replacement in our model.

Another simple solution to sampling with replacement is to run the ISWoR(1) protocol in parallel s times. Naively extrapolating the above bounds indicates that the cost is $\tilde{O}(ks)$. This is far from optimal, and can be improved as follows.

We define a protocol ISWoR(s) (Infinite window Sampling With Replacement) that uses the idea of "round sharing": effectively, it does run a modified version of the ISWoR(1) protocol in parallel s times; however, the jth round is terminated only when *every* instance has terminated the jth round. That is, in round $j \geq 0$ for each element e that arrives at a local site, the site generates s binary strings $b_1(e), \ldots, b_s(e)$. If any of these strings has a prefix of j 0's, then the element is forwarded to the coordinator, along with the index (or indices) of the successes.

The coordinator receives a sequence of elements, each tagged with some index i. For each index i, the coordinator retains a single element as $T[i]$, along with its current binary string $b[i]$. During round 0, the first time that an element arrives for a particular index i, the coordinator stores it as $T[i]$. Then for each element e received in round j for index i with binary string $b(e)$, the coordinator ensures that both strings $b(e)$ and $b[i]$ have enough bits generated so that $b(e) \neq b[i]$. The two strings are interpreted as integers: if $b[i] < b(e)$, then e is discarded; else, $b(e) < b[i]$, and e replaces $T[i]$, and $b[i]$ is overwritten with $b(e)$. The jth round terminates when the jth bit of $b[i]$, $b[i][j]$ is 0 for all i. At this point, the coordinator begins the $(j+1)$-th round by informing all sites to sample with $p = 2^{-j-1}$. At any moment, the coordinator can obtain a sample of size s with replacement by reporting $T[i]$ for all $i = 1, \ldots, s$.

Directly implementing the above algorithm for each site requires $O(s)$ time per element, simply because we have to generate s random binary strings for each element. We next describe how to reduce this time to

$O(1 + s/2^j)$ in round j, which becomes $O(1)$ for n sufficiently larger than s. The number of samples for which each element is selected with probability 2^{-j} is distributed as the binomial distribution $B(s, 2^{-j})$. Thus, we can generate a random number X from this distribution, and then select a set I of X indices uniformly from $[s]$. For each index $i \in I$, the j bit prefix of its binary string $b_i(e)$ is implicitly set to 0^j, and the pair (e, i) is sent to the coordinator. The resulting distribution of elements and indices sent is identical to that generated by directly generating s random binary strings for each element and sending those with a 0^j as a prefix. The pseudo-code for the site and coordinator is shown in Algorithm 3 and 4 respectively.

Algorithm 3: ISWR(s) for site in round j

```
foreach e do
    pick X from the binomial distribution B(s, 2^{-j}), and pick a set I of size X uniformly at random
    from [s] ;
    foreach i ∈ I do
        generate b_i(e) ;
        replace the first j bits of b_i(e) with 0^j ;
        send (e, i) to Coordinator.
```

Algorithm 4: ISWR(s) for coordinator in round j

```
foreach (e, i) received do
    if b_i(e) < b[i] then T[i] ← e, b[i] ← b(e);
    if ∀1 ≤ i ≤ s : b[i][j] = 0 then
        j ← j + 1;
        broadcast new j and go to the next round;
```

Finally, as for ISWoR(s), in the case $k \geq 2s$, we can further reduce the communication cost by using biased random bits. The next theorem demonstrates the corectness of this protocol, and analyzes the cost based on how long it takes to complete each round.

Theorem 3.3 *The protocol* ISWR(s) *continuously maintains a sample of size s with replacement drawn uniformly from $A(t)$. The communication cost is $O((k + s \log s) \log_{2+k/s} n)$; the coordinator needs $O(s)$ space and $O((k + s \log s) \log_{2+k/s} n)$ total time; each site needs $O(1)$ space and $O(1 + \frac{s}{n} \log s \log_{2+k/s} n)$ time per element amortized. These bounds hold with high probability.*

Proof: For uniformity of the sampling, consider just a single value of i and the corresponding $T[i]$. The protocol described carries out the Bernoulli binary sampling procedure to track a single sampled element. The effect is to select the element from the input whose $b(e)$, interpreted as an integer, is the least. Each element has an equal chance of attaining the least such string, since nothing specific to the element or the order in which it arrives influences this process. Therefore $T[i]$ is a uniform sample over the input.

To analyze the communication cost, observe that each round is essentially a coupon collector problem. That is, conditioned on an element being selected in round j to be sent to the coordinator, it is equally likely to have been selected for any of the s samples. A round must have terminated by the time we have "collected" at least one element for every i. So the round terminates after $O(s \log s)$ elements are received, with high probability. Here, we do not consider that the same element might be selected for multiple samples, or that

the previous round may have already provided elements with the required prefix to their binary string, since this only helps to reduce the cost.

For the number of rounds, we can use a variant of the analysis of the ISWoR(s) protocol to bound: the protocol will finish round j when there are $O(s \log s)$ events which each occur with probability 2^{-j}, out of ns trials (since each element is chosen with probability p with s independent repetitions). A variant of the previous analysis indicates that there is a polynomially small chance of reaching round $\log(ns/s \log s) + 2 = O(\log n)$. Therefore, the total communication cost is bounded by $O((k + s \log s) \log n)$, with high probability.

The amortized processing time per element can be broken into the time to determine how many copies of the element are sampled, plus the total number of sampled elements across all rounds spread across all elements. This is bounded (whp) by $O(1 + \frac{1}{n} s \log s \log n)$: the site has to perform $O(1)$ work for each of the $O(s \log s)$ sampled elements in each of the $O(\log n)$ rounds.

For the case $k \geq 2s$, the total number of rounds will be reduced to $\log_{k/s}(n/s)$. Similar analysis will give communication cost $O((k+s \log s) \log_{k/s} n)$ and processing time per element $O(1+\frac{s}{n} \log s \log_{k/s} n)$.

The time cost of the coordinator is constant per element received, and therefore bounded by the size of the communication, $O((k + s \log s) \log_{2+k/s} n)$. The space costs of each site and of the coordinator follow straightforwardly from the protocol description. □

4 Sampling from a Sequence-Based Sliding Window

In this and the next section, we consider sampling in a sliding window. We emphasize that the sliding window is defined on the union of the k streams, not on the individual streams.

We first consider how to sample from a sequence-based sliding window, that is, the sample is uniform from the last W elements received by the whole system. This model becomes particularly challenging in the distributed setting, due to the need to decide *when* an element in the sample expires (is no longer among the W most recent): its expiration is an implicit event defined by the arrivals of sufficiently many new elements. To this end, we make use of a "threshold monitoring" protocol which can determine, for a given r, when exactly r new elements have arrived. For completeness we present a simplified protocol to achieve this, based on a more general solution presented in [8].

Threshold protocol. The Threshold(r) protocol proceeds in $O(\log r)$ rounds. The protocol is initiated by a message from the coordinator to all k sites telling them to begin round 1. The coordinator should terminate the protocol when exactly r elements have been observed across the k sites. Each site maintains a counter of elements that have been observed but not reported to the coordinator. In round j, each site counts each arriving element. When the count reaches (or exceeds) $\lfloor r2^{-j}/k \rfloor$, the site announces this fact to the coordinator, and reduces its counter by $\lfloor r2^{-j}/k \rfloor$. Correspondingly, the coordinator increases its global counter by $\lfloor r2^{-j}/k \rfloor$. After the coordinator receives k such messages, it starts round $j+1$ by announcing this to all sites. (Note that a round change can trigger messages from sites whose counter c is in the range $\frac{r2^{-j}}{2k} \leq c < \frac{r2^{-j}}{k}$). The protocol reaches the final round when the global counter maintained at the coordinator is in the range of $[r - O(k), r]$. In the final round, each site simply sends a message to the coordinator at every arrival of the elements to increase the global counter by 1, until exactly r element arrivals have been counted. It is not difficult to see that each round requires $O(k)$ communication, so the total cost of this protocol is therefore $O(k \log r)$. The protocol correctly identifies exactly the moment when r elements have arrived across all sites since the protocol was initiated.

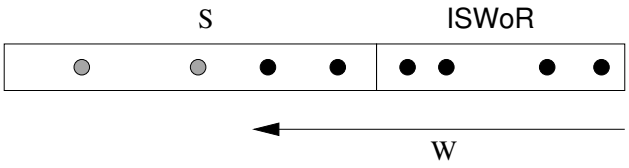

Figure 2: Schematic of SSWoR protocol for $s = 4$

4.1 Sliding Window Sampling without replacement

A simple solution to sampling from a sequence-based sliding window is periodic sampling, that is, whenever a sampled element expires, the next arriving element is sampled. This trivial predictability is not acceptable in many applications (see the discussions in e.g. [2, 4]), so usually we require that samples from disjoint windows must be independent.

Instead, we define a new protocol, SSWoR(s) (for Sequence-based window Sampling Without Replacement), which makes extensive use of the above Threshold protocol. The coordinator runs an instance of the protocol to demark every multiple of W arrivals: as soon as an instance of Threshold(W) terminates, a fresh instance is initiated. Within each such window of W, an instance of the ISWoR protocol is executed to draw a sample of size s. At the end of a window, the current sample S drawn by the ISWoR protocol is "frozen", and a new instance of the protocol is initiated. Assume for now that the coordinator can determine which elements in S have "expired" (i.e. fall outside of the window of W most recent elements). To draw a sample of size s without replacement, the coordinator extracts all elements in S that have not expired. It then uniformly samples without replacement from the sample provided by the instance of ISWoR on the current window to make up the shortfall. Note that all these elements are by definition unexpired.

Figure 2 shows a schematic view of the samples stored by the protocol for a sample size of $s = 4$. The frozen sample, S, is drawn from a window of W elements. Two elements (shown in gray) have expired, so a sample of non-expired elements is found by taking the two remaining samples (shown in black). Two more samples are taken from the 4 samples picked by ISWoR by subsampling uniformly. When the Threshold protocol indicates that the ISWoR instance has seen W total elements, all the elements in S will have expired, and the new sample is frozen, and forms the new S. In the next theorem, we show that the sample is drawn uniformly, by adapting an argument due to Braverman *et al.* [4]. The cost is bounded by studying the cost of the parallel invocations of the Threshold protocol.

Theorem 4.1 *The* SSWoR(s) *protocol continuously maintains a uniform sample of size s drawn without replacement from the W most recent elements. Samples from disjoint windows are independent. The communication cost is $O(ks \log W)$ per window; the coordinator needs $O(s)$ space and $O(ks \log W)$ time per window; each site needs $O(1)$ space and $O(1)$ time per element. These bounds hold with high probability.*

Proof: The correctness of the protocol, in that the resulting sample is drawn uniformly, follows from the results of Braverman *et al.* [4]. They show that given a uniform sample without replacement from a window with some expired elements, and a uniform sample of all subsequent (non-expired) elements, combining the two samples as described above for the SSWoR protocol results in a sample without replacement that is uniform from the non-expired elements only. Further, it is also shown that by this method the samples from disjoint windows are independent. So the bulk of our work is in analyzing the costs of our protocol.

First, observe that the communication cost of running the ISWoR(s) protocol is $O((k + s) \log W)$ per window, so tracking the size of window with Threshold(W) at a cost of $O(k \log W)$ is dominated by this. However, the bulk of the cost of the protocol arises from deciding when each sampled element expires. The

straightforward way of doing this is to initialize a separate Threshold(W) protocol whenever an element is sampled by ISWoR. An instance of the Threshold(W) protocol for an element e could be terminated prematurely if e is replaced by another element in ISWoR; else the protocol terminates normally at the time e expires. Since there are $O(s)$ running instances of the Threshold(W) protocol in the system, this requires $O(s)$ space at each site to maintain their state. Below we show how to reduce this space cost by running only one instance of the protocol at any given time.

For every element a_i sampled by ISWoR, we contact all k sites to compute its index i. Provided each site counts how many elements have arrived locally, the index i for an element is the sum of all these local counts when it is sampled. When a window freezes and S is produced, the sampled elements in S will start to expire. These s sampled elements are stored in order of their computed indices. Based on these indices, it is possible to compute how many more elements must arrive before each of them expires one by one. This can be achieved by running a Threshold(r) protocol for every sampled element with possibly a different r. Since the *order* of expiration is known in advance, it makes more sense to run these instances sequentially rather than in parallel. Thus the communication cost per window is $O(k\sum_{j=1}^{s}\log r_j) = O(ks\log\frac{W}{s})$ (and so is the coordinator's running time), which dominates the other communication costs. All the elements in S must expire before the next window is frozen, so the coordinator's space is $O(s)$ and each site's space is $O(1)$. Each site spends $O(W + s\log\frac{W}{s})$ time per window, which is $O(1 + \frac{s}{W}\log\frac{W}{s}) = O(1)$ per element. □

Although the $\tilde{O}(ks)$ communication cost may be much more than the infinite-window case, we show that this is actually the best that can be hoped for with sequence-based windows, by demonstrating a lower bound for any such protocol.

Theorem 4.2 *Any protocol that maintains a sample of size s without replacement over k distributed streams for a sequence-based sliding window of size W needs communication $\Omega(ks\log W)$ per window in expectation for sufficiently large W.*

Proof: Consider the process when the window slides from the first W elements to the next W elements. When the first window completes, the algorithm returns s sampled elements. We will argue that the algorithm has to know the precise time when each of these s elements expire. Suppose the algorithm only knows that a sampled element e expires in a time interval $[t_1, t_2]$. In order to not make a mistake by returning an expired element in the sample, it has to remove e from the sample with probability one before t_1. But if e's actual expiration time is t_2, then the probability that e is sampled within $[t_1, t_2]$ is zero, rendering the protocol incorrect.

With s sampled elements in a window of size W, with high probability there will be $\Omega(s)$ pairs of adjacent elements, each of which are at least $\Omega(W/s)$ elements apart. This becomes $\Omega(s)$ independent threshold problems with $r = \Omega(W/s)$. A lower bound in [8] shows that any randomized algorithm (with no errors) for the threshold problem with has to communicate $\Omega(k\log\frac{r}{k})$ messages in expectation. So the total communication cost is $\Omega(ks\log\frac{W}{ks})$, which is $\Omega(ks\log W)$ for sufficiently large W. □

4.2 Sliding Window Sampling with replacement

We define a SSWR(s) protocol (for Sequence-based window Sampling With Replacement). As in the infinite window case, the core idea is to run a protocol to sample a single element s times in parallel. Here, things are somewhat simpler than the ISWR case, because the need to track whether elements have expired dominates the other costs; as a result, we do not use the round-sharing approach since it does not generate

an asymptotic improvement in this case. Hence, the SSWR(s) protocol runs s instances of the SSWoR(1) protocol in parallel, which all share the same instance of Threshold(W) to determine when to freeze the current window and start a new one. The result is slightly simpler, in that each parallel instance of the protocol retains only a single element in S, and a single element from the current window, which replaces S when the element expires. It is straightforward to analyze the correctness and cost of this protocol given the above analysis, so we state without proof:

Theorem 4.3 *The* SSWR(s) *protocol continuously maintains a uniform sample of size s drawn with replacement from the W most recent elements. The communication cost is $O(ks\log(W/s))$ per window; the coordinator needs $O(s)$ space and $O(ks\log(W/s))$ time per window; each site needs $O(1)$ space and $O(1+\frac{s}{n}\log s\log n)$ time amortized per element. These bounds hold with high probability.*

5 Sampling from a Time-Based Sliding Window

The case of sampling from a time-based sliding window allows reduced communication bounds. This is because the coordinator can determine when an element expires from the window directly, based on the current time and the timestamp of the element. Therefore, it is not necessary to run any instances of the Threshold protocol, resulting in a much lower cost. This stands in contrast to the centralized case, where typically time-based windows are more costly to compute over. Nevertheless, the fact that sampling from time-based windows could be less costly does not mean the problem is easier. In fact the protocols here are more complicated than the previous ones in order to achieve near-optimal bounds. In the following, we use w to denote the duration of the sliding window in time, and n_t to denote the size of the window ending at time t, i.e., the number of elements with timestamps in $[t-w,t]$.

Below we first provide a relatively simple protocol for sampling without replacement over a time-based sliding window, which identifies the key challenges for this model. But this simple solution requires each site to retain a full history of elements that have arrived in the time window $[t-w,t]$. Subsequently, we give an improved protocol that removes this requirement.

5.1 A simple time-based protocol

The idea is to maintain a sample of size $s'\geq s$ over a partial stream starting from some "landmark" time T. At time t, as long as there are at least s active elements (namely, in the window $[t-w,t]$) among the sample of size s' that we are maintaining, then a sample of size s for the current sliding window can be obtained by sub-sampling from these elements. When there are fewer than s active elements left in the sample, we restart the protocol.

A good value for s', on one hand, should be large enough so as to minimize the number of restarts. On the other hand, it cannot be too large since otherwise maintaining a sample of size s' will be expensive. It turns out that setting $s'=c\cdot(s+\log n_T)$ (for some constant c) strikes the right balance.

The full-space protocol. In this protocol each site retains all active elements. We first run the ISWoR(s) protocol until $t=w$. The protocol at the coordinator side then does the following.

1. Set $T=t$, and compute n_T by contacting all sites. Restart the ISWoR(s') protocol to maintain a sample of size s' from time $T-w$. This is possible since each site retains all active elements.

2. At time t,

(a) If there are fewer than s' elements in the sample, then the coordinator must have actually collected all the elements in the window $[T-w, t]$. From these, we can subsample s elements from the active ones (or just report all active elements if there are fewer than s in total).

(b) Else, we check if there are at least s active elements in the sample. If so, we subsample from these elements to pick a sample of size s. Otherwise we go back to step 1.

The correctness of the protocol is straightforward: given two partial streams D_1, D_2 with $D_2 \subseteq D_1$, if S is a uniform random sample in D_1, then $S \cap D_2$ is a random sample of D_2.

To bound the communication cost we argue that the protocol restarts $O(\log W)$ times in a window with W elements. Consider the first time T in this window when we restart. Recall that we maintain a random sample of size s' for the window $[T-w, t]$. As long as half of the elements in this window have timestamps in the range $[t-w, t]$, then with high probability we will have at least s unexpired elements from the s' sampled elements for some constant c large enough, by a Chernoff bound. Therefore with high probability, the first restart happens when $n_1 \geq W/2$ elements arriving after time $T - w$ expire, causing T to reset. Similarly, we can show that with high probability, the second restart happens when another $n_2 \geq (W - n_1)/2 \geq W/4$ elements arriving after time $T - w$ expire (note that the number of elements arriving after time $T - w$ is at least $W - n_1$), and so on. Therefore, with high probability, the total number of rounds will be $O(\log W)$. So the total communication is $O((s + k) \log^2 W)$ with high probability.

However, the simple protocol above needs each site to keep all the active elements due to the restarts. Below we show how to avoid explicit restarts by computing sufficient information so that there are always enough elements present at different sampling rates to draw a sample. At the same time, we reduce the space cost to $\tilde{O}(s)$ at each site, which will be shown to be near-optimal. The new protocol below also improves the communication by an $O(\log W)$ factor.

5.2 Time-based Sampling without replacement

The key challenge in the time-based sliding window setting arises because the element arrivals may not occur uniformly over time. In particular, later arrivals may be much less frequent than earlier ones. As a result, any current sample may be dominated by earlier elements. When these elements expire, there may be insufficiently many later elements sampled to provide a sample of size s. In the above protocol, this necessitates the "restart" step, to revisit the later elements and redraw a sufficiently large sample of them. In this section, we provide a protocol which allows the coordinator to draw a sample of sufficiently many elements from the history without having to resample from past elements.

We define the TSWoR(s) (Time-based window Sampling Without Replacement) protocol based on the ISWoR protocol. To keep track of the current window, an instance of the ISWoR(s) protocol is run. This is terminated after w time units, and a fresh instance is started—we refer to each such period of w as an "epoch", and we denote the start point of the most recent epoch as T. In addition, each site maintains a sample of recent elements at various rates of sampling. These are kept private to the site until the end of an epoch, at which point the coordinator collects certain information from the sites about their current samples.

Level sampling at sites. For each element e that is observed at a site, the site assigns it to several "levels" based on $b(e)$: if $b(e)$ has a prefix of l zeroes, it is assigned to all levels $1, \ldots, l + 1$. Note that an element is assigned to level l with probability 2^{-l+1}. The site then retains a queue for each level l consisting of the most recent elements assigned to this level until either (i) at least s are also assigned to level $l + 1$, or (ii) all active elements at that level are retained. We call such a structure a *level-sampling (LS)* structure. The process to maintain an LS structure is shown in Algorithm 5.

Algorithm 5: Update Level-Sampling

```
foreach e do
    l ← 0 ;
    repeat
        l ← l + 1 ;
        insert e in queue l ;
        while queue l has > s elements from levels > l do
            delete oldest element from queue l;
    until b(e)[l] = 1 ;
```

An example is shown in Figure 3 with $s = 3$. In the figure, circles represent elements that have been sampled at particular levels, where a hollow circle is sampled at level l and a filled circle is an element that was also sampled at a higher level. Level 5 has a single element that was sampled with probability $p = 1/32$, while level 4 contains three elements, two of which were sampled at level 4, and the one that was sampled at level 5 (so condition (ii) holds). Level 2 retains only the most recent elements so that there are $s = 3$ included which were sampled at levels 3 and above (the three filled circles at level 2), meeting condition (i). The same is true at level 1.

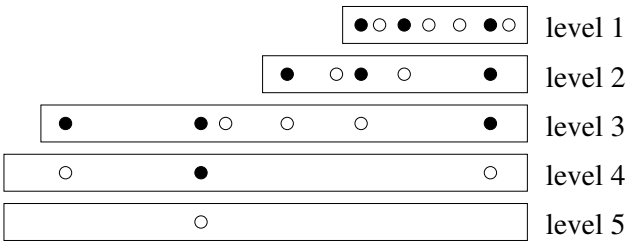

Figure 3: Example level-sampling data structure for $s = 3$

Collection of sampled elements. At the end of each epoch, the coordinator aims to collect an LS structure equivalent to the one resulting if one site had seen the union of all elements. This is done most efficiently as a k-way merge: starting at the greatest level with any elements sampled, each site sends the most recent sampled element at level l. The coordinator determines which of these is the most recent globally, adds this to its queue at level l, and prompts the corresponding site for the next most recent sampled element at level l. The current level concludes when the coordinator has either obtained all unexpired elements from all sites at that level, or until at least s elements have been collected that also belong to level $l + 1$.

Production of a sample. At any time t, the coordinator can produce a uniform random sample of size s from the current window of duration w. The coordinator considers the LS structure of the most recent complete epoch, and identifies the level l which covers the time interval $[t-w, T]$ and still has at least s non-expired sampled elements: this is guaranteed to exist by definition of the procedure (except in the extreme case when there are fewer than s non-expired elements from that epoch, in which case all these elements are retained). It also takes the current set of elements from the instance of ISWoR which is operating in round j. Then we have a set of elements A from the current epoch $[T, t]$ (Bernoulli) sampled with probability 2^{-j},

and a set of elements B from $[t-w,T]$ sampled with probability 2^{-l+1}. If $j = l-1 = 0$, then it means that we have actually collected all elements in the window $[t-w,t]$ and sampling will be trivial. Otherwise at least one of A and B has at least s elements. Letting $\ell = \max(j, l-1)$, the coordinator selects all those elements whose $b(e)$ has a prefix of ℓ 0's, so these represent a Bernoulli sample with sampling probability $2^{-\ell}$. This results in the set C of at least s elements, from which uniformly selecting s elements gives the final sample.

We next have to show the correctness of this protocol, and analyze its costs. For correctness, we show that we can combine the information from fixed windows to obtain a sample of sufficient size, such that every element in the sample is drawn from the time range $[t-w,t]$, and each element in this range has the same probability of entering the sample. We then bound the number of elements collected in the merging step to bound the communication and space of the protocol.

Theorem 5.1 *The protocol* TSWoR(s) *maintains a random sample of size s without replacement from all elements with timestamps in the range $[t-w,t]$. The communication cost is $O((k+s)\log W)$ per window (where W is the number of elements within it); the coordinator needs $O(s\log W)$ space and $O((k+s)\log W)$ time per window; each site needs $O(s\log W)$ space and $O(1)$ time per element. These bounds hold with high probability.*

Proof: For the correctness, we claim that the result of the sampling process is to draw a Bernoulli sample C of size at least $\min(s, n_t)$ so that each element in C is from the range $[t-w,t]$, and every element in this range is selected into C with equal probability. Having established this, the fact that the resulting sample is a uniform sample without replacement of size $\min(s, n_t)$ follows easily.

To see this uniformity, consider the two epochs with unexpired elements. First, each element in the current epoch (by definition, unexpired) is Bernoulli sampled by the ISWoR(s) protocol running round j with probability 2^{-j}. Meanwhile, each (unexpired) element in the previous epoch that is retained at level l is also the result of Bernoulli sampling with sampling probability 2^{-l+1}, irrespective of which site it was observed at. The coordinator picks the level l where at least s non-expired elements are retained. Such a level is guaranteed to exist based on the definition of the LS structure: consider a level l whose earliest element is expired, and where the earliest element retained at level $l-1$ is not expired. By the requirements on $l-1$, it must contain at least s elements which are present at level l, and these are unexpired. So there are at least s unexpired elements stored at level l. The subsampling procedure then reduces the probability of sampling of whichever set (A or B) is at the higher probability, so the result set C is drawn with the same probability $2^{-\ell}$. So the probability of any unexpired element from any site to reach C is the same, $2^{-\ell}$.

For the communication cost, we have to analyze the number of elements at each level in the LS structure stored by the coordinator. If the coordinator collects s' elements from sites to fill level l, the communication cost is $O(k+s')$ to do the k-way merge. We argue that the total number of elements collected at level l is bounded with high probability. The elements stored in the LS structure at level l all have 0^l as a prefix of $b(e)$. The elements that are sampled to level l have $b(e)[l+1] = 1$, whereas the elements also sampled to higher levels have $b(e)[l+1] = 0$. So the probability that an element is sampled to level l, conditioned on the fact that it is sampled to level l or higher is $1/2$. Hence the number of elements at level l is bounded by $O(s)$, with high probability. Likewise, over the W elements in the epoch $[T-w,T]$, with high probability the highest level reached is $O(\log W)$. Therefore the communication cost of collecting the sampled elements to the coordinator at time T is $O((k+s)\log W)$, which is also the coordinator's running time in this epoch.

By the same argument, the amount of space required by each site to store its LS data structure is also bounded by $O((k+s)\log W)$. Processing each element at a site requires determining which level it belongs

at in the LS structure, and possibly pruning some old elements, as well as running the ISWoR protocol on it. The total time spent is amortized to a constant per element. □

Optimality. Since sampling from time-based window is more general than the infinite-window case, so the communication lower bound of Theorem 3.2 also applies here with $n = W$, namely, $\Omega(k \log_{k/s} W + s \log W)$. Note that when $k = O(s)$, the upper and lower bounds match; when $k = \omega(s)$, there is a gap of $O(\log(k/s))$ between the upper and lower bounds. In fact, if we use biased random bits as in the ISWoR protocol for the case $k \geq 2s$, we could achieve the optimal $O(k \log_{k/s} W)$ communication bound, too. However, this will increase the space of each site to $O(k \log_{k/s} W)$.

A space bound of $\Omega(s \log W)$ follows from [14] since our model is more general than the centralized setting. However it is unclear if all the participants (coordinator and each site) need to pay this much space; all our previous protocols only needed $O(1)$ space on each remote site while only the coordinator needs $\tilde{O}(s)$ space. Below we argue that this is not possible for time-based windows by providing a lower bound on the communication.

Theorem 5.2 *Any protocol that maintains a sample of size s over k distributed streams for a time-based sliding window requires that each of at least $k/2$ sites must store at least $s/2$ elements, unless the protocol incurs a communication cost of $\Omega(W)$.*

Proof: We generate inputs as follows. We first send s elements to site S_1. By the sampling requirement S_1 needs to send all of them to the coordinator as sampled elements. Then we send the next s elements to S_2. Assume S_2 does not have space to store $s/2$ elements. If S_2 sends fewer than $s/2$ elements to the coordinator, then some of the s elements must have been discarded. Then we stop generating further elements until these s elements received by S_2 are the only active ones in the sliding window. This will cause a failure in the sampling protocol. Otherwise we continue sending s elements to $S_3, S_4, \ldots, S_k$, one by one. By the same argument, for any site that does not have $s/2$ space, it has to send at least $s/2$ elements to the coordinator. If there are more than $k/2$ such sites, then a constant fraction of the elements have been transmitted. Finally we can use the same construction in a round-robin fashion to cause $\Omega(W)$ communication in each window. □

5.3 Time-based Sampling with replacement

A naive solution to drawing a sample with replacement for a time-based sliding window is to execute the TSWoR(1) protocol s times in parallel. This is certainly correct, but would be costly, requiring $\tilde{O}(ks)$ communication. Instead, we show below how to achieve (almost) the same result by more careful use of communication.

The protocol. We use again the notion of epochs to define the TSWR(s) (Time-based window Sampling With Replacement) protocol. For the current epoch, the ISWR(s) protocol is used to maintain a sample at the coordinator of active elements in the range $[T, t]$. For the previous epoch, each site maintains s independent copies of the LS structure. For $i = 1, \ldots, s$, each element has a random bit string $b_i(e)$, which determines the ith LS structure: e is sampled to level l if the first $l - 1$ bits of $b_i(e)$ are all zero, and the queue at level l keeps the most recent sampled elements until (i) it sees an element also sampled to level $l + 1$, or (ii) no more active elements are sampled to this level.

At the end of the epoch, the coordinator needs to build the s LS structures on the union of the streams. Simply building each of these in turn would cost $\tilde{O}(ks)$ communication. Instead, the coordinator merges

all of them in parallel: for each level l, it first obtains the most recent element from each site across all s instances of the LS structure. It then does a k-way merge to find the second, third, ..., most recent elements on level l across all s instances, until it has obtained the necessary samples for each of the s instances, i.e., until condition (i) or (ii) holds on the union of the streams for every LS instance.

To form the ith sample at time t, the coordinator extracts all samples from the appropriate level of the merged LS structure as B_i, at level l (note that B_i may contain more than one sampled element). It also extracts the ith sample from the ISWR(s) protocol from round j, along with its associated binary string $b[i]$, as A. For each element in B_i, we also have its binary string $b_i(e)$. We then interpret the binary strings as integers, and pick the element e with the smallest $b_i(e)$ as the ith sampled element (ties are broken by examining longer prefixes of the bit strings and drawing more random bits as needed). The correctness is shown in the next theorem based on the correctness of protocols in the infinite window case. The communication cost, and hence the running time, is bounded by arguing that the number of elements needed in total from all sites is tightly bounded.

Theorem 5.3 *The protocol* TSWR(s) *draws a uniform sample of size s with replacement from all elements with timestamps in the range $[t-w, t]$. The communication cost is $O((k + s \log s) \log W)$, and so is the coordinator's running time. Each site needs $O(s \log W)$ space and $O(s)$ time per element. These bounds hold with high probability.*

Proof: For the communication cost, since the cost of running ISWR(s) is $O((k + s \log s) \log W)$, we focus on the merging procedure. For each level l, we consider all those elements that were kept by any site on level l by any of the s instances. Walking backwards from T, we encounter each of these elements in turn, and the coordinator can stop collecting elements after it has received one element for each i at level $l + 1$. Each element collected is equally likely to be for any of the s instances, and has probability $\frac{1}{2}$ to appear at level $l + 1$. So, by appealing to the coupon collector's problem, the coordinator only needs to collect $O(s \log s)$ elements until the level can be terminated, with high probability. Thus the communication cost of the merging for each level is $O(k + s \log s)$. As there are $O(\log W)$ levels with high probability, the communication bound follows. Since each instance of the LS structure needs $O(\log W)$ space and $O(1)$ time per element to maintain, the site's space and time bounds are $O(s \log W)$ and $O(s)$, respectively.

To see that the samples are drawn uniformly, we adapt the argument of Theorem 3.3. For a particular value of i, the element drawn as the sample is the e in the time range $[t-w, t]$ with the lowest $b_i(e)$. By interrogating the stored LS structure, the coordinator recovers a set of elements with 0^l as a prefix of $b_i(e)$, and guarantees that there are no other unexpired elements from that epoch with the same prefix. From the instance of ISWR(s), the coordinator recovers e with 0^j as a prefix of $b_i(e)$, and guarantees that there is no other e from the same epoch with a lower $b_i(e)$. The remainder of the process combines these two sets of elements to find the element from $[t-w, t]$ with the smallest $b_i(e)$ is recovered as the ith sample. □

5.4 Processing time for Time-based Sampling with replacement

The per-element processing time at each site in the above TSWR(s) protocol is $O(s)$ because we have to maintain s LS structures. This can be quite expensive when s is large. In this section we show how it can be reduced to $O(1)$ for W sufficiently larger than s. The idea is to delay detailed sampling at most LS instances, and only fully materialize the LS structure at the end of the epoch. By this point, many elements which would potentially have been sampled to low levels can be pruned away, so the space usage remains bounded. More precisely, for each arriving element e, we first decide the greatest level $l_{\max}(e)$ it reaches across the s LS instances, and store it at all levels $1, \ldots, l_{\max}(e)$ in just one LS instance chosen uniformly at random from

the s instances. Note that $l_{\max}(e)$ follows the distribution $\Pr[l_{\max}(e) = j] = (1 - 2^{-j-1})^s - (1 - 2^{-j})^s$. In the full sampling case, this element is also sampled to various levels in the other $s - 1$ LS instances. We will postpone this sampling to the end of the epoch. Meanwhile, based on the arrival of other elements, we can also determine when this element expires (whp) from all other LS instances, and thus prune it entirely without ever having materialized its presence in these instances. This will be sufficient to reduce the time costs.

At the end of the epoch, for each *original element* e that is still kept in the LS instance where it was first inserted when it arrived (we describe later how elements are kept), we insert copies of it to the other $s - 1$ LS structures. We call these the *delayed copies* of e. For each of these LS structures, we decide its level via

$$\Pr[l(e) = l | l_{\max}(e)] = 2^{-l}/(1 - 2^{-l_{\max}(e)}) = 2^{l_{\max}(e)-l}/(2^{l_{\max}(e)} - 1),$$

which fixes the prefix of its binary string. After inserting a delayed copy into an LS structure, old elements are removed based on conditions (i) and (ii) defined above in Section 5.3.

In order to bound the space used, we prune elements as time goes on. An element e can be safely pruned when none of the final configurations of the s LS instances at the end of the window could possibly contain e's delayed copies. Recall that the level a delayed copy of e reaches in an LS instance cannot exceed $l_{\max}(e)$, so if this LS instance already contains an original element e' with $l_{\max}(e') > l_{\max}(e)$ and a later timestamp, then e (or its delayed copy) would have been removed in this LS instance at the end of the epoch. Thus e can be pruned when every LS instance contains such an e'. Checking each of the s LS instances one by one would be slow. Instead we keep all the original elements in one queue, each tagged with its $l_{\max}(e)$ and the index of the LS instance where it belongs to as an original element. After every $\lfloor s \log s \log W \rfloor$ arrivals we perform a batched pruning as follows. We scan the queue starting from the most recent element. For each level l, we keep an array of s bits, initialized to all zeros. When we see an element e from the ith LS instances with $l_{\max}(e) > l$, we set the ith bit to 1. When all s bits are 1 we can prune all elements e with $l_{\max}(e) = l$ with older timestamps. The next theorem analyzes the time and space costs of this modified process.

Theorem 5.4 *In the modified protocol for* TSWR(s)*, each site needs* $O(1 + \frac{s^2}{W} \log s \log W)$ *time per element amortized. The space at each site is* $O(s \log s \log W)$*. The other bounds are the same as in Theorem 5.3.*

Proof: We first bound the space used at each site, showing that with high probability, after each batched pruning, only $O(s \log s \log W)$ elements are left. Since as argued before there are $O(\log W)$ levels with high probability, it suffices to argue that for each l, there are $O(s \log s)$ elements with $l_{\max}(e) = l$ across all s LS instances.

When we perform the pruning, a level l will not accommodate more elements with $l_{\max}(e) = l$ when we see an element from each of the s LS instances with $l_{\max}(e) > l$. By the coupon collector bound, with high probability this happens after seeing $O(s \log s)$ elements at level l. One can verify that $\Pr[l_{\max}(e) > l] \geq \frac{1}{2} \Pr[l_{\max}(e) = l]$, by considering the function

$$f(s) = \Pr[l_{\max}(e) > l] - \frac{1}{2} \Pr[l_{\max}(e) = l] = (1 - (1 - 2^{-j-1})^s) - \frac{1}{2}((1 - 2^{-j-1})^s - (1 - 2^{-j})^s),$$

and seeing that its derivative $f'(s) > 0$ so $f(s) \geq f(1) > 0$. Thus when we see $O(s \log s)$ elements with $l_{\max}(e) > l$ with high probability there are at most $O(s \log s)$ elements with $l_{\max}(e) = l$, by a Chernoff bound. This shows that there are $O(s \log s)$ elements with $l_{\max}(e) = l$, hence a total of $O(s \log s \log W)$.

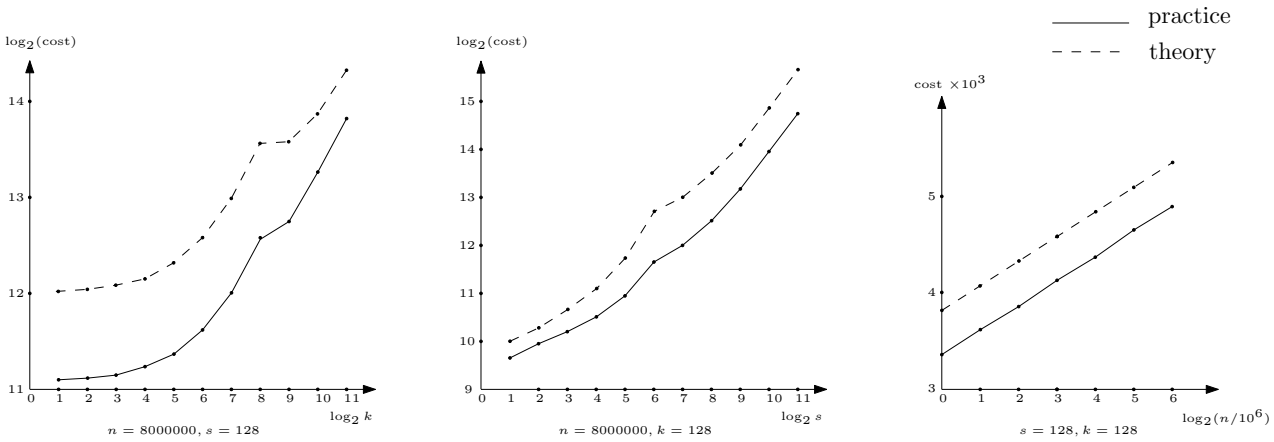

Figure 4: ISWoR on synthetic data

This is the number of elements after each pruning. Since the pruning is performed after every $O(s \log s \log W)$ elements, the total space required is bounded by $O(s \log s \log W)$.

Next we bound the amortized processing time per element. During a batched pruning, recall that we keep an array of s bits for each l and declares the level "full" when all s bits are one. By the coupon collector problem we need to see $O(s \log s)$ elements before this happens. After a level is full we no longer need to update this array, except pruning elements with $l_{\max}(e) = l$. Thus the total time spent for a batched pruning is $O(s \log s \log W)$, namely $O(1)$ per element amortized. At the end of the epoch, the cost of sampling from all the $O(s \log s \log W)$ remaining delayed elements is $O(s^2 \log s \log W)$ (whp). Combining the ISWR part, the amortized processing time per element is $O(1 + \frac{s^2}{W} \log s \log W)$. □

6 Experimental Study

To better understand the real-world costs of the protocols we have described, we perform a brief experimental study. The goal of this study is to compare the empirical behavior to the worst case behavior from the analysis, and to give some intuition into the absolute values of the costs for different input parameters. We focus on the ISWoR and TSWoR protocols, and study their communication cost using both synthetic and real data sets. We postpone study of sequence-based algorithms, due to their increased cost, and because they are arguably less intuitive than time-based windows. We highlight sampling without replacement, since this represents the core problem (sampling with replacement is based on sampling without replacement as a special case).

To compare the empirical behavior to the theoretical bounds, we also plot the (asymptotic) worst-case function. Note that our analysis does not expose the exact constants of proportionality inside the big-Oh notation: instead, we pick some small constants (will be specified for each figure) for illustration purpose. Our results show that with this setting, the two curves are very close not just in shape, but also in absolute value.

Experimental environment. We implemented a simulator which reads the streams for each site, and computes the messages sent to and from the coordinator. We used a mixture of real and synthetic data. The real data set is obtained from the 1998 World Cup web server request traces, available at the Internet Traffic

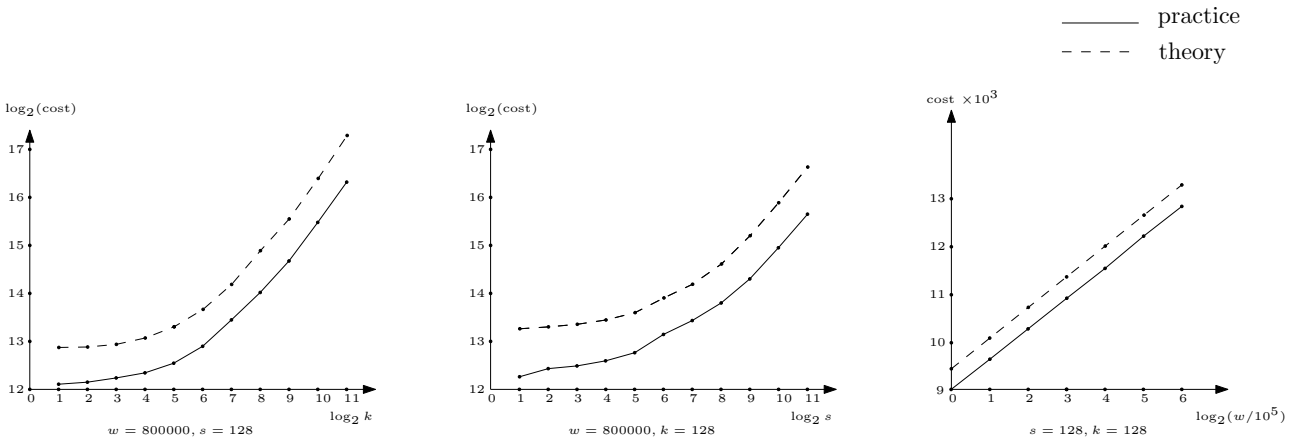

Figure 5: TSWoR on synthetic data

Archive. We used the part 1 trace from day 46, which contains 7×10^6 records. Each record is associated with a timestamp and a "server id", which we used to designate the site receiving the record.

To generate synthetic data, we allocated data elements uniformly to one of k sites, and picked the interval between each successive element uniformly in the range 1 to 8 time units (note that the sampling protocols are indifferent to other data about the elements in the streams). In all cases, we measure the communication cost in terms of the number of messages sent; note that all protocols use messages of constant size.

Experimental Results. In Figure 4 we report the communication cost for the protocol ISWoR on synthetic data sets with different parameter settings. For the leftmost plot, we fixed the total number of elements at $n = 8 \times 10^6$, and a sample size of $s = 128$, while varied the number of sites k from 2^1 to 2^{11}. The y-axis shows the communication cost as the base-2 logarithm of the number of messages exchanged during the whole protocol. Recall that our analysis of the ISWoR protocol sets the cost as $O(k \log_{k/s} n + s \log n)$. When $k < s$, this cost is $O(s \log n)$: there are $O(\log n)$ rounds, and each round terminates with a message to all k sites when the coordinator has received $O(s)$ elements. Therefore, the dependence on k is relatively weak in this range. But when $k > s$, the dominating term becomes $O(k \log_{k/s} n)$, and indeed we see an increase in cost which is sublinear in k (as k doubles, the cost less than doubles). The overall cost is small compared to the data size: even with $k = 2^{11} \approx 2000$ sites, the total number of messages is only ≈ 16000.

We use the more accurate formula $2 \times (k \log_{\max\{k/s,2\}}(n/s) + s \log(n/s))$ to plot the corresponding theoretical curve. For illustration purpose we simply set the multiplicative constant hidden inside the big-O notation to be 2. It is striking how well these two curves are matched, both in shape, but also in absolute value. The both exhibit the same kink, which occurs when $k/s = 2$, i.e. when $k = 2^8$, as the base of the logarithm switches, and the remainder of the growth is dominated by the second term in the formula. The same formula is used to plot the middle curve in Figure 4. For the rightmost curve we use $(k \log_{\max\{k/s,2\}}(n/s) + s \log(n/s) + 500)$ in order to fit it into the figure better. Note that in the the rightmost plot the y-axis is plotted on a linear, rather than logarithmic, scale.

In the middle plot we fixed $n = 8 \times 10^6$ and $k = 128$ while the sample size s was varied from 2^1 to 2^{11}. The parameter s has the most effect on the cost when $s > k$, and the trend approaches linear increase in s as s grows.

In the rightmost plot of Figure 4, we fixed $k = s = 128$ and varied n from 10^6 to 64×10^6. The communication cost is observed to increase linearly with $\log n$, and requires only 5,000 messages to contin-

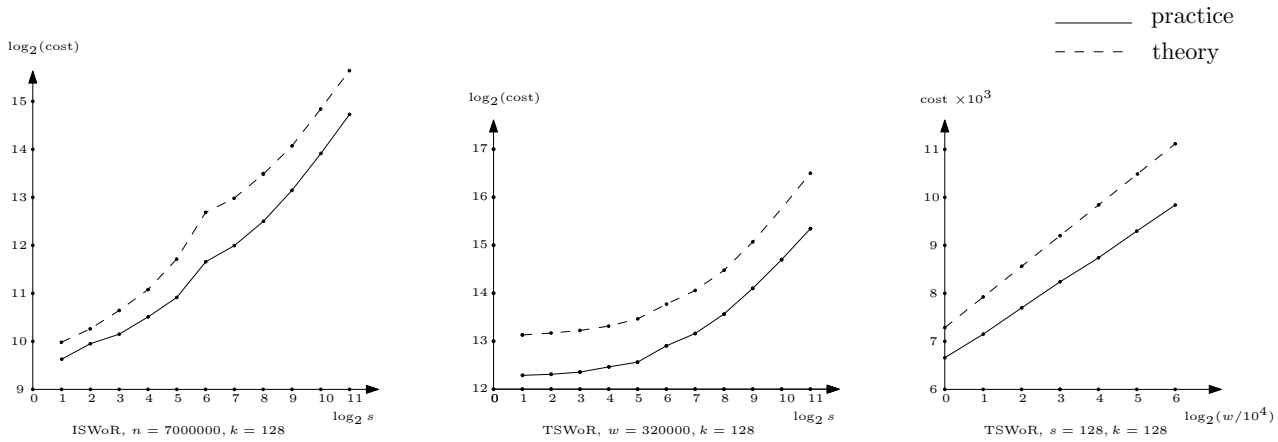

Figure 6: Experiments with ISWoR and TSWoR on Real data

uously maintain a sample of 128 elements across 128 sites (compared to the 64×10^6 messages required to centralize the data). This demonstrates that the protocol scales well to large quantities of data.

In Figure 5 we report the experiment results of the TSWoR protocol on the synthetic data sets, while varying k, s, and the window length w, respectively. The general trends are all very similar to the ISWoR protocol. There is close to linear growth in k, for k larger than s, and linear growth in s, for s larger than k. The dependence on $\log W$ (the logarithm of the actual number of elements arriving in each window; in our experiments $W \approx w/4.5 \approx 178000$) is also linear. The main difference is that the costs are all a constant factor higher than in the infinite window case. This is due to the cost of performing the k-way merges to build the level-sampling structures at the end of each epoch.

The corresponding theoretical curves in the leftmost and the middle plots are plotted by the more accurate formula $2 \times (k \log_{\max\{k/s,2\}}(n/s) + s\log(n/s)) + 2 \times (2k+s)\log W$. Again we set the constant hidden inside the big-O notation to be 2. The first term is the cost of the run of ISWoR and the second term is the merge cost of k level-sampling data structures. Recall that in the level-sampling data structure we do not use biased random bits as in ISWoR since otherwise the space cost of each site will increase. For the rightmost plot we use the formula $(k \log_{\max\{k/s,2\}}(n/s) + s\log(n/s)) + (2k+s)\log W + 2000$. That is, we replace the multiplicative constant 2 with an additive constant 2000, for the same reason as stated before.

Figure 6 shows the results of our experiments on the real data set derived from the 1998 world cup logs. These plots show similar dependencies as for the synthetic case: (sub)linear growth in communication cost as a function of s for both the infinite and time-based window cases (leftmost and middle plots), and logarithmic dependence on the data size (rightmost plot). We argue that this is because the protocols depend very weakly if at all on the instance of the data, and can be well characterized in terms of the coarse parameters of the instance (n, k, s and the actual number of elements handled by the protocol). The behavior in practice again matched that predicted by the analysis. One may notice that the shapes of the theoretical curve and the practical curve differ a bit in the rightmost plot. This is because in the real data, a majority of data elements were sent to only a few sites, while in the theoretical curves we still assume that data elements were allocated uniformly to all sites, which is the worst case in terms of communication cost.

Lastly, we observe that the implemented protocols are very time efficient. For example, our simulator only takes a few seconds to process the World Cup data set (with 7 million records) on a commodity desktop PC, while simulating the actions of all the sites as well as the coordinator.

7 Conclusion and Open Problems

In this paper we have generalized classical reservoir sampling algorithms to the case where we want to continuously maintain a random sample over multiple distributed streams. To minimize communication, we needed new techniques and analysis since those for a single stream rely on information (the current total number of elements) that is inherently hard to maintain in the distributed setting.

At the end of Section 1, we mentioned a number of applications of random sampling. Random sampling indeed solves these problems, but it is unclear if it always gives the best solution. On a single stream, better algorithms are known that either do not use random sampling at all (e.g., the heavy hitter [25] and quantile problem [19]), or use some more sophisticated sampling algorithms (e.g., ϵ-approximations in bounded VC dimensions [5]). While some of these problems have been studied in this distributed streaming setting, only their deterministic complexity has been understood [33]. It remains open to see how randomization can help reduce communication for these problems in this model.

Finally, our results are all worst-case, in that they assume no prior knowledge about the likely arrival distribution of items. In some cases, a prior distribution is known. It seems plausible that this could be used to give tighter bounds, provided that the true arrival distribution is close to the prior distribution. Such problems have been dubbed "Stochastic Streaming", but few results are known in this setting [27].

References

[1] C. Arackaparambil, J. Brody, and A. Chakrabarti. Functional monitoring without monotonicity. In *ICALP*, 2009.

[2] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, 2002.

[3] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, 2003.

[4] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling for sliding windows. In *PODS*, 2009.

[5] B. Chazelle. *The Discrepancy Method*. Cambridge University Press, 2000.

[6] G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *VLDB*, 2005.

[7] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *SIGMOD*, 2005.

[8] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. In *SODA*, 2008.

[9] G. Cormode, S. Muthukrishnan, and W. Zhuang. What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In *ICDE*, pages 20–31, 2006.

[10] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of ACM SIGCOMM*, 2003.

[11] N. Duffield, C. Lund, and M. Thorup. Priority sampling for estimation of arbitrary subset sums. *J. ACM*, 54(6), 2007.

[12] P. S. Efraimidis and P. G. Spirakis Weighted Random Sampling with a Reservoir. *Information Processing Letters*, 97:181–185, 2006.

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

[14] R. Gemulla and W. Lehner. Sampling time-based sliding windows in bounded space. In *SIGMOD*, pages 379–392, 2008.

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

[16] R. Gemulla, W. Lehner, and P. J. Haas. Maintaining bounded-size sample synopses of evolving datasets. *VLDB J.*, 17(2):173–202, 2008.

[17] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *International Conference on Very Large Data Bases*, 2001.

[18] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, pages 331–342, 1998.

[19] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, 2001.

[20] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete and Computational Geometry*, 2:127–151, 1987.

[21] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, A. D. Joseph, M. Jordan, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *IEEE INFOCOM*, 2007.

[22] R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD*, 2006.

[23] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[24] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, 2005.

[25] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems*, 2006.

[26] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. NOW publishers, 2005.

[27] S. Muthukrishnan. Stochastic Data Streams. In *MFCS*, 2009.

[28] F. Olken. *Random Sampling from Databases*. PhD thesis, Berkeley, 1997.

[29] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distribtuted data streams. In *SIGMOD*, 2006.

[30] I. Sharfman, A. Schuster, and D. Keren. Shape sensitive geometric monitoring. In *PODS*, 2008.

[31] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[32] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, Mar. 1985.

[33] K. Yi and Q. Zhang. Optimal tracking of distributed heavy hitters and quantiles. In *PODS*, 2009.