6-10-2019

# Parallel Streaming Random Sampling

Kanat Tangwongsan
*Mahidol University International College*

Srikanta Tirthapura
*Iowa State University*, snt@iastate.edu

# Parallel Streaming Random Sampling

**Abstract**
This paper investigates parallel random sampling from a potentially-unending data stream whose elements are revealed in a series of element sequences (minibatches). While sampling from a stream was extensively studied sequentially, not much has been explored in the parallel context, with prior parallel random-sampling algorithms focusing on the static batch model. We present parallel algorithms for minibatch-stream sampling in two settings: (1) sliding window, which draws samples from a prespecified number of most-recently observed elements, and (2) infinite window, which draws samples from all the elements received. Our algorithms are computationally and memory efficient: their work matches the fastest sequential counterpart, their parallel depth is small (polylogarithmic), and their memory usage matches the best known.

**Disciplines**
Databases and Information Systems | Electrical and Computer Engineering

**Comments**
This is the pre-print of the article Tangwongsan, Kanat, and Srikanta Tirthapura. "Parallel Streaming Random Sampling." *arXiv preprint arXiv:1906.04120* (2019). Posted with permission.

# Parallel Streaming Random Sampling

Kanat Tangwongsan* Srikanta Tirthapura†

**Abstract**

This paper investigates parallel random sampling from a potentially-unending data stream whose elements are revealed in a series of element sequences (minibatches). While sampling from a stream was extensively studied sequentially, not much has been explored in the parallel context, with prior parallel random-sampling algorithms focusing on the static batch model. We present parallel algorithms for minibatch-stream sampling in two settings: (1) sliding window, which draws samples from a prespecified number of most-recently observed elements, and (2) infinite window, which draws samples from all the elements received. Our algorithms are computationally and memory efficient: their work matches the fastest sequential counterpart, their parallel depth is small (polylogarithmic), and their memory usage matches the best known.

## 1 Introduction

Consider a model of data processing where data is revealed to the processor in a series of element sequences (minibatches) of varying sizes. A minibatch must be processed soon after it arrives. However, the data is too large for all the minibatches to be stored within memory, though the current minibatch is available in memory until it is processed.

Such a minibatch streaming model is a generalization of the traditional data stream model, where data arrives as a sequence of elements. If each minibatch is of size 1, our model reduces to the streaming model. Use of minibatches are common. For instance, in a *data stream warehousing system* [JS15], data is collected for a specified period (such as an hour) into a minibatch and then ingested, and statistics and properties need to be maintained during the ingestion. Minibatches may be relatively large, potentially of the order of Gigabytes or more, and could leverage parallelism (e.g., a distributed memory cluster or a shared-memory multicore machine) to achieve the desired throughput. Furthermore, this model matches the needs of modern "big data" stream processing systems such as Apache Spark Streaming [ZDL+13], where newly-arrived data is stored as a distributed data set (an "RDD" in Spark) that is processed in parallel. Queries are posed on all the data received up to the most recent minibatch.

This paper investigates the foundational aggregation task of random sampling in the minibatch streaming model. Algorithms in this model observe a (possibly infinite) sequence of minibatches $B_1, B_2, \ldots, B_t, \ldots$. We consider the following variants of random sampling, all of which have been well studied in the context of sequential streaming algorithms. In the **infinite window** model, a random sample is chosen from all the minibatches seen so far. Thus, after observing $B_t$, a random sample is drawn from $\cup_{i=1}^{t} B_i$. In the **sliding window** model with window size $w$, the sample after observing $B_t$ is chosen from the $w$ most-recent elements. Typically, the window size $w$ is much larger than a minibatch size. [1] In this work, the window size $w$ is provided at query time, but an upper bound $W$ on $w$ is known beforehand.

*CS Program, Mahidol University International College, kanat.tan@mahidol.edu

†Dept. of Electrical and Computer Engineering, Iowa State University, snt@iastate.edu

[1] One could also consider a window to be the $w$ most recent minibatches, and similar techniques are expected to work.

We focus on optimizing the work and parallel depth of our parallel algorithms. This is a significant point of departure from the traditional streaming algorithms literature, which has mostly focused on optimizing the memory consumed. In addition, we consider memory to be a scarce resource and design for scenarios where the size of the stream is very large—and the stream, or even a sliding window of the stream, does not fit in memory. Hence, this work strives for parallel computational efficiency in addition to memory efficiency.

## 1.1 Our Contributions

We present parallel random-sampling algorithms for the minibatch streaming model, in both infinite window and sliding window settings. These algorithms can use the power of shared-memory parallelism to speedup the processing of a new minibatch as well as a query for random samples.

▷ *Efficient Parallel Algorithms.* Our algorithms are provably efficient in parallel processing. We analyze them in the work-depth model, showing (1) they are work-efficient, i.e., total work across all processors is of the same order as an efficient sequential algorithm, and (2) their parallel depth is logarithmic in the target sample size, which implies that they can use processors nearly linear in the input size while not substantially increasing the total work performed. In the infinite-window case, the algorithm is work-optimal since the total work across all processors matches a lower bound on work, which we prove in this paper, up to constant factors. Interestingly, for all our algorithms, the work of the parallel algorithm is sublinear in the size of the minibatch.

▷ *Small Memory.* While the emphasis of this work is on improving processing time and throughput, our algorithms retain the property of having a small memory footprint, matching the best sequential algorithms from prior work.

Designing such parallel algorithms requires overcoming several challenges. Sliding-window sampling is typically implemented with Priority Sampling [BDM02, BOZ09], whose work performed (per minibatch) is linear in the size of the minibatch. Parallelizing it reduces depth but does not reduce work. Generating skip offsets, à la Algorithm Z [Vit85] (reservoir sampling), can significantly reduce work but offers no parallelism. Prior algorithms, such as in [Vit85], seem inherently sequential, since the next location to sample from is derived as a function of the previously chosen location. This work introduces a new technique called $\mathsf{R}^3$ sampling, which combines **r**eversed **r**eservoir sampling with **r**ejection sampling. $\mathsf{R}^3$ sampling is a new perspective on Priority Sampling that mimics the sampling distribution of Priority Sampling, but is simpler and has less computational dependency, making it amendable to parallelization. To enable parallelism, we draw samples simultaneously from different areas of the stream using a close approximation of the distribution. This leads to slight oversampling, which is later corrected by rejection sampling. We show that all these steps can be implemented in parallel. In addition, we develop a data layout that permits convenient update and fast queries. As far as we know, this is the first efficient parallelization of the popular reservoir-sampling-style algorithms.

## 1.2 Related Work

Reservoir sampling (attributed to Waterman) was known since the 1960s. There has been much follow-up work, including methods for speeding up reservoir sampling by "skipping past elements" [Vit85], weighted reservoir sampling [ES06], and sampling over a sliding window [BOZ09, XTB08, BDM02, GL08].

The difference between the distributed streams model [Cor13, CMYZ12, GT01, CTW16] considered earlier, and the parallel stream model considered here is that in the distributed streams model, the focus is on minimizing the communication between processors while in our model, processors can coordinate using shared memory, and the focus is on work-efficiency of the parallel algorithm. Prior work on shared-memory parallel streaming has considered frequency counting [DAAE09, TTW14] and aggregates on graph streams [TPT13], but to our knowledge, there is none so far on random sampling. Prior work on warehousing

of sample data [BH06] has considered methods for sampling under minibatch arrival, where disjoint partitions of new data are handled in parallel. Our work also considers how to sample from a single partition in parallel, and can be used in conjunction with a method such as [BH06].

## 2 Preliminaries and Notation

A *stream* $\mathcal{S}$ is a potentially infinite sequence of minibatches $B_1 B_2, \ldots$, where each minibatch consists of one or more elements. Let $\mathcal{S}_t$ denote the stream so far until time $t$, consisting of all elements in minibatches $B_1, B_2, \ldots, B_t$. Let $n_i = |B_i|$ and $N_t = \sum_{i=1}^{t} n_i$, so $N_t$ is the size of $\mathcal{S}_t$. The size of a minibatch is not known until the minibatch is received, and the minibatch is received as an array in memory. A *stream segment* is a finite sequence of consecutive elements of a stream. For example, a minibatch is a stream segment. A *window* of size $w$ is the stream segment consisting of the $w$ most recent elements.

Let $[n]$ denote the set $\{1, \ldots, n\}$. For sequence $X = \langle x_1, x_2, \ldots, x_{|X|} \rangle$, the $i$-th element is denoted by $X_i$ or $X[i]$. For convenience, negative index $-i$, written $X[-i]$ or $X_{-i}$, refers to the $i$-th index from the right end—i.e., $X[|X| - i + 1]$. Following common array slicing notation, let $X[a:]$ be the subsequence of $X$ starting from index $a$ onward. An event happens with high probability (**whp**) if it happens with probability at least $1 - n^{-c}$ for some constant $c \geqslant 1$. Let $\texttt{UniformSample}(a, b)$, $a \leqslant b$, be a function that returns an element from $\{a, a+1, \ldots, b\}$ chosen uniformly at random. For $0 < p \leqslant 1$, $\texttt{coin}(p) \in \{H, T\}$ returns heads ($H$) with probability $p$ and tails ($T$) with probability $1 - p$. For $m \leqslant n$, an $m$-permutation of a set $S$, $|S| = n$, is an ordering of $m$ elements chosen from $S$.

We analyze algorithms in the work-depth model assuming concurrent reads and arbitrary-winner concurrent writes. The *work* of an algorithm is the total operation count, and *depth* (also called parallel time or span) is the length of the longest chain of dependencies within that algorithm. The gold standard in this model is for an algorithm to perform the same amount of work as the best sequential counterpart (work-efficient) and to have polylogarithmic depth. This setting has been fertile ground for research and experimentation on parallel algorithms. Moreover, results in this model are readily portable to other related models, e.g., exclusive read and exclusive write, with a modest increase in cost (see, e.g., [BM04]).

*Parallel semisorting* is the problem of reordering an input sequence of keys so that like sorting, equal keys are arranged contiguously, but unlike sorting, different keys are not necessarily in sorted order. We rely on the following result:

**Theorem 1 ([GSSB15])** *On input a sequence $X = \langle x_1, \ldots, x_n \rangle$, where $x_i$ can be uniformly hashed to $[n^k]$ in constant time, parallel semisorting can be implemented in $O(n)$ expected work and space, and $O(\log n)$ depth **whp**.*

A *Random permutation* of a finite set $n$-element set $X$ can be generated in parallel using $O(n)$ work and $O(\log n)$ depth (e.g., [Rei85]). Later, we use the following bound:

**Theorem 2 (Theorem 1.1 of [DP09])** *Let $X = \sum_{i=1}^{n} X_i$, where $X_i$s are independent random variables in $[0, 1]$. For $t > 2e \cdot \mathbf{E}[X]$, we have $\mathbf{Pr}[X \geqslant t] \leqslant 2^{-t}$.*

We measure the space complexity of our algorithms in terms of the number of elements stored. Our space bounds do not represent bit complexity. Often, the space used by the algorithm is a random variable, so we present bounds on the expected space complexity.

## 3 Parallel Sampling from a Sliding Window

This section discusses parallel algorithms for sampling without replacement from a sliding window (SWOR-Sliwin). The task is as follows: For target sample size $s$ and maximum window size $W$, SWOR-Sliwin is

to maintain a data structure $\mathcal{R}$ that supports two operations: (i) $\texttt{insert}(B_i)$ incorporates a minibatch $B_i$ of new elements that arrived at time $i$ into $\mathcal{R}$, and (ii) For parameters $q \leqslant s$ and $w \leqslant W$, $\texttt{sample}(q, w)$ when posed at time $i$ returns a random sample of $q$ elements chosen uniformly without replacement from the $w$ most recent elements in $\mathcal{S}_i$.

In our implementation, $\texttt{sample}(q, w)$ does something stronger, and returns a $q$-permutation (not only a set) chosen uniformly at random from the $w$ newest elements from $\mathcal{R}$—this can additionally be used to generate a sample of any size $j$ from 1 till $q$ by only consider the first $j$ elements of the permutation.

One popular approach to sampling from a sliding window in the sequential setting [BDM02, BOZ09] is the Priority Sampling algorithm: Assign a random priority to each stream element, and in response to `sample(s, w)` return the $s$ elements with the smallest priorities among the latest $w$ arrivals. To reduce the space consumption to be sub-linear in the window size, the idea is to store only those elements that can potentially be included in the set of $s$ smallest priorities for any window size $w$. A stream element $e$ can be discarded if there are $s$ or more elements with a smaller priority than $e$ that are more recent than $e$. Doing so systematically leads to an expected space bound of $O(s + s\log(W/s))$ [BDM02][2].

As stated, this approach expends work linear in the stream length to examine/assign priorities, but ends up choosing only a small fraction of the elements examined. This motivates the question: *How can one determine which elements to choose, ideally in parallel, without expending linear work to generate or look at random priorities?* We are most interested in the case where $W \gg n_i \geqslant s$, where $n_i$ is the size of minibatch $i$. The main result of this section is as follows:

**Theorem 3** *There is a data structure for* SWOR-Sliwin *that uses $O(s + s\log(W/s))$ expected space and supports the following operations:*

*(i)* $\texttt{insert}(B)$ *for a new minibatch $B$ uses $O(s + s\log(\frac{W}{s}))$ work and $O(\log W)$ parallel depth; and*

*(ii)* `sample(q, w)` *for sample size $q \leqslant s$ and window size $w \leqslant W$ uses $O(q)$ work and $O(\log W)$ parallel depth.*

Note that the work of the data structure for inserting a new minibatch is only logarithmic in the maximum window size $W$ and independent of the size of the minibatch. To prove this theorem, we introduce $\mathsf{R}^3$ sampling, which brings together reversed reservoir sampling and rejection sampling. We begin by describing reversed reservoir sampling, a new perspective on priority sampling that offers more parallelism opportunities. After that, we show how to implement this sampling process efficiently in parallel via rejection sampling.

### 3.1 Simple Reversed Reservoir Algorithm

We now describe *reversed reservoir* (RR) sampling, which mimics the behaviors of priority sampling but provides more independence and more parallelism opportunities. This process will be refined and expanded in subsequent sections. After observing sequence $X$, Simple-RR(Algorithm 1) yields uniform sampling without replacement of up to $s$ elements for any suffix of $X$. For maximum sample size $s > 0$ and integer $i > 0$, define $p_{-i}^{(s)} = \min\left(1, \frac{s}{i}\right)$.

We say the $i$-th most-recent element has age $i$; this position/element will be called age $i$ when the context is clear. The algorithm examines $X$ in reverse, $X[-1], X[-2], \ldots$, and stores a subset in data structure $A$, which records the index of an element in $X$ as well as a slot (from $[s]$) into which the element is mapped. Multiple elements may be mapped to the same slot. The probability of $X[-i]$ being chosen into $A$ decreases as $i$ increases.

This algorithm samples an element at index $-i$ (age $i$) with probability $p_{-i}^{(s)} = \min(1, s/i)$, the same probability it would in priority sampling (aka.bottom-$k$ sampling). Also, every element sampled is assigned a random *slot* number between 1 and $s$. This is used to generate a random permutation.

[2]The original algorithm stores the largest priorities but is equivalent to our view.

**Algorithm 1:** SIMPLE-RR$(X, s)$ — Naïve reversed reservoir sampling

**Input:** a stream segment $X = \langle x_1, \ldots, x_{|X|} \rangle$ and a parameter $s > 0$, $s \leqslant |X|$.
**Output:** a set $\{(k_i, \ell_i)\}$, where $k_i$ is an index into $X$ and $\ell_i \in [s]$

$\pi \leftarrow$ Random permutation of $[s]$, $A_0 = \varnothing$
**for** $i = 1, 2, \ldots, s$ **do** $A_i = A_{i-1} \cup \{(i, \pi_i)\}$
**for** $i = s+1, 2, \ldots, |X|$ **do**
    **if** $\texttt{coin}(p_{-i}^{(s)}) == H$ **then**
        $\ell \leftarrow \texttt{UniformSample}(1, s)$
        $A_i = A_{i-1} \cup \{(i, \ell)\}$
    **else** $A_i = A_{i-1}$
**return** $A_{|X|}$

Reserved reservoir sampling has a number of nice properties:

**Lemma 4** *For input a stream segment $X$ and a parameter $0 < s \leqslant |X|$, the number of elements sampled by* SIMPLE-RR *is expected $s + O(s \log(|X|/s))$.*

*Proof:* The elements at indicies $-1, -2, \ldots, -s$ are always chosen, contributing $s$ elements to the output. For $i = s+1, \ldots, |X|$, the probability that $x_{-i}$ is sampled is $s/i$, so the expected number of samples among these elements is

$$\sum_{i=s+1}^{|X|} \frac{s}{i} \leqslant s \int_{x=s}^{|X|} \frac{s}{x} dx = s \ln(|X|/s),$$

which completes the proof. ∎

Let $A$ denote the result of SIMPLE-RR. Using this, sampling $s$ elements without replacement from any suffix of $X$ is pretty straightforward. Define

$$\chi(A) = (\nu_A(1), \nu_A(2), \ldots, \nu_A(s))$$

where $\nu_A(\ell) = \arg\max_{k \geqslant 1}\{(k, \ell) \in A\}$ is[3] the oldest element assigned to slot $\ell$. Given $A$, we can derive $A_i$ for any $i \leqslant |X|$ by considering the appropriate subset of $A$. We have that $\chi(A_i)$ is an $s$-permutation of the $i$ most recent elements of $X$.

**Lemma 5** *If $R$ is any $s$-permutation of $X[-i:]$, then*

$$\mathbf{Pr}[R = \chi(A_i)] = \frac{(i-s)!}{i!} \tag{3.1}$$

*Proof:* We proceed by induction on $i$. The base case of $i = s$ is easy to verify since $\pi$ is a random permutation of $[s]$ and $\chi(A_s)$ is a permutation of $X[-s:]$ according to $\pi$. For the inductive step, assume that equation (3.1) holds for for any $R$ that is an $s$-permutation of $X[-i:]$. Now let $R'$ be an $s$-permutation of $X[-(i+1):]$. Consider two cases:

- Case I: $x_{-(i+1)}$ appears in $R'$, say at at $R'_\ell$. For $R' = \chi(A_{i+1})$, it must be the case that $x_{-(i+1)}$ was chosen and was assigned to slot $\ell$. Furthermore, $\chi(A_i)$ must be identical to $R'$ except in position $\ell$,

[3] Because $|X| \geqslant s$, the function $\nu$ is always defined.

where it could have been any of the $i-(s-1)$ choices. This happens with probability

$$(i-[s-1])\cdot\tfrac{(i-s)!}{i!}\cdot p_{-(i+1)}\cdot\tfrac{1}{s} = \frac{(i-s)!}{i!}\cdot\frac{i-s+1}{s}\cdot\frac{s}{i+1} = \frac{([i+1]-s)!}{(i+1)!}$$

- Case II: $x_{-(i+1)}$ does not appear in $R'$. Therefore, $R'$ must be an $s$-permutation of $X[-i:]$ and $x_{-(i+1)}$ was not sampled. This happens with probability

$$\tfrac{(i-s)!}{i!}\cdot(1-p_{-(i+1)}) = \tfrac{(i-s)!}{i!}\cdot\tfrac{i-s+1}{i+1} = \tfrac{([i+1]-s)!}{(i+1)!}$$

In either case, this gives the desired probability. ∎

Note that the space taken by this algorithm (the size of $A_{|X|}$) is $O(s + s\log(|X|/s))$, which is optimal [GL08]. The steps are easily parallelizable but still need $O(|X|)$ work, which can be much larger than the $(s + s\log(|X|/s))$ bound on the number of elements the algorithm must sample. We improve on this next.

### 3.2 Improved Single-Element Sampler

This section addresses the special case of $s = 1$. Our key ingredient will be the ability to compute the next index that will be sampled, without touching elements that are not sampled.

Let $x_{-i}$ be an element just sampled. We can now define a random variable $\textsc{Skip}(i)$ that indicates how many elements past $x_{-i}$ will be skipped over before selecting index $-(i+\textsc{Skip}(i))$ according to the distribution given by Simple-RR. Conveniently, this random variable can be efficiently generated in $O(1)$ time using the inverse transformation method [Ros09] because its cumulative distribution function (CDF) has a simple, efficiently-solvable form: $\mathbf{Pr}[\textsc{Skip}(i)\leqslant k] = 1-\prod_{t=i+1}^{i+k}(1-p_{-t}) = 1-\frac{i}{i+k} = \frac{k}{i+k}$. This leads to the following improved algorithm:

**Algorithm 2:** Fast-Single-RR$(X)$ — Fast RR sampling for $s = 1$

**Input:** a stream segment $X = \langle x_1,\ldots,x_{|X|}\rangle$.
**Output:** a set $\{(k_i,\ell_i)\}$, where $k_i$ is an index into $X$ and $\ell_i = 1$
▷ *The same input/output behaviors as* Simple-RR.

```
1: i ← 1
2: while i < |X| do
3:     A ← A ∪ {(i, 1)}
4:     i ← i + Skip(i)
5: return A
```

This improvement significantly reduces the number of iterations:

**Lemma 6** *Let $T_{\mathsf{FSR}}(n)$ be the number of times the* ***while****-loop in the* Fast-Single-RR *algorithm is executed on input $X$ with $n = |X|$. Then, $\mathbf{E}[T_{\mathsf{FSR}}(n)] = O(1+\log(n))$. Also, for $m \geqslant n$ and $c \geqslant 4$, $\mathbf{Pr}[T_{\mathsf{FSR}}(n) \geqslant 1 + c\cdot\log(m)] \leqslant m^{-c}$.*

*Proof:* Let $Z_i$ be an indicator variable for whether $x_{-i}$ contributes to an iteration of the **while**-loop. Hence, $T_{\mathsf{FSR}}(n) = 1+Z$, where $Z = \sum_{i=2}^{|X|} Z_i$. But $\mathbf{Pr}[Z_i = 1] = 1/i$, so $\mathbf{E}[Z] = \frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \leqslant \ln n$. This proves the expectation bound.

Let $t = c \log_2 m$, so $t \geqslant 4 \cdot \log_2 m > 2e \ln(2) \cdot \log_2 m = 2e \ln m \geqslant 2e \ln n \geqslant 2e\,\mathbf{E}[Z]$. Therefore, by Theorem 2, we have that

$$\mathbf{Pr}[T_{\mathsf{FSR}}(n) \geqslant 1 + c \cdot \log_2(m)] \leqslant \mathbf{Pr}[Z \geqslant c \cdot \log_2 m] \leqslant 2^{-t} = m^{-c},$$

which concludes the proof. ∎

Immediately, this means that if $A = \textsc{Fast-Single-RR}(X)$ is kept as a simple sequence (e.g., an array), the running time—as well as the length of $A$—will be $O(1 + \log(|X|))$ in expectation. Moreover, $A$ follows the same distribution as Simple-RR with $s = 1$, only more efficiently computed.

**Remark 3.1** *Vitter [Vit85] studied a related problem that requires sampling from the same distribution as* Simple-RR. *He developed a sophisticated algorithm (Algorithm Z) for generating the skip offsets for $s \geqslant 1$. Our* Fast-Single-RR *algorithm addresses the special case where $s = 1$, which is significantly simpler and does not require the same level of machinery as Vitter's Algorithm Z.*

### 3.3 Improved Multiple-Element Sampler

In the general case of reversed reservoir sampling, generating skip offsets from the distribution for $s > 1$ turns out to be significantly more involved than for $s = 1$. While this is still possible, e.g., using a variant of Vitter's Algorithm Z [Vit85], prior algorithms appear inherently sequential.

This section describes a new parallel algorithm that builds on Fast-Single-RR. In broad strokes, it first "oversamples" using a simpler distribution and subsequently, "downsamples" to correct the sampling probability. To enable parallelism, we logically divide the stream segment into $s$ "tracks" of roughly the same size and have the single-element algorithm work on each track in parallel.

**Division of Work via Tracks.** The aim of the first phase is to sample each element $x_{-i}$ with a slightly higher probability than $p^{(s)}_{-i}$, in a way that results in about $s + s\log(|X|/s)$ elements sampled at the end of the phase—and the sampling could be carried out in parallel, with depth less than $s$. To this end, we logically divide the stream segment into $s$ tracks of about the same size and in parallel, have the single-element algorithm work on each track.

**Track View.** Define $\textsc{Create-View}(X, k)$ to return a view corresponding to track $k$ on $X$: if $Y = \textsc{Create-View}(X, k)$, then $Y_{-i}$ is $X[-\alpha^{(k)}_s(i)]$, where $\alpha^{(k)}_s(i) = i \cdot s + k$. That is, track $k$ contains, in reverse order, indices $-(s+k), -(2s+k), -(3s+k), \ldots$. Importantly, these views never have to be materialized.

Algorithm 3 combines the ideas developed so far. We now argue that Fast-RR yields the same distribution as Simple-RR:

**Lemma 7** *Let $A$ be a return result of* $\textsc{Fast-RR}(X, s)$. *Then, for $j = 1, \ldots, |X|$ and $\ell \in [s]$, $\mathbf{Pr}[(j, \ell) \in A] = \frac{1}{s} \cdot p^{(s)}_{-j}$.*

*Proof:* For $j \leqslant s$, age $j$ is paired with a slot $\ell$ drawn from a random permutation of $[s]$, so $\mathbf{Pr}[(j, \ell) \in A] = \frac{1}{s} = \frac{1}{s} \cdot 1 = \frac{1}{s} \cdot p^{(s)}_{-j}$. For $j > s$, write $j$ as $j = s \cdot i + \tau$, so age $j$ appears as age $i$ in view $X^{(\tau)}_{\bullet}$. Now age $j$ appears in $A$ if both of these events happen: (1) age $i$ was chosen into $T_\tau$ and (2) the coin turned up heads so it was retained in $T'_\tau$. These two independent events happen together with probability

$$p^{(1)}_{-i} \cdot \frac{i \cdot s}{\alpha^{(\tau)}_s(i)} = \frac{1}{i} \cdot \frac{i \cdot s}{s \cdot i + \tau} = \frac{s}{j} = p^{(s)}_{-j}.$$

**Algorithm 3:** FAST-RR$(X, s)$ — Fast reversed reservoir sampling

**Input:** a stream segment $X = \langle x_1, \ldots, x_{|X|} \rangle$ and a parameter $s > 0$, $s \leqslant |X|$.
**Output:** a set $\{(k_i, \ell_i)\}$, where $k_i$ is an index into $X$ and $\ell_i \in [s]$

$\pi \leftarrow$ draw a random permutation of $[s]$
$T_0 \leftarrow \{(i, \pi_i) \mid i = 1, 2, \ldots, s\}$
**for** $\tau = 1, 2, \ldots, s$ **in parallel do**
  $X_\bullet^{(\tau)} \leftarrow$ CREATE-VIEW$(X, \tau)$
  $T_\tau \leftarrow$ FAST-SINGLE-RR$(X_\bullet^{(\tau)})$
  $T'_\tau \leftarrow \{(i, \ell) \in T_\tau \mid \texttt{coin}(i \cdot s/\alpha_s^{(\tau)}(i)) = H\}$ ▷ *filter, keep if coin shows heads*
  $T''_\tau \leftarrow \{(i, \texttt{UniformSample}(1, s)) \mid (i, \_) \in T'_\tau\}$ ▷ *map*
**return** $T_0 \cup T''_1 \cup T''_2 \cup \cdots \cup T''_s$

Once age $j$ is chosen, it goes to slot $\ell$ with probability $1/s$. Hence, $\mathbf{Pr}[(j, \ell) \in A] = \frac{1}{s} \cdot p_{-j}^{(s)}$. ■

Because the cost of an algorithm depends on the choice of data structures, we defer the cost analysis of FAST-RR to the next section, after we discuss how the reserved samples will be stored.

### 3.4 Storing and Retrieving Reserved Samples

*How should we store the sampled elements?* An important design goal is for samples of any size $q \leqslant s$ to be generated without first generating $s$ samples. To this end, observe that restricting $\chi(A)$ to its first $q \leqslant s$ coordinates yields a $q$-permutation over the input. This motivates a data structure that stores the contents of different slots separately.

Denote by $\mathcal{R}(A)$, or simply $\mathcal{R}$ in clear context, the *binned-sample* data structure for storing reserved samples $A$. The samples are organized by their slot numbers $(\mathcal{R}_i)_{i=1}^{s}$, with $\mathcal{R}_i$ storing slot $i$'s samples. Within each slot, samples are binned by their ages. In particular, each $\mathcal{R}_i$ contains $\lceil \log_2(\lceil |X|/s \rceil) \rceil + 1$ bins, numbered $0, 1, 2, \ldots, \lceil \log_2(\lceil |X|/s \rceil) \rceil$—with bin $k$ storing ages $j$ in the range $2^{k-1} < \lceil j/s \rceil \leqslant 2^k$. Below, bin $t$ of slot $i$ will be denoted by $\mathcal{R}_i[t]$.

Additional information is kept in each bin for fast queries: every bin $k$ stores $\phi(k)$, defined to be the age of the oldest element in bin $k$ and all younger bins for the same slot number.

Below is an example. Use $s = 3$ and $|X| = 16$. Let the result from FAST-RR be

$$A = \{(1, 2), (2, 3), (3, 1), (7, 1), (10, 3), (11, 3), (14, 2)\}.$$

Then, $\mathcal{R}$ keeps the following bins, together with $\phi$ values:

| **Bin**: | $\mathcal{R}_i[0]$ | $\mathcal{R}_i[1]$ | $\mathcal{R}_i[2]$ | $\mathcal{R}_i[3]$ |
|---|---|---|---|---|
| Slot $i = 1$ | $\{3\}_{\phi=3}$ | $\varnothing_{\phi=3}$ | $\{7\}_{\phi=7}$ | $\varnothing_{\phi=7}$ |
| Slot $i = 2$ | $\{1\}_{\phi=1}$ | $\varnothing_{\phi=1}$ | $\varnothing_{\phi=1}$ | $\{14\}_{\phi=14}$ |
| Slot $i = 3$ | $\{2\}_{\phi=2}$ | $\varnothing_{\phi=2}$ | $\{10, 11\}_{\phi=11}$ | $\varnothing_{\phi=11}$ |

From this construction, the following claims can be made:

**Claim 8** *The expected size of the bin $\mathcal{R}_i[t]$ is $\mathbf{E}[|\mathcal{R}_i[t]|] \leqslant 1$.*

*Proof:* Bin $t$ of $\mathcal{R}_i$ is responsible for negative indices $j$ in the range $2^{t-1} < \lceil -j/s \rceil \leqslant 2^t$, for a total of $s(2^t - 2^{t-1}) = s \cdot 2^{t-1}$ indices. Among these indices, the index that has the highest probability of being

sampled is $-(s2^{t-1}+1)$, which is sampled into slot $i$ with probability $\frac{1}{s}\cdot\frac{s}{s2^{t-1}+1}\leqslant\frac{1}{s\cdot 2^{t-1}}$. Therefore,

$$\mathbf{E}[|\mathcal{R}_i[t]|]\leqslant s\cdot 2^{t-1}\cdot\frac{1}{s\cdot 2^{t-1}}=1,$$

which concludes the proof. ∎

**Claim 9** *The size of slot $\mathcal{R}_i$ is expected $O(1+\log(|X|/s))$. Furthermore, for $c\geqslant 4$, $\mathbf{Pr}[|\mathcal{R}_i|\leqslant 1+c\log_2(|X|)]\geqslant 1-|X|^{-c}$.*

*Proof:* Let $Y_t=\mathbf{1}_{\{x_{-t}\text{ is chosen into slot }i\}}$, so $|\mathcal{R}_i|=\sum_{t=1}^{|X|}Y_t$. Since $\mathbf{E}[Y_t]=p_{-t}^{(s)}/s=\frac{1}{s}\min(1,s/t)$, we have

$$\mathbf{E}[|\mathcal{R}_i|]=\sum_{t=1}^{|X|}\mathbf{E}[Y_t]=1+\sum_{t=s+1}^{|X|}\frac{1}{t}\leqslant 1+\int_{t=s}^{|X|}\frac{dt}{t}=1+\ln\left(\tfrac{|X|}{s}\right),$$

which proves the expectation bound. Because $Y_t$'s are independent, using an argument similar to the proof of Lemma 6, we have the probability bound. ∎

**Data Structuring Operations.** Algorithm 4 shows algorithms for constructing a binned-sample data structure and answering queries. To Construct a binned-sample data structure, the algorithm first arranges the entries into groups by slot number, using a parallel semisorting algorithm, which reorders an input sequence of keys so that like sorting, equal keys are arranged contiguously, but unlike sorting, different keys are not necessarily in sorted order. Parallel semisorting of $n$ elements can be achieved using $O(n)$ expected work and space, and $O(\log n)$ depth [GSSB15]. It then, in parallel, processes each slot, putting every entry into the right bin. Moreover, it computes a min-prefix, yielding $\phi(\cdot)$ for all bins. There is not much computation within a slot, so we do it sequentially but the different slots are done in parallel. To answer a Sample query, the algorithm computes, for each slot $i$, the oldest age within $X[-w:]$ that was assigned to slot $i$. This can be found quickly by figuring out the bin $k$ where $w$ should be. Once this is known, it simply has to look at $\phi$ of bin $k-1$ and go through the entries in bin $k$. This means a query touches at most two bins per slot.

**Cost Analysis.** We now analyze Fast-RR, Construct, and Sample for their work and parallel depth. More concretely, the following claims are made:

**Claim 10** *By storing $T_0, T_i$'s, and $T_i'$'s as simple arrays,* Fast-RR$(X,s)$ *runs in expected $O(s+s\log\frac{|X|}{s})$ work and $O(\log|X|)$ parallel depth.*

*Proof:* Generating a permutation of $[s]$ can be done in $O(s)$ work and $O(\log s)\leqslant O(\log|X|)$ depth. There are $s$ tracks, each, in parallel, calling Fast-Single-RR, which is a sequential algorithm. Once $T_\tau$ is known, the remaining steps are simple `map` and `filter` operations, which have $O(|T_\tau|)$ work and $O(O(|T_\tau|)\leqslant O(\log|X|)$ depth. Therefore, the dominant cost is Fast-Single-RR. By Lemma 6, each track performs $O(1+\ln(|X|/s))$ work in expectation. Summing across $s$ tracks, the total work is expected $O(s+s\log(|X|/s))$. In terms of parallel depth, by Lemma 6, each track finishes after $1+4\log_2(|X|)$ iterations with probability at least $1-|X|^{-4}$. Applying the union bound, we have that the expected depth overall is at most $O(\log|X|)$ provided that $|X|\geqslant s$. ∎

**Algorithm 4:** Construction of binned-sample data structure and query

▷ *Below, use the convention that* $\max \varnothing = -\infty$

1: CONSTRUCT$(A, n, s)$**:**
  **Input:** $A$ is a sequence of reserved samples, $n$ is the length of the underlying stream segment $X$, and $s$ is the target sample size used to generate $A$.
  **Output:** an instance of binned-sample structure $\mathcal{R}(A)$
2: Use semisorting to arrange $A$ into $G_1, G_2, \ldots, G_s$ by slot number
3: **for** $i = 1, \ldots, s$ **in parallel do**
4: Create bins $\mathcal{R}_i[0], \ldots, \mathcal{R}_i[\beta], \beta = \lceil \log_2(\lceil n/s \rceil) \rceil$
5: **foreach** $(j, \_) \in G_i$ **do**
6: Write $j$ into $\mathcal{R}_i[k]$, where $2^{k-1} < \lceil j/s \rceil \leqslant 2^k$
7: Let $\phi(\mathcal{R}_i[0]) = \max \mathcal{R}_i[0]$
▷ *prefix max*
8: **for** $k = 1, \ldots, \beta$ **do**
9: $\phi(\mathcal{R}_i[k]) \leftarrow \max(\phi(\mathcal{R}_i[k-1]), \max \mathcal{R}_i[k])$
10: **return** $\mathcal{R}$
11: SAMPLE$(\mathcal{R}, q, w)$**:**
  **Input:** $\mathcal{R}$ is a binned-sample structure, $q$ is the number of samples desired, $w$ tells the algorithm to draw sample from $X[-w:]$.
  **Output:** a $q$-permutation of $X[-w:]$
12: **for** $i = 1, \ldots, q$ **in parallel do**
13: Let $k$ be such that $2^{k-1} < \lceil w/s \rceil \leqslant 2^k$
14: $\gamma \leftarrow \max\{j \in \mathcal{R}_i[k] \mid j \leqslant w\}$ ▷ *The oldest that is at least as young as* $w$
15: $r_i \leftarrow \max(\gamma, \phi(\mathcal{R}_i[k-1]))$
16: **return** $(r_1, r_2, \ldots, r_q)$

**Claim 11** CONSTRUCT$(A, n, s)$ *runs in* $O(s + s \log \frac{n}{s})$ *work and* $O(\log n)$ *parallel depth.*

*Proof:* Let $\psi = s + O(s \log(n/s))$. The algorithm starts with a semisorting step, which takes $O(|A|)$ work and $O(\log |A|)$ depth to arrange the entries of $A$ into $G_1, \ldots G_2$. Since $|A|$ is expected $\psi$ but never exceeds $n$. This step takes $O(\psi)$ work and $O(\log n)$ depth. For each $i = 1, \ldots, s$, the size of $G_i$ is expected $\frac{1}{s}\psi$ (Claim 9). Therefore, the work performed for each $i$ is expected $\frac{1}{s}\psi$, for a total of $\psi$ across all $s$ slots in expectation. Because by Claim 9, the size of a $G_i$ exceeds $1 + 4\log_2(n)$ with probability at most $1/n^4$. The overall depth is at most $O(\log n)$ **whp**. ■

**Lemma 12** SAMPLE$(\mathcal{R}, q, t)$ *runs in* $O(q)$ *work and* $O(\log n)$ *parallel depth, where n is the length of X on which* $\mathcal{R}$ *was built.*

*Proof:* The algorithm looks into $q$ slots in parallel. For each slot $\mathcal{R}_i$, it takes $T_i = O(1 + |\mathcal{R}_i[k]|)$ sequential time, which is expected $O(1)$ by Claim 8. Hence, the total work is expected $O(s)$. Then, it follows from Claim 9 that $T_i$ exceeds $1 + 4\log_2(n)$ with probability at most $1/n^4$, so the overall depth is at most $O(\log n)$ **whp**. ■

### 3.5 Handling Minibatch Arrival

This section describes how to incorporate a minibatch into our data structure to maintain a sliding window of size $W$. Assume that the minibatch size is $n_i \leqslant W$. If not, we can only consider its $W$ most recent elements.

When a minibatch arrives, retired sampled elements must be removed and the remaining sampled elements are "downsampled" to maintain the correct distribution.

Remember that the number of selected elements is $O(s + s\log(W/s))$ in expectation, so we have enough budget in the work bound to make a pass over them to filter out retired elements. Instead of revisiting every element of the window, we apply the process below to the selected elements to maintain the correct distribution. Notice that an element at age $i$ was sampled into slot $\ell$ with probability $\frac{1}{s}p_{-i}^{(s)}$. A new minibatch will cause this element to shift to age $j$, $j > i$, in the window. At age $j$, an element is sampled into slot $\ell$ with probability $\frac{1}{s}p_{-j}^{(s)}$. To correct for this, we flip a coin that turns up heads with probability $p_{-j}^{(s)}/p_{-i}^{(s)} \leqslant 1$ and retain this sample only if the coin comes up heads.

Therefore, `insert`$(B_i)$, $|B_i| = n_i$ handles a minibatch arrival as follows:

Step i: Discard and downsample elements (above) in $\mathcal{R}$; the index shifts by $n_i$.

Step ii: Apply Fast-RR on $B_i$, truncated to the last $W$ elements if $n_i > W$.

Step iii: Run Construct on the result of Fast-RR with a modification where the it appends to an existing $\mathcal{R}$ as opposed to creating a new structure.

Overall, this leads to the following cost bound for `insert`:

**Lemma 13** `insert` *takes* $O(s + s\log(W/s))$ *work and* $O(\log W)$ *depth.*

*Proof:* The cost of discarding and downsampling elements in the existing $\mathcal{R}$ is no more than the cost of running Construct with $n = W$ because the cost of the former is upper-bounded by the cost of going through every element in $\mathcal{R}$ once. Hence, this step takes $O(s + s\log(W/s))$ work and $O(\log W)$ depth. When Fast-RR is run, it is run with input length $\min(W, n_i)$. Thus, it performs no more work than allowed by the lemma. Finally, Construct is called with $n \leqslant W$, costing $O(s + s\log(W/s))$ work and $O(\log W)$ depth, which is also the total cost of `insert`. ■

### 3.6 Total Cost of SWOR-Sliwin

The cost of handling a minibatch's arrival is given by Lemma 13, and the cost of answering a query is given by Lemma 12. Altogether, this proves Theorem 3. Furthermore, across $t$ minibatches, the total work is

$$O\left(t + \sum_{i=1}^{t}(s + s\ln(W/s))\right).$$

## 4 Parallel Sampling from an Infinite Window

This section addresses sampling without replacement from the infinite window, which includes all elements seen so far in the stream. This is formulated as the SWOR-Infwin task: For each time $i = 1, \ldots, t$, maintain a random sample of size $\min\{s, N_i\}$ chosen uniformly without replacement from $\mathcal{S}_i$.

While SWOR-Infwin can be reduced to the sliding-window setting, by setting the window size to the number of elements received so far, in this section, we show that there is an algorithm for infinite window that is simpler and more efficient. We further show it to be work optimal, up to constant factors.

For $p, q \in [r]$, let $\mathcal{H}(p, q, r)$ be the *hypergeometric random variable*, which can take an integer value from 0 to $\min\{p, q\}$. Suppose there are $q$ balls of type 1 and $(r-q)$ balls of type 2 in an urn. Then, $\mathcal{H}(p, q, r)$ is the number of balls of type 1 drawn in $p$ trials, where in each trial, a ball is drawn at random from the urn without replacement. It is known that $\mathbf{E}[\mathcal{H}(p, q, r)] = \frac{pq}{r}$.

**Work Lower Bound.** We first show a lower bound on the work of any algorithm for SWOR-INFWIN, sequential or parallel, by considering the expected change in the sample output after a new minibatch is received.

**Lemma 14** *Any algorithm that solves* SWOR-INFWIN *must have expected work at least*

$$\Omega\left(t + \sum_{i=1}^{t} \min\left\{n_i, \frac{sn_i}{N_i}\right\}\right)$$

*over t timesteps.*

*Proof:* First consider the number of elements that are sampled from each minibatch. If $N_i \leqslant s$, then the entire minibatch is sampled, resulting in a work of $\Omega(n_i)$. Otherwise, the number of elements sampled from the new minibatch $B_i$ is $\mathcal{H}(s, n_i, N_i)$. The expectation is $\mathbf{E}[\mathcal{H}(s, n_i, N_i)] = \frac{s \cdot n_i}{N_i}$, which is a lower bound on the expected cost of processing the minibatch. Next, note that any algorithm must pay $\Omega(1)$ for examining minibatch $B_i$, since in our model the size of the minibatch is not known in advance. If an algorithm does not examine a minibatch, then the size of the minibatch may be as large as $\Omega(N_i)$, causing $\Omega(1)$ elements to be sampled from it. The algorithm needs to pay at least $\Omega(t)$ over $t$ minibatches. Hence, the total expected work of any algorithm for SWOR-INFWIN after $t$ steps must be $\Omega\left(t + \sum_{i=1}^{t} \min\{n_i, \frac{sn_i}{N_i}\}\right)$. ■

**Sequential Algorithm for SWOR-INFWIN.** We present a simple sequential algorithm for SWOR-INFWIN, whose work matches the lower bound from Lemma 14. It uses a subroutine for sampling without replacement from a static array.

**Observation 15 ([AD85, Vit87])** *There is an algorithm for choosing a random sample of size s without replacement from a (static) array of size r using $O(s)$ work.*

The idea in a solution to SWOR-INFWIN, described in Algorithm 5, is as follows. When a minibatch $B_i$ arrives, a random variable $\kappa$ is generated according to the hypergeometric distribution to determine how many of the $s$ samples need to be chosen from $B_i$, rather than minibatches that arrived before $B_i$. The algorithm then chooses a random sample of size $\kappa$ without replacement from $B_i$, and updates the sample $S$ accordingly.

**Algorithm 5:** Work-Optimal Sequential Algorithm for SWOR-INFWIN.

```
Initialization: Sample S ← ∅
  ▷ n_i = |B_i| and N_i = Σ_{j=1}^{i} n_i
if minibatch B_i is received then
    if N_i ⩽ s then
        S ← S ∪ B_i ▷ Store the entire minibatch
    else
        Let κ be a random number drawn from H(s, n_i, N_i)
        Let S_i be a set of κ elements sampled without replacement from B_i
        Replace κ randomly chosen elements in S with S_i.
```

**Lemma 16** *Algorithm 5 is a sequential solution to* SWOR-INFWIN *with work $O\left(t + \sum_{i=1}^{t} \min\{n_i, \frac{sn_i}{N_i}\}\right)$ to process t minibatches $B_1, \ldots, B_t$. This work is optimal given the lower bound from Lemma 14.*

*Proof:* If $N_i \leqslant s$, then the arriving minibatch $B_i$ is added to the sample in its entirety, for a total work of $\Theta(n_i)$. If $N_i > s$, then the algorithm has to pay for (1) generating a random variable according to the hypergeometric distribution, which can be done in $O(1)$ time by an algorithm such as described in [Sta90], (2) selecting a sample of size $\kappa \leqslant n_i$ from $B_i$, which can be done in $O(\kappa)$ time, from Observation 15, and (3) replacing $\kappa$ randomly chosen elements from $S$ – this can be done in $O(\kappa)$ time by choosing $\kappa$ random elements without replacement from $S$ using Observation 15, and overwriting these locations with the new samples. The overall time for processing this batch is $O(1+\kappa) = O(1+\mathcal{H}(s,n_i,N_i))$. The expected time for processing the minibatch is $O(\mathbf{E}[1+\mathcal{H}(s,n_i,N_i)]) = O(1+\frac{s\cdot n_i}{N_i})$. Overall, the cost of processing $B_i$ is $O(1+\min\{n_i, \frac{s\cdot n_i}{N_i}\})$, and the total work of the algorithm is $O\left(t+\sum_{i=1}^{t}\min\{n_i, \frac{sn_i}{N_i}\}\right)$. ■

**Parallel Algorithm for SWOR-Infwin.** Our solution is presented in Algorithm 6. The main idea is as follows: When a minibatch $B_i$ arrives, generate a random variable $\kappa$ according to the hypergeometric distribution to determine how many of the $s$ samples will be chosen from $B_i$, as opposed to prior minibatches. Then, choose a random sample of size $\kappa$ without replacement from $B_i$ and update the sample $S$ accordingly. We leverage Sanders et al. [SLHS+18]'s recent algorithm for parallel sampling without replacement (from static data), restated below in the work-depth model:

**Observation 17 ([SLHS+18])** *There is a parallel algorithm to draw $s$ elements at random without replacement from $N$ elements using $O(s)$ work and $O(\log s)$ depth.*

Our algorithm uses static parallel sampling without replacement in two places: once to sample new elements from the new minibatch, and then again to update the current sample. In more detail, when a minibatch arrives, the algorithm **(i)** chooses $\kappa$, the number of elements to be sampled from $B_i$, in $O(1)$ time; **(ii)** samples $\kappa$ elements without replacement from $B_i$ in parallel; and **(iii)** replaces $\kappa$ randomly chosen elements in $S$ with the new samples using a two-step process, by first choosing the locations in $S$ to be replaced, followed by writing the new samples to the chosen locations. Details appear in Algorithm 6.

**Algorithm 6:** Parallel Algorithm for SWOR-Infwin.

```
Initialization: Sample S ← ∅
if minibatch B_i is received then
    ▷ Recall n_i = |B_i| and N_i = Σ_{j=1}^{i} n_i
    if N_i ⩽ s then Copy B_i into S in parallel
    else
        Let κ be a random number generated by H(s, n_i, N_i)
        S_i ← κ elements sampled without replacement from B_i (Obs. 17)
        R_i ← κ elements sampled without replacement from {1, ..., s} (Obs. 17)
        for j = 1 to κ do Replace S[R_i[j]] ← S_i[j]
```

**Theorem 18** *Algorithm 6 is a work-efficient algorithm for* SWOR-Infwin. *The total work to process $t$ minibatches $B_1, \ldots, B_t$ is $O\left(t+\sum_{i=1}^{t}\min\{n_i, \frac{sn_i}{N_i}\}\right)$ and the parallel depth of the algorithm for processing a single minibatch is $O(\log s)$. This work is optimal up to constant factors, given the lower bound from Lemma 14.*

*Proof:* When a new minibatch $B_i$ arrives, for the case case $N_i \leqslant s$, copying $n_i$ elements from $B_i$ to $S$ can be done in parallel in $O(n_i)$ work and $O(1)$ depth, by organizing array $S$ so that the empty locations in the array are all contiguous, so that the destination for writing an element can be computed in $O(1)$ time.

For the case $N_i > s$, random variable $\kappa$ can be generated in $O(1)$ work. The next two steps of sampling $\kappa$ elements from $B_i$ and from $\{1,\ldots,n\}$ can each be done using $O(\kappa)$ work and $O(\log\kappa)$ depth, using Observation 17. The final for loop of copying data can be performed in $O(\kappa)$ work and $O(1)$ depth. Hence, the expected total work for processing $B_i$ is $1 + \min\{n_i, \frac{sn_i}{N_i}\}$, and the depth is $O(\log\kappa)$. Added up over all the $t$ minibatches, we get our result. Since $\kappa \leqslant s$, the parallel depth is $O(\log s)$. ■

## 4.1 Fixed Length Sliding Window

In the **fixed length sliding window** model with window size $w$, the sample is desired from the $w$ most recent elements, where the window size $w \gg s$ is known in advance. While it is more restrictive than the variable length sliding model, this model is relevant when the aggregation function and their parameters are known in advance, as is often the case in a streaming system with long-lived continuous queries. Algorithms for this model are simpler than those for variable length sliding window.

**Task SWOR-FIXED-WIN:** For each time $i = 1\ldots t$, after observing minibatch $B_i$, maintain a random sample of size $\min\{s, N_i\}$ chosen uniformly without replacement from the $w$ most recent elements in the stream. We first present a lower bound on the work of any algorithm for SWOR-FIXED-WIN.

**Lemma 19** *For any algorithm for* SWOR-FIXED-WIN*, the expected work to process minibatch $i$ is at least* $\Omega\left(1 + \min\left\{s, \frac{s\cdot n_i}{w}\right\}\right)$. *The expected work to process $t$ minibatches is at least* $\Omega\left(t + s\sum_{i=1}^{t}\min\left\{1, \frac{n_i}{w}\right\}\right)$.

*Proof:* After minibatch $B_i$ is received, the random sample must contain $s$ elements randomly chosen from the $w$ most recent elements in $\mathcal{S}_i$. In expectation, the number of elements that will be chosen from $B_i$ is (1) If $n_i \geqslant w$, then $s$, since the entire window is contained within mini-batch $B_i$, and (2) If $n_i < w$, then $s\frac{n_i}{w}$, since $s$ elements are chosen out of a total of $w$ elements. Since the batch size $n_i$ is not known in advance, the work to process the new batch is at least $\Omega(1)$, since the batch has to be examined. If the batch were not examined, it is possible that $n_i \geqslant w$, and the sample will no longer be correct. The work to process $B_i$ is at least $\Omega(1 + \min\left\{s, \frac{s\cdot n_i}{w}\right\})$, and the result for $t$ batches follows. ■

**Algorithm for SWOR-FIXED-WIN:** We first consider a sequential algorithm for SWOR-FIXED-WIN, which follows by combining our algorithm for SWOR-INFWIN with a reduction (due to Braverman et al. [BOZ09]) from the problem of sampling from a fixed-size sliding window to the problem of sampling from an infinite window.

This reduction is based on partitioning the stream into "buckets", each with a fixed starting and ending point. Each bucket has a width of $w$ elements, where $w$ is the size of the sliding window. For $i = 1, 2, \ldots$, the $i$th bucket consists of the stream elements at positions $((i-1)\cdot w + 1)$ till $i \cdot w$. Note that buckets are distinct from minibatches, which could be of arbitrary sizes. A bucket is also different from a query window, whose boundary need not coincide with a bucket boundary. The algorithm maintains a sample without replacement of size $s$ within each bucket $i$, using the infinite window algorithm. Each per-bucket sample is completed when the bucket ends, and a new sample is started for the next bucket. A bucket that overlaps with the current sliding window is called "active". Only active buckets are retained and the rest are discarded. Clearly, there are no more than two active buckets at any time. We call the older of the active buckets as the "old" bucket (OLDBKT) and the more recent active bucket as the "new" bucket (NEWBKT). It is possible that there is no old bucket, if the current window is exactly aligned with the new bucket. We have a sample of size $s$ from OLDBKT,

called OLDSMP, and a sample of size $s$ from NEWBKT, called NEWSMP. Except for corner cases when the new bucket has seen less than $s$ elements, we have that the sizes of both OLDSMP and NEWSMP equal $s$. When a sample is desired from the sliding window, there are two cases. (1) If the window overlaps only with NEWBKT, then return NEWSMP. (2) If the window overlaps partially with OLDBKT and partially with NEWBKT, then the algorithm selects as many elements from OLDBKTthat are not expired yet – say this is of size $s' \leqslant s$. It then selects the remaining $(s - s')$ elements by (uniformly) subsampling from NEWSMP. It is shown in [BOZ09] that this constitutes a random sample chosen without replacement from a sliding window of size $w$.

**Parallel Algorithm for SWOR-FIXED-WIN.** We now show how to implement this reduction in a parallel algorithm. We note that the focus of [BOZ09] was on the space complexity, while we are interested in the time complexity and work of the sampling algorithm. For maintaining the sample without replacement of size $s$ within each bucket, we use one instance of the parallel algorithm for SWOR-INFWIN (Algorithm 6) for each bucket. When queried, the algorithm uses $O(s)$ work to combine the two sampled.

**Lemma 20** *There is a parallel algorithm for* SWOR-FIXED-WIN *whose total work for processing $t$ mini-batches is $O\left(t + s\left(1 + \frac{N_t}{w}\right)\log\frac{w}{s}\right)$. Its parallel depth for processing a single minibatch is $O(\log s)$.*

*Proof:* For the above algorithm, consider the total work involved in processing a single bucket of $w$ elements. If the entire bucket was contained within a minibatch, the total work to process it is $\Theta(s)$ (or possibly even smaller), since no more than $s$ elements are chosen from this bucket (Observation 17). Suppose that the bucket was split across two or more minibatches. The total work to maintain a sample for this bucket depends on the sizes of the minibatches that constitute this bucket – the larger the minibatches, the closer the work is to $O(s)$. But in the worst case, each minibatch could be of size 1, and in this case the total work is the number of times the sample changes over observing $s$ elements – this is $s + \sum_{i=(s+1)}^{w} \frac{s}{i} = O(s + s\log(w/s))$. Over $t$ minibatches, the number of different buckets is $N_t/w$, which leads to a total work of $O(sN_t(\log(w/s))/w)$. We also note that $O(1)$ time is required for each minibatch. Further, in the case $N_t < w$, when a single bucket has not completed yet, we still need $O(s\log w)$ work. The parallel depth follows because for each minibatch that arrives, it is needed to determine which of Cases (1)-(3) apply here, which can be done in $O(1)$ steps, followed by updating OLDBKT and NEWBKT. The depth of the update process for these follows from Observation 17 and Theorem 18. ■

# 5 Parallel Sampling with Replacement

We now consider SWR-INFWIN, sampling with replacement. Intuitively, parallelizing sampling with replacement is easier than without replacement because random samples can be independently drawn without checking for distinctness.

**Observation 21** *There is an $O(s)$-work $O(1)$-depth parallel algorithm for computing a sample of size $s$ with replacement from an input array of size $n$.*

The above is easy to see: Each sample element can be independently chosen by selecting a random integer in the interval $[1, n]$ and choosing the corresponding element from the input. We will use the binomial random variable. For integer $\alpha > 0$ and $0 < \beta \leqslant 1$, let $\mathcal{B}(\alpha, \beta)$ be the the number of successes in $\alpha$ trials, where each trial has probability $\beta$ of success.

**SWR-INFWIN: Sampling with replacement, infinite window.** A simple algorithm is to run $s$ independent parallel copies of a single element stream sampling algorithm, which is clearly correct. When minibatch

$B_i$ is received, each single element sampling algorithm decides whether or not to replace its sample, with probability $n_i/N_i$, which can be done in $O(1)$ time. The algorithm has $O(s)$ work per minibatch, and parallel depth $O(1)$. However, we can do better than this as follows.

Suppose the current samples after observing $i$ minibatches are stored in an array $S[1,2,\ldots,s]$. For each sample, the probability of it being replaced after the minibatch arrives is $p_i = n_i/N_i$. Instead of testing each sample separately to see if needs to be replaced, the algorithm carries out the following steps:

(A) Generate a random number $s' \leqslant s$ according to $\mathcal{B}(s, p_i)$ to determine how many of the samples need to be replaced.
(B) Sample $L$, a set of $s' \leqslant s$ elements *without replacement* from the set $\{1,2,\ldots,s\}$, using a parallel sampling algorithm (Observation 17).
(C) For each location $\ell \in L$, sample $S[\ell]$ is overwritten with a randomly chosen element from $B_i$.

Steps (A) and (B) in the above procedure can be shown to generate the same distribution of locations as iterating through each location in $S$, and separately determining whether the sample needs to be replaced. Step (A) can be performed in $O(1)$ work, while Step (B) can be performed in $O(s')$ work with $O(\log s') = O(\log s)$ parallel depth. Since $\mathbf{E}[s'] = sp_i = sn_i/N_i$, the expected work of this step is $O(sn_i/N_i)$. Step (C) can be performed in $O(s')$ work and $O(1)$ depth. Below are the overall properties of the algorithm:

**Theorem 22** *There is a parallel algorithm for* SWR-INFWIN *such that for a target sample size $s$, the total work to process minibatches $B_1,\ldots,B_t$ is $O(t+\sum_{i=1}^{t} sn_i/N_i)$, and the depth for processing any one minibatch $B_i$ is $O(\log s)$. This work is optimal, up to constant factors.*

This work bound is optimal—the expected number of elements in the sample that change due to a new minibatch is $(sn_i/N_i)$. The theorem means the minibatches, at least the initial few, should be large. To see why: if they are of size 1, the first element would require $\Theta(s)$ work, because every sample needs to be updated! Similarly, the first $s/2$ elements each requires $O(s)$ work, totaling $O(s^2)$ work for the initial $\Theta(s)$ elements. With minibatches sized $\Omega(s)$, the total work decreases significantly. Importantly, this is not an artifact of our algorithm—any algorithm for SWR-INFWIN needs this cost when minibatches are small.

# 6 Conclusion

We presented low-depth, work-efficient parallel algorithms for the fundamental data streaming problem of streaming sampling. Both the sliding-window and infinite-window cases were addressed. Interesting directions for future work include the parallelization of other types of streaming sampling problems, such as weighted sampling and stratified sampling.

# References


[AD85] J. H. Ahrens and U. Dieter. Sequential random sampling. *ACM Trans. Math. Softw.*, 11(2):157–169, June 1985.

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.

[BH06] Paul G. Brown and Peter J. Haas. Techniques for warehousing of sample data. In *ICDE*, page 6, 2006.

[BM04] Guy E. Blelloch and Bruce M. Maggs. Chapter 10: Parallel algorithms. In *The Computer Science and Engineering Handbook*. Chapman and Hall/CRC, 2nd edition, 2004.

[BOZ09] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. In *PODS*, pages 147–156, 2009.

[CMYZ12] Graham Cormode, S. Muthukrishnan, Ke Yi, and Qin Zhang. Continuous sampling from distributed streams. *Journal of the ACM*, 59(2):10:1–10:25, 2012.

[Cor13] Graham Cormode. The continuous distributed monitoring model. *SIGMOD Record*, 42(1):5–14, 2013.

[CTW16] Y. Chung, S. Tirthapura, and D. P. Woodruff. A simple message-optimal algorithm for random sampling from a distributed stream. *TKDE*, 28(6):1356–1368, 2016.

[DAAE09] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *PVLDB*, 2(1):217–228, 2009.

[DP09] Devdatt Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

[ES06] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181 – 185, 2006.

[GL08] Rainer Gemulla and Wolfgang Lehner. Sampling time-based sliding windows in bounded space. In *SIGMOD*, pages 379–392, 2008.

[GSSB15] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. A top-down parallel semisort. In *SPAA*, pages 24–34, 2015.

[GT01] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA*, pages 281–291, 2001.

[JS15] Theodore Johnson and Vladislav Shkapenyuk. Data stream warehousing in tidalrace. In *CIDR*, 2015.

[Rei85] John H. Reif. An optimal parallel algorithm for integer sorting. In *FOCS*, pages 496–504, 1985.

[Ros09] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, 10th edition, 2009.

[SLHS+18] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient parallel random sampling - vectorized, cache-efficient, and online. *ACM Trans. Math. Softw.*, 44(3):29:1–29:14, 2018.

[Sta90] E. Stadlober. The ratio of uniforms approach for generating discrete random variates. *J. Comput. Appl. Math.*, 31(1):181–189, 1990.

[TPT13] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel triangle counting in massive streaming graphs. In *CIKM*, pages 781–786, 2013.

[TTW14] Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Parallel streaming frequency-based aggregates. In *SPAA*, pages 236–245, 2014.

[Vit85] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[Vit87] J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1):58–67, 1987.

[XTB08] B. Xu, S. Tirthapura, and C. Busch. Sketching asynchronous data streams over sliding windows. *Distributed Computing*, 20(5):359–374, 2008.

[ZDL+13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.