External Memory Stream Sampling

Xiaocheng Hu Miao Qiao Yufei Tao

CUHK
Hong Kong

ABSTRACT

This paper aims to understand the I/O-complexity of maintaining a *big sample set*—whose size exceeds the internal memory's capacity—on a data stream. We study this topic in a new computation model, named the *external memory stream* (EMS) *model*, that naturally extends the standard external memory model to stream environments. A suite of EMS-indigenous techniques are presented to prove matching lower and upper bounds for with-replacement (WR) and without-replacement (WoR) sampling on append-only and time-based sliding window streams, respectively. Our results imply that, compared to RAM, the EMS model is perhaps a more suitable computation model for studying stream sampling, because the new model separates different problems by their hardness in ways that could not be observed in RAM.

Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems

Keywords

Stream; Sampling; I/O-Efficient Algorithms; Lower Bound

1. INTRODUCTION

Uniform sampling on data streams is a fundamental topic that has been extensively studied, but under the assumption that the sample set fits in (internal) memory [3, 4, 8, 19]. In this work, we do away with the assumption, and aim to understand the I/O complexity of maintaining *big* sample sets.

1.1 Motivation of Big Sampling

Big sampling—acquiring a sample set whose size exceeds the memory's capacity—is hardly a new concept. Jermaine et al. [12] were the first to design algorithms to do so by taking disk accesses into account; and their work has triggered a line of research [7, 12, 15, 16, 18] on this topic. The importance of big sampling is reflected in the fact that, the sample size must grow rapidly with the cardinality of the "ground" input set to allow high accuracy in many analytic tasks. Next, we explain this in two areas where sampling has been widely applied: subset-size estimation, and designing approximation algorithms.

Estimation with Bounded Relative Errors. Consider the following basic estimation problem. Let S be a set of n elements. Given a *predicate* which can be any function $f : S \to \{0, 1\}$, we want to estimate the number k of elements $e \in S$ with $f(e) = 1$. Clearly, the entire S must be stored if a precise k is demanded for an arbitrary predicate.

Suppose that it suffices to derive an estimate $\hat{k}$ such that, with probability at least $1-\delta$, the relative error $|k-\hat{k}|/k$ is at most ϵ. This problem can be solved by taking s samples *with replacement* (namely, each sample is uniformly distributed in S independently), and obtaining $\hat{k}$ by first counting and then scaling up the number of samples e with $f(e) = 1$. How large should s be? A standard application of Chernoff bounds shows $s = O(\frac{n}{\epsilon^2 k} \log \frac{1}{\delta})$.

To strike a balance between the space usage and the smallest k supported, a threshold Δ is set such that the guarantee holds for all $k \geq \Delta$. The number of samples required is thus $O(\frac{n}{\epsilon^2 \Delta} \log \frac{1}{\delta})$. Δ is typically in relation to n, e.g., $\Delta = n^{0.99}$ (as n grows, a predicate needs to retrieve more elements to be protected by the guarantee), making $O(\frac{n^{0.01}}{\epsilon^2} \log \frac{1}{\delta})$ samples necessary, namely, *polynomial* to n. Note that here ϵ is not necessarily a constant: in some applications the number of samples may be fixed, in which case the value of ϵ (i.e., guarantee-able error bound) needs to increase with n.

Note that the exemplified Δ was deliberately chosen to be very large; in reality, Δ is often much smaller, but this can only *increase* the sample size.

Approximation Algorithms. Uniform sampling is a common tool in designing algorithms for finding approximate answers to problems whose exact results are expensive to compute. For example, in *triangle counting*, we are given a graph $G = (V, E)$, and want to count the number of 3-vertex cliques in G. In the streaming version of the problem, the edges of G arrive continuously; the goal is to maintain an estimate of the number of triangles that has a relative error at most ϵ with a probability at least $1 - \delta$. Currently the best algorithm [17] requires space

PODS'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

Http://dx.doi.org/10.1145/2745754.2745757.

$O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\cdot\frac{|E|d_{max}}{t})$, where d_{max} is the maximum degree of a vertex in G, and t is the number of triangles in G. The algorithm at its core samples $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\cdot\frac{|E|\cdot d_{max}}{t})$ edges of E (in a strategic manner that uses uniform sampling as a black box). The number of samples can be very large, when either d_{max} is large or t is small.

1.2 The External Memory Stream Model

Big sampling perhaps makes the best sense on big data, which are characterized by "3V": (high) volume, velocity, and variety. The first two V's imply that big-data sampling is an external memory problem, and simultaneously, also a data stream problem.

We propose an extension of the standard *external memory* (EM) model [1] to the stream scenario. A machine is equipped with internal memory of M words, and a disk of an unbounded size which has been formatted into *blocks* of size B words. It holds that $M \geq 2B$. The data *stream* is defined to be an unbounded sequence of elements, with each element fitting in a word. An algorithm is allowed the following operations. First, it may perform an I/O to exchange a block of data between the memory and the disk. Second, it may perform any CPU calculation on the data inside the memory. Third, it may perform a PULL operation to fetch the next element from the stream, which will be placed inside the memory, overwriting a word there. This element is said to have *arrived*, and disappears forever from the stream.

We measure the *time* of an algorithm as the number of I/Os it performs, and its *space* as the number of disk blocks it occupies. CPU calculation and the PULL operation are for free. We will refer to the above as the *external memory stream* (EMS) *model*.

Allowing PULL to be I/O-free is consistent with the fact that, when fetched from a network socket, a stream element is always saved into memory in the existing operating systems. In fact, the "directly-into-memory" feature is not really new, and finds its counterpart in the conventional EM model as well. Specifically, one may compare PULL to an "insertion" to an EM structure. Such insertions are (implicitly) assumed to be given in memory; this assumption is crucial in, for example, the analysis of online *buffer trees* [2, 20].

What differentiates the EMS model from the EM model is, in our opinion, the amount of space consumption. Suppose that currently N stream elements constitute the input to a computation problem. We would typically be interested in algorithms that consume *far less* than N/B blocks, but more than M words (to avoid degenerating into traditional stream algorithms in RAM).

1.3 Problem Definitions and Previous Results

Let P be a ground set of N elements. There are two standard definitions of sample set:

- A *without-replacement* (WoR) *sample set* with size R can be any of the $\binom{N}{R}$ size-R subsets of P with the same probability.[1]
- A *with-replacement* (WR) *sample set* with size R is a sequence of R elements, each of which is taken uniformly at random from P independently.

Note that the element ordering is irrelevant for a WoR sample set, but matters for a WR sample set.

We revisit, in the EMS model, two problems that have received considerable attention in RAM. The first one is:

Problem 1: Full Stream Sampling. Maintain a WoR or WR sample set of size R on all the elements that have arrived.

In RAM, the WoR version of Problem 1 can be optimally solved with *reservoir sampling* [19], which uses $O(R)$ space, processes each incoming element in $O(1)$ time, and reports a sample set in $O(R)$ time. As a folklore, a size-R WoR sample set can be converted to a WR sample set of the same size in $O(R)$ time. Therefore, all the performance guarantees of reservoir sampling carry over to the WR version of Problem 1 as well.

In the scenario where the sample set does not fit in memory, previous work on Problem 1 did not explicitly consider the EMS model. Nevertheless, some of the solutions developed can be adapted to work in this model with good guarantees. The *geometric file* of Jermaine et al. [12, 18] processes a stream of N elements with total cost of $O(\frac{R^2}{MB}\log\frac{N}{R})$ expected I/Os, and consumes $O(R/B)$ space at all times, such that a sample set of size R can be output at any moment in $O(R/B)$ I/Os. Gemulla and Lehner [7] presented an improved algorithm that (when adapted to EMS) has the same space and sample reporting cost as the geometric file, but can process N stream elements with $O(\frac{R}{B}\log\frac{N}{R})$ expected I/Os in total.

There have been no explicit studies on maintaining massive WR sample sets. However, the folklore RAM algorithm for size-R WoR-to-WR conversion can be implemented in $perm(R) = \Theta(\min\{R, \frac{R}{B}\log_{M/B}\frac{R}{B}\})$ I/Os in EM.[2] Hence, the algorithm of [7] can also output a size-R WR sample set in $perm(R)$ I/Os.

An interesting question then arises: *is it possible to carry out a size-R WoR-to-WR conversion in $o(perm(R))$ I/Os?* Recently, a negative answer has been provided in [10] for the special case where R is as large as the size of the ground set. The argument of [10], unfortunately, does not extend to general values of R.

Problem 2: Time-Based Sliding Window Sampling. Each element e from the stream now carries a real-valued *timestamp* t_e, which is non-descending, namely, $t_e \geq t_{e'}$ where e' is the element arriving right before e. Define a *sliding window* as the set of elements whose timestamps fall in $[t_{now} - \tau, t_{now}]$, where $\tau > 0$ is a fixed real-valued parameter and t_{now} is the timestamp of the last arrived element. The objective is to maintain a WoR or WR sample set with size R of the current sliding window.

Let n be the length of the current sliding window. Note that n may vary with time from 1 to any arbitrarily

[1] Specially, we regard a WoR sample set of size $R > N$ to be P itself.

[2] $perm(R)$ is the number of I/Os needed to permute R elements in EM [1].

problem	model	cost of handling N elements	space	WoR sample reporting	WR sample reporting	source	remark
Prob. 1	RAM	$O(N)$	$O(R)$	$O(R)$	$O(R)$	[19]	
Prob. 1	EMS	$O(\frac{R}{B}\log\frac{N}{R})^\dagger$	$O(R/B)$	$O(R/B)$	$O(perm(R))$	[7]	
Prob. 1	EMS	$\Omega(\frac{R}{B}\log\frac{N}{R})^\dagger$	trivially $\Omega(R/B)$	trivially $\Omega(R/B)$	$\Omega(perm(R))^\ddagger$	**new**	**optimal**
Prob. 2	RAM	$O(N)$	$O(R\log n)$	$O(R)$	$O(R)$	[4]	n = window length
Prob. 2	RAM/EMS		$\Omega(R\log\frac{n}{R})$ elements			[8]	
Prob. 2	RAM	$O(N)$	$O(R\log\frac{n}{R})$	$O(R)$	$O(R)$	**new**	**optimal**
Prob. 2	EMS	$\Theta(N/B)$	$O(\frac{R}{B}\log\frac{n}{R})$	$\Theta(R/B)$	$\Theta(perm(R))^\ddagger$	**new**	**optimal**

†Expected. ‡The lower bound applies to the element-processing cost in the 3rd column.

Table 1: Summary of our and previous results

large integer. In RAM, Braverman et al. [4] described an algorithm that uses $O(R\log n)$ space, processes each element in $O(1)$ amortized time, and reports a WoR (hence, also WR) sample set in $O(R)$ time. In [8], Gemulla and Lehner proved that any algorithm solving Problem 2 must store $\Omega(R\log\frac{n}{R})$ elements.[3] In other words, there is a tiny gap between the lower bound and the space of [4].

Problem 2 has not been investigated in the scenario where the sample set is disk resident. Currently the only approach to solve the problem in the EMS model is to run the RAM algorithm of [4] by treating the disk as virtual memory, but the approach obviously suffers from severe I/O penalty.

1.4 Our Results

We resolve the I/O complexities of both problems in the EMS model. For Problem 1, our contributions are on lower bounds:

THEOREM 1. *In the EMS model, when $N \geq R \geq cM$ with c being a sufficiently large constant,*

1. *any algorithm solving the WoR or WR version of Problem 1 must perform $\Omega(\frac{R}{B}\log\frac{N}{R})$ expected I/Os to process N stream elements.*
2. *any algorithm that uses $o(perm(R)\cdot\log\frac{N}{R})$ expected I/Os to process N stream elements must incur $\Omega(perm(R))$ I/Os to return a WR sample set.*

Therefore, the algorithm of [7], as well as its WR extension described in Section 1.3, is already optimal. Note that the theorem officially separates WoR sampling from WR sampling—conditioned on spending $O(\frac{R}{B}\log\frac{N}{R})$ I/Os processing N stream elements, a WoR sample set can be reported in $O(R/B)$ I/Os, whereas a WR one requires $\Omega(perm(R))$ I/Os to report. This, interestingly, implies that the folklore WoR-to-WR conversion algorithm is already the best in EM:

COROLLARY 1. *In the EM model, when $R \geq cM$ with c being a sufficiently large constant, any algorithm converting a size-R WoR sample set of a ground set P into a size-R WR sample set of P must incur $\Omega(perm(R))$ I/Os.*

On Problem 2, we attack upper and lower bounds simultaneously, and conquer both by designing an optimal EMS algorithm, as shown in next two theorems:

THEOREM 2. *For Problem 2, there is an EMS algorithm that performs $O(N/B)$ I/Os to process N stream elements, uses $O(\frac{R}{B}\log\frac{n}{R})$ space (where n is the length of the current sliding window), and outputs at any moment a WoR sample set in $O(R/B)$ I/Os, and a WR sample set in $O(perm(R))$ I/Os.*

THEOREM 3. *When $N \geq cR$ and $R \geq cM$ with c being a sufficiently large constant,*

1. *any algorithm solving the WoR or WR version of Problem 2 must perform $\Omega(N/B)$ expected I/Os to process N stream elements.*
2. *any algorithm that uses $o(perm(R)\cdot(N/R))$ expected I/Os to process N stream elements must incur $\Omega(perm(R))$ expected I/Os to return a WR sample set.*

The above statements are true regardless of τ.

Note that the space optimality of Theorem 2 follows from the space lower bound $\Omega(R\log\frac{n}{R})$ of [8].

Our algorithm for Problem 2 adopts novel ideas beyond the solution of Braverman et al. [4]. Perhaps the fastest (and yet convincing) way to illustrate this is to point out that, setting $B = 1$ and M to an appropriate constant, our algorithm also works in RAM, where we improve the result of Braverman et al. [4]:

COROLLARY 2. *For Problem 2, in the RAM model, there is an algorithm that uses $O(R\log\frac{n}{R})$ space (where n is the length of the current sliding window), processes each stream element in $O(1)$ amortized time, and outputs at any moment a WoR sample set in $O(R)$ time.*

This is the first RAM algorithm for Problem 2 that is optimal for *all* values of n and R. Table 1 gives a quick summary of our and previous results.

EMS vs. RAM. By comparing the results of Problems 1 and 2, one can see the differences in their I/O complexities. In particular, while Problem 2 requires $\Theta(1/B)$ amortized I/Os per element, it is possible to lower the corresponding cost to $O(\frac{1}{B}\frac{R\log N}{N})$ for Problem 1. Such a separation is absent in RAM, where an algorithm obviously needs to pay at least constant time just to look at an element. This has two implications. First, the $O(1)$-time "optimality" in RAM (for element processing) hardly touches the essence of the problems. Second, efficiency is a far more important issue in the EMS model than in RAM (where attention focuses primarily on space usage and seldom on efficiency).

[3]Gemulla and Lehner claimed $\Omega(R\log N)$ in [8], but a close look at their proof reveals that their lower bound is actually $\Omega(R\log\frac{n}{R})$.

Note also that the above separation owes to the I/O-free PULL operation in the EMS model—if an algorithm was forced to read the stream from the disk, the element-processing cost would be vacuously dominated by $\Omega(N/B)$ for both problems.

Remarks. The selection of Problems 1 and 2 is a careful one. Problem 1 (due to its relative simplicity) is the best for demonstrating our lower bound techniques. Problem 2, on the other hand, demands new ideas to design an efficient EMS algorithm, even given its state-of-the-art solution in RAM. In fact, the methods developed in this paper can be applied to settle other sampling problems optimally in the EMS model as well, e.g., *sequence-based sliding window sampling* [4].

2. PRELIMINARIES

Let us start with three basic algorithms that will be needed frequently throughout the paper:

2.1 Offline WoR Sampling in EM

In this problem, we are given a *static* ground set P in an array (hence, $|P|$ is known). The goal is to obtain a size-R WoR sample set of P.

Here is a simple algorithm [6, 13] performing $O(|P|/B)$ I/Os. For the j-th ($1 \leq j \leq |P|$) element of P, add it to the sample set with probability $(R-x)/(|P|-j+1)$, where x is the number of samples already taken from the first $j-1$ elements of P.

2.2 WoR-to-WR Conversion in EM

In this problem, we are given a size-R WoR sample set S of a ground set P. The goal is to convert S into a size-R WR sample set of P.

Next, we show how to do so in $O(perm(R))$ I/Os, assuming $n = |P|$ and $R \leq n$.[4] Our algorithm executes in three steps:

- Generate an array A of size R. First, set $A[1] = 1$ and $J = 1$. Then, in ascending order of $i \in [2, R]$, decide $A[i]$ by choosing uniformly at random an integer $z \in [1, n]$. If $z \leq J$, set $A[i] = z$; otherwise, set $A[i] = J+1$ and increase J by 1.
- Take a size-J WoR sample set T of S (using the algorithm in Section 2.1). Randomly permute the elements of T in $O(perm(R))$ I/Os [9].
- Create an array Q of size R where $Q[j]$ ($1 \leq j \leq R$) is the $A[j]$-th element of T. It can be done in $O(perm(R))$ I/Os.

Q is returned as the final WR sample set. See appendix for a correctness proof.

2.3 Full Stream Sampling in EMS

In this subsection, we review the algorithm of [7] for solving Problem 1, which can be regarded as an implementation of reservoir sampling in the EMS model.

[4]If $R > n$, then $S = P$, in which case the problem is straightforward.

Handling Incoming Elements. The first R stream elements constitute the initial WoR sample set S. Create an empty linked list L at this point.

Given the i-th ($i \geq R+1$) stream element e, we add e to L with probability R/i, and discard e otherwise. When $|L| = R$, update the WoR sample set S by performing a *merge* of L and S (to be explained shortly). The algorithm then empties L and continues as described above.

A merge of L and S is carried out as follows:

- Perform a "clean-up" to possibly shrink L by inspecting its elements in reverse arrival order. At the beginning, $x = 0$. For the j-th ($1 \leq j \leq |L|$) most-recently arrived element e of L, toss a coin with head probability x/R. If it heads, remove e from L; otherwise, retain e in L, and increase x by 1.
- Take a WoR sample set S' of size $R - |L|$ from S in $O(R/B)$ I/Os with offline sampling.
- Reset $S = S' \cup L$.

The cost of a merge is $O(R/B)$ I/Os.

The algorithm uses $O(R/B)$ space at all times. It processes N elements with $O(N/B)$ I/Os in the *worst* case. The expected cost, on the other hand, is much lower. To see this, note that after N elements there have been in expectation $\sum_{i=R+1}^{N}(R/i) = O(R\log(N/R))$ insertions into L. Therefore, the algorithm has launched in expectation $O(\log(N/R))$ merges, which perform in total $O(\frac{R}{B}\log\frac{N}{R})$ expected I/Os.

WoR Sample Set Any Time. To report a WoR sample set before $|L|$ reaches R, simply perform a merge of S and L right away, and return the resulting S. The cost is $O(R/B)$.

3. HARDNESS OF PERMUTATION IN EMS

In this section, we will formulate and study a problem called (N, R)-*permutation*. Not only does this problem explain the hardness of WR sampling in the EMS model (as shown in the next section), but also it sheds light on some subtle but crucial differences between the EM and EMS models from a technical point of view.

The problem is defined as follows. The input is a stream of N elements. Denote by P the set of those elements. Given the value of N and an integer $R \leq N$, an algorithm should

- either *succeed* by writing to the disk a random R-permutation[5] of P, namely, any of the $\frac{N!}{(N-R)!}$ R-permutations is output with the same chance,
- or declare *failure*.

The algorithm must succeed with at least a constant probability.

The problem admits a solution that never fails, and terminates in $O(perm(R))$ I/Os. We can first take a size-R WoR sample set S of P. Since N is known, this is in fact offline sampling, and can be completed in $O(R/B)$ I/Os (see

[5]An R-permutation is an ordered list of R distinct elements in P.

Section 2.1). After that, we output a random permutation of S in $O(perm(R))$ I/Os.

The rest of the subsection will prove that it is impossible to do any better, even by failing:

LEMMA 1. *When $R \geq 2M$, any algorithm solving the (N, R)-permutation problem must incur $\Omega(perm(R))$ I/Os in expectation.*

EMS Subtlety. Permutation is one of the most fundamental problems in EM, and also one of the best understood, with lower bounds already established in various contexts [1, 11, 14]. Unfortunately, all the techniques behind those lower bounds can be used to prove only bounds far worse than the one in Lemma 1 in the EMS model. This, interestingly, is not due to the looseness of those techniques, but rather, due to an inherent difference between the EM and EMS models.

In general, a permutation lower bound in EM is proved by observing that an I/O cannot generate too many new permutations, which, in turn, relies on the fact that in EM *no new elements can enter memory between two I/Os*. This is no longer true in the EMS model: new elements may be added to the memory by the PULL operation, even though no I/O is performed. As a consequence, an I/O may potentially create a huge number of new permutations, thus significantly weakening the power of the existing arguments.

In what follows, we will present an EMS-indigenous technique to derive Lemma 1.

Producing One *R*-Permutation. Let us first look at a different problem in the EMS model which we call *one-R-permutation*. The input stream is a sequence of R pairs of the form (e_i, p_i), where e_i is an element drawn from a certain domain, and p_i is an integer in $[1, R]$. It is required that $p_1, p_2, ..., p_R$ are distinct. The output is a permutation π of $e_1, e_2..., e_R$ in the disk where e_i is placed at the p_i-th position of π, for each $i \in [1, R]$.

This problem is in fact a disguise of the classic problem of permuting R elements in EM. Notice that $\Omega(R/B)$ is a clearly a lower bound for one-R-permutation. Hence, one can reduce the EM permutation problem to one-R-permutation, by simulating a stream with a scan of the R elements in the input. The opposite reduction is also straightforward: in the EMS model, one can first spend $O(R/B)$ I/Os storing all the incoming R elements, and then, solve one-R-permutation as a permutation problem in EM. It thus follows that the *worst-case* I/O complexity is $\Theta(perm(R))$ for one-R-permutation.

However, it is not the worst case that will interest us in the subsequent discussion; instead, we will be concerned with *every instance* of the problem. Specifically, for every possible sequence $(p_1, ..., p_R)$, we will need the *smallest* I/O cost of all the possible strategies for producing the target permutation π. For example, when $p_1, ..., p_R$ happen to be in ascending order, then an *optimal strategy* would spend only $\lceil R/B \rceil$ I/Os. Interestingly, we do not need to design the optimal strategy for each $(p_1, ..., p_R)$; it suffices to know that it definitely *exists*.

Proof of Lemma 1. The crux of our proof is to show that, given an algorithm $\mathcal{A}$ solving the (N, R)-permutation problem in H expected I/Os, we can design an algorithm $\mathcal{A}'$ in the EM model to produce a random permutation of R elements in $O(H)$ expected I/Os. The latter problem has a lower bound of $\Omega(perm(R))$ expected I/Os[6]. It will thus follow that $H = \Omega(perm(R))$.

From now on, we will fix the arrival order of the elements in P. Denote by c the success probability of $\mathcal{A}$. Let S be an arbitrary size-R subset of P. We denote by $cost(\mathcal{A} \mid S)$ the expected I/O cost of $\mathcal{A}$, under the event that the algorithm succeeds by outputting an R-permutation with elements only from S. The probability of the event is $\mathbf{Pr}[S] = c/\binom{N}{R}$ for any S. We have:

$$H \geq \sum_{\forall S} cost(\mathcal{A} \mid S) \cdot \mathbf{Pr}[S].$$

Let $S^\star$ be the S with the smallest $cost(\mathcal{A} \mid S)$. Thus:

$$\begin{aligned} H &\geq \sum_{\forall S} cost(\mathcal{A} \mid S^\star) \cdot \mathbf{Pr}[S] \\ &= cost(\mathcal{A} \mid S^\star) \sum_{\forall S} \mathbf{Pr}[S] \\ &= c \cdot cost(\mathcal{A} \mid S^\star). \end{aligned} \quad (1)$$

Henceforth, we will view $S^\star$ as a sequence where its elements are arranged by arrival order in the stream.

Next we design an algorithm $\mathcal{A}^\star$ which may fail with probability $1 - c$, but when it does not, it *always* produces a random permutation of $S^\star$. Furthermore, the expected cost of $\mathcal{A}^\star$ is at most H. For this purpose, we take the view that a randomized algorithm has free access to a sequence of random bits, such that the algorithm's execution is deterministic once all those bits are fixed. For every such sequence Σ of fixed bits, we design the (deterministic) behavior of $\mathcal{A}^\star$ conditioned on Σ:

- If $\mathcal{A}$ fails on Σ, we instruct $\mathcal{A}^\star$ to fail immediately without performing any I/O.
- If the output of $\mathcal{A}$ (given Σ) is a permutation $\pi^\star$ of $S^\star$, we instruct $\mathcal{A}^\star$ to take an optimal strategy to solve the one-R-permutation instance that aims to produce $\pi^\star$ from the stream sequence $S^\star$ (recall how we view $S^\star$ as a sequence). Such a strategy always exists, as explained before.
- The remaining case is that the output of $\mathcal{A}$ (given Σ) is a permutation π of an element set $S \neq S^\star$. Let us regard S as a sequence of its elements sorted by arrival order. We resort to the following one-one mapping from the elements of S to those of $S^\star$: for each $i \in [1, R]$, the i-th element of S maps to the i-th element of $S^\star$. The mapping converts π into a permutation $\pi^\star$ of $S^\star$. We instruct $\mathcal{A}^\star$ to follow an optimal strategy to solve the one-R-permutation instance that aims to produce $\pi^\star$ from the stream sequence $S^\star$.

We claim that the algorithm $\mathcal{A}^\star$ thus designed serves our purposes. First, it is clear that $\mathcal{A}^\star$ succeeds with probability c, and when it does, it always outputs a permutation of $S^\star$. Second, $\mathcal{A}^\star$ (if succeeds) returns a random permutation of $S^\star$ because (i) $\mathcal{A}$ outputs a random R-permutation of P and

[6]Proof: Combine Yao's Minimax theorem and the permutation lower bound of Aggarwal and Vitter [1].

(ii) the same number of R-permutations of P are mapped to each permutation of $S^\star$ output by $\mathcal{A}^\star$.

It remains to analyze the expected cost $H^\star$ of $\mathcal{A}^\star$. By definition:

$$H^\star = \sum_{\forall \text{ permutation } \pi^\star \text{ of } S^\star} optcost(\pi^\star) \cdot \frac{c}{R!} \quad (2)$$

where $optcost(\pi^\star)$ is the cost of an optimal strategy solving the one-R-permutation problem that produces $\pi^\star$ from the stream sequence $S^\star$. Denote by $cost(\mathcal{A} \mid \pi^\star)$ the expected cost of $\mathcal{A}$ under the condition that $\mathcal{A}$ outputs $\pi^\star$. It holds by definition of $optcost(\pi^\star)$ that $optcost(\pi^\star) \leq cost(\mathcal{A} \mid \pi^\star)$. Therefore:

$$\begin{aligned} (2) &\leq c \sum_{\forall \text{ permutation } \pi^\star \text{ of } S^\star} cost(\mathcal{A} \mid \pi^\star)\frac{1}{R!} \\ &= c \cdot cost(\mathcal{A} \mid S^\star) \\ (\text{by (1)}) &\leq H. \end{aligned}$$

We are now ready to design the promised algorithm $\mathcal{A}'$ in the EM model which, given a set X of R elements, outputs a random permutation of X. We simply create a (virtual) data stream of length N where the elements of X are placed at the same positions as the elements of $S^\star$ in the input stream of $\mathcal{A}^\star$. Since the contents of the elements outside X are irrelevant, we can generate such a stream by reading X in $O(R/B)$ I/Os. Then, $\mathcal{A}'$ repetitively runs $\mathcal{A}^\star$ on that stream until it succeeds (when it does, the output is guaranteed to be a random permutation of X). The total expected cost of $\mathcal{A}'$ is $O(R/B + H^\star) = O(H)$, where the equality used the fact that $H = \Omega(R/B)$ when $R \geq 2M$. We thus complete the proof of Lemma 1.

4. HARDNESS OF FULL STREAM SAMPLING

In this section, we will explain why it is impossible to do any better than the algorithm of [7] (see Section 2.3), by presenting a proof of Theorem 1. As required by the theorem, we assume that $N \geq R \geq cM$ with c being a sufficiently large constant.

We say that the i-th element of the stream has *sequence number* i. Divide the stream into *epochs* such that epoch 1 includes the first R elements of the stream, and epoch $j \geq 2$ covers elements with sequence numbers in

$$[1 + 2^{j-2}R, 2^{j-1}R].$$

Note that, at the end of each epoch $j \geq 2$, epoch j accounts for exactly half of the elements that have arrived.

Element Processing. We will first prove Statement 1 of the theorem. We will argue that any algorithm $\mathcal{A}$ must perform $\Omega(R/B)$ expected I/Os in each of the $\Theta(\log \frac{N}{R})$ epochs, which brings the total cost to $\Omega(\frac{R}{B} \log \frac{N}{R})$.

Consider first WoR sampling, for which we choose $c = 3$. For the first epoch, since $\mathcal{A}$ must remember all the first R elements, it needs $\Omega((R - M)/B) = \Omega(R/B)$ I/Os to write at least $R - M$ elements to the disk. For each epoch $j \geq 2$, if a sample request arises at the end of the epoch, with half probability, $\mathcal{A}$ must return at least $R/2$ samples from epoch j (recall that epoch j covers half of the already-arrived elements). Therefore, with half probability, $\mathcal{A}$ must have written at least $R/2 - M = \Omega(R)$ of samples from epoch j to the disk, necessitating $\Omega(R/B)$ I/Os.

The above argument can be extended to WR sampling, for which we choose $c = 9$. For each epoch $j \geq 3$, imagine that we, at the end of the epoch, request $\mathcal{A}$ to return a size-R sample set stored in an array Q. We observe:

LEMMA 2. *For each $z \in [1, R]$, $Q[z]$ has at least $1/4$ probability to satisfy two conditions simultaneously: (i) it comes from epoch j, and (ii) $Q[z]$ is unique in Q (i.e., there does not exist $z' \neq z$ with $Q[z'] = Q[z]$).*

PROOF. First, $Q[z]$ clearly has half probability to come from epoch j. Conditioned on $Q[z]$ originating from epoch j, next we show that it is unique in Q with at least half probability. This will prove the correctness of the lemma.

Epoch $j \geq 3$ has at least $2R$ elements by definition. As the elements of Q are independent, $Q[z]$ equals one of the other $R-1$ elements with probability at most $(R-1)/2R < 1/2$. □

COROLLARY 3. *With at least $1/7$ probability, Q contains at least $R/8$ unique elements coming from epoch j.*

PROOF. Otherwise, the expected number of unique elements from the epoch is strictly smaller than $\frac{1}{7} \cdot R + \frac{6}{7} \cdot \frac{R}{8} = R/4$, contradicting Lemma 2. □

Therefore, with probability at least $1/7$, $\mathcal{A}$ must spend at least $(R/8 - M)/B = \Omega(R/B)$ I/Os writing $R/8$ elements from epoch j to the disk.

WR Reporting. Next we will prove Statement 2 of Theorem 1, namely, if $\mathcal{A}$ spends $o(perm(R) \cdot \log \frac{N}{R})$ expected I/Os processing N stream elements, it must incur $\Omega(perm(R))$ I/Os returning a WR sample set.

Our hard *workload* includes the stream constructed at the beginning of the section, and a WR sample request at the end of each epoch $j \geq 3$. We claim that the total I/O cost of processing the entire workload is $\Omega(perm(R) \cdot \log \frac{N}{R})$ in expectation, which suffices for establishing Statement 2 (because there are $O(\log \frac{N}{R})$ sample requests).

Our proof of the claim is a corollary of Lemma 1. Fix an arbitrary epoch $j \geq 3$. Let H be the total expected I/O cost of $\mathcal{A}$ in processing the elements of the epoch and the sample request at the end of the epoch. By Corollary 3, with probability at least $1/7$, the WR sample set Q fetched at the end of epoch j contains at least $R/8$ distinct elements from epoch j. The sequence of those $R/8$ elements in Q is a random $(R/8)$-permutation of the elements in the epoch. Choosing $c = 16$, we ensure $R/8 \geq 2M$; and hence, $H = \Omega(perm(R/8)) = \Omega(perm(R))$ by Lemma 1. It thus follows that $\Omega(perm(R) \cdot \log \frac{N}{R})$ expected I/Os are needed to process the entire workload.

5. TIME-BASED SLIDING WINDOW SAMPLING

We now set off to tackle Problem 2 defined in Section 1.3. An element is said to be *alive* if it falls in the current sliding window, and *expired* otherwise.

As a major obstacle in overcoming Problem 2, an algorithm often does *not* know the number n of alive elements, but nonetheless would need to make decisions

based on n. This is especially true in the so-called *two-bucket sampling problem* which, as identified by Braverman et al. [4], is a subproblem vital to settling Problem 2. We present a new algorithm to solve the subproblem I/O-efficiently in Section 5.1. In Section 5.2, we combine this algorithm with ideas from the *exponential histogram* [5] to solve Problem 2. Finally, we analyze the hardness of Problem 2 in Section 5.3.

5.1 Two-Bucket Sampling

Problem. Let U_1 and U_2 be two subsequences of the data stream satisfying the following 5 conditions:

1. They are disjoint and consecutive (i.e., the first element in U_2 arrived directly after the last element in U_1)
2. They together cover all the alive elements; and U_1 covers at least one alive element (but perhaps also some expired elements). In other words, if we denote by α and β the lengths of U_1 and U_2, respectively, then n can be anywhere from $\beta+1$ to $\alpha+\beta$.
3. $\alpha \geq R$ and $\beta \geq 2\alpha$.
4. We have taken two size-R WoR sample sets T_1 and S_1 of U_1, and a size-R WoR sample set S_2 of U_2, such that T_1, S_1, S_2 are independent from each other.
5. Each sample $e \in T_1$ also carries an *index*: if e is the i-th element of U_1, then its index equals i.

Our goal is to compute a size-R WoR sample set S of the current sliding window in $O(R/B)$ I/Os.

Algorithm. We denote by γ the number of alive elements in U_1; in other words, $n = \beta+\gamma$. Remember that the value of γ is unknown (which renders n unknown). The following is a crucial lemma, whose proof is non-trivial, and hence, deferred to the end of the subsection:

LEMMA 3. *Given values c_1, c_2 satisfying $0 \leq c_1 \leq c_2 \leq R$, a Bernoulli trial returns a binary random variable that equals 1 with probability $\frac{\alpha-c_1}{\beta+\gamma-c_2}$, and 0 with the remaining probability. Given T_1, we can support R independent Bernoulli trials, each with its own c_1, c_2. The total number of I/Os required is $O(R/B)$.*

Next we describe our algorithm for two-bucket sampling. First, generate a random number Y defined as follows. Imagine taking a size-R WoR sample set from the set of integers in $[1, \beta+\gamma]$; Y equals the number of sampled integers falling in $[1, \alpha]$. Y can be easily obtained by repeating the following step R times ($Y = 0$ initially): cast a coin with head probability $\frac{\alpha-Y}{\beta+\gamma-j}$, where j is the number of times the step is already performed ($j = 0$ for the first time); increase Y by 1 if the coin heads. By Lemma 3, the generation of Y requires $O(R/B)$ I/Os.

Second, take a size-Y WoR sample set $\hat{S}_1$ of S_1. Then, our final S is the union of (i) the alive elements of $\hat{S}_1$ (we can tell if an element is alive from its timestamp)—denote by h the number of them—and (ii) a size-$(R-h)$ WoR sample set of S_2. Since only offline WoR sampling is performed, both $\hat{S}_1$ and S can be produced in $O(R/B)$ I/Os.

Correctness. Let X be a random variable equal to the output of our algorithm. Fix an arbitrary size-R subset S of the current sliding window. We will prove that

$$\mathbf{Pr}[X = S] = 1/\binom{\beta+\gamma}{R}.$$

Define $X' = X \cap U_1, X'' = X \cap U_2, S' = S \cap U_1$, and $S'' = S \cap U_2$. Thus:

$$\begin{aligned}\mathbf{Pr}[X = S] &= \mathbf{Pr}[X'' = S'' \mid X' = S'] \cdot \mathbf{Pr}[X' = S'] \\ &= \frac{1}{\binom{\beta}{R-|S'|}} \cdot \mathbf{Pr}[X' = S']\end{aligned}$$

We analyze $\mathbf{Pr}[X' = S']$ by expanding it:

$$\begin{aligned}&\mathbf{Pr}[X' = S'] \\ =\ & \sum_{y=|S'|}^{R} \mathbf{Pr}[X' = S' \mid Y = y] \cdot \mathbf{Pr}[Y = y] \\ =\ & \sum_{y=|S'|}^{R} \frac{\binom{\alpha-\gamma}{y-|S'|}}{\binom{\alpha}{y}} \cdot \frac{\binom{\alpha}{y}\binom{\beta+\gamma-\alpha}{R-y}}{\binom{\beta+\gamma}{R}} \\ =\ & \frac{1}{\binom{\beta+\gamma}{R}} \sum_{y=|S'|}^{R} \binom{\alpha-\gamma}{y-|S'|}\binom{\beta-(\alpha-\gamma)}{R-y} \\ =\ & \frac{\binom{\beta}{R-|S'|}}{\binom{\beta+\gamma}{R}}.\end{aligned}$$

It thus follows that $\mathbf{Pr}[X = S] = 1/\binom{\beta+\gamma}{R}$, as desired.

Proof of Lemma 3. We will first tackle another subproblem—which we call *random seeding*—whose goal is to compute from T_1 an array *rand* of size R defined as follows:

- Each cell *rand*$[j]$ ($1 \leq j \leq R$) stores a pair (b, i), where i is a random variable drawn uniformly from $[1, \alpha]$, and b is a bit that equals 1 if the i-th element of U_1 is alive, or 0 otherwise (hence, $\mathbf{Pr}[b = 1] = \gamma/\alpha$).
- The R cells of *rand* are independent from each other.

This subproblem can be trivially solved in $O(R)$ time in RAM. Specifically, we can obtain from T_1 a size-R WR sample set of U_1 in $O(R)$ time. Given a $j \in [1, R]$, let e be the j-th element in the WR sample set. We then set *rand*$[j] = (b, i)$ with i being the index of e, and b indicating whether e is alive (which we can tell from its timestamp). In EM, however, this strategy takes $\Theta(perm(R))$ I/Os which is the cost to produce the WR sample set. Next, we present an alternative solution with I/O cost $O(R/B)$.

Our algorithm is designed based on the WoR-to-WR conversion algorithm in Section 2.2. First, generate the array A as described in Section 2.2, and accordingly, the value of J (i.e., the number of distinct values in A). Then, take a size-J WoR sample set of T from T_1. Recall that the algorithm of Section 2.2 would now produce a random permutation of T—denote that permutation as Π. We can no longer afford to do the same (because it requires $O(perm(R))$ I/Os). The key change here is to determine only the positions of two special elements of T in Π, as explained next.

Define e_l as the element with the largest index of all the expired elements in T. Denote by i_l the index of e_l; if e_l does not exist, then $i_l = 0$. Similarly, define e_r as the element with the smallest index of all the alive elements in T. Denote by i_r the index of e_r; if e_r does not exist, then $i_r = \alpha + 1$. It is clear that e_l, e_r, i_l, and i_r can be obtained by scanning T once in $O(J/B) = O(R/B)$ I/Os.

Recall that we cannot afford to generate Π; instead, we only generate the position p_l of e_l in Π, and the position of p_r of e_r in Π. Specifically, the integer p_l is chosen from $[1, J]$ uniformly at random, after which p_r is chosen from $[1, J] \setminus \{p_l\}$ uniformly at random.

Next, generate the *rand* array as follows. First, for each j where $A[j] = p_l$ (or p_r), place i_l (or i_r, resp.) in the second field of $rand[j]$. Second, for every remaining $j \in [1, R]$, place an integer drawn uniformly at random from $[1, \alpha] \setminus [i_l, i_r]$ in the second field of $rand[j]$. At this moment, every cell of *rand* contains in its second field an integer i that is in $[1, i_l] \cup [i_r, \alpha]$. We finalize the cell as $(1, i)$ if $i \geq i_r$, or $(0, i)$ otherwise. The above steps can be accomplished in $O(R/B)$ I/Os.

LEMMA 4. *The rand array produced by our algorithm fulfills the requirements of random seeding.*

PROOF. We denote by I the sequence of indexes in the *rand* array produced by our algorithm (namely, for each $j \in [1, R]$, if $rand[j] = (b, i)$, then $I[j] = i$). Fix an arbitrary sequence $V \in [1, \alpha]^R$. We will prove that $\mathbf{Pr}[I = V] = 1/\alpha^R$, which is sufficient to establish the lemma.

Recall that U_1 contains γ alive elements. Precisely, the elements of U_1 with indexes in $[1, \alpha - \gamma]$ have expired, while those with indexes $[\alpha - \gamma + 1, \alpha]$ are alive. Given a sequence $\Sigma \in [1, \alpha]^R$, let us define four values: (i) $i_l(\Sigma)$ is the largest integer in Σ that is at most $\alpha - \gamma$ (if no such integer exists, $i_l(\Sigma) = 0$), (ii) $i_r(\Sigma)$ is the smallest integer in Σ that is at least $\alpha - \gamma + 1$ (if no such integer exists, $i_r(\Sigma) = \alpha + 1$), (iii) $c_l(\Sigma)$ is the number of times that $i_l(\Sigma)$ appears in Σ, and (iv) $c_r(\Sigma)$ is the number of times that $i_r(\Sigma)$ appears in Σ. Given integers i_1, i_2, z_1, z_2, we denote by $X(i_1, i_2, z_1, z_2)$ the set of all sequences $\Sigma \in [1, \alpha]^R$ such that $i_l(\Sigma) = i_1, i_r(\Sigma) = i_2, c_l(\Sigma) = z_1$, and $c_r(\Sigma) = z_2$.

Our algorithm conceptually decides a random permutation Π of T as follows: (i) the positions of e_l and e_r are determined as described earlier, and then (ii) randomly permute the other elements of T over the remaining positions. Π, in turn, determines a WR sample set Q together with the array A, as described by our WoR-to-WR algorithm in Section 2.2. From Q, we can define a sequence $Q' \in [1, \alpha]^R$ where $Q'[j]$ $(1 \leq j \leq R)$ is the index of element $Q[j]$. Notice that Q' distributes uniformly at random from $[1, \alpha]^R$. We can now write $\mathbf{Pr}[I = V]$ into the following:

$$\begin{aligned} & \mathbf{Pr}[I = V] \\ = & \sum_{i_1, i_2, z_1, z_2} \Big(\mathbf{Pr}[I = V \mid Q' \in X(i_1, i_2, z_1, z_2)] \cdot \\ & \mathbf{Pr}[Q' \in X(i_1, i_2, z_1, z_2)] \Big) \end{aligned}$$

There are two crucial observations. First, if Q' belongs to $X(i_1, i_2, z_1, z_2)$, then so does I. In other words, $\mathbf{Pr}[I = V \mid Q' \in X(i_1, i_2, z_1, z_2)]$ is positive *only if* $i_1 = i_l(V)$, $i_2 = i_r(V)$, $z_1 = c_l(V)$, and $z_2 = c_r(V)$. Second, conditioned on that Q' belongs to $X(i_l(V), i_r(V), c_l(V), c_r(V))$, the I returned by our algorithm is drawn uniformly at random from $X(i_l(V), i_r(V), c_l(V), c_r(V))$. It thus follows that

$$\begin{aligned} & \mathbf{Pr}[I = V] \\ = & \mathbf{Pr}[I = V \mid Q' \in X(i_l(V), i_r(V), c_l(V), c_r(V))] \cdot \\ & \mathbf{Pr}[Q' \in X(i_l(V), i_r(V), c_l(V), c_r(V))] \\ = & \frac{1}{|X(i_l(V), i_r(V), c_l(V), c_r(V))|} \cdot \\ & \frac{|X(i_l(V), i_r(V), c_l(V), c_r(V))|}{\alpha^R} \\ = & 1/\alpha^R. \end{aligned}$$

□

Next, we show how to perform a Bernoulli trial using only a single pair (b, i) in the *rand* array, which will then complete the proof of Lemma 3 because the array allows us to perform R such independent trials. In fact, we will explain how to generate from (b, i) a binary random variable Y that equals 1 with probability $\frac{\beta - c_2}{\beta + \gamma - c_2}$. With Y, a Bernoulli trial can be supported by casting another coin with head probability $\frac{\alpha - c_1}{\beta - c_2}$ (note that $\frac{\alpha - c_1}{\beta - c_2} \leq 1$ under the condition $0 \leq c_1 \leq c_2 \leq R \leq \alpha \leq \beta/2$).

Motivated by [4], we generate Y from (b, i) by the following procedure. If $b = 0$, then immediately we set $Y = 1$. Otherwise, we toss a coin with head probability $\frac{\alpha\beta'}{(\beta' + \alpha - i)(\beta' + \alpha - i + 1)}$, where $\beta' = \beta - c_2$. We set $Y = 0$ if the coin heads, or $Y = 1$ otherwise.

LEMMA 5. $\mathbf{Pr}[Y = 1] = \frac{\beta - c_2}{\beta + \gamma - c_2}$.

PROOF. We will instead prove $\mathbf{Pr}[Y = 0] = \gamma/(\beta' + \gamma)$. This is true because

$$\begin{aligned} \mathbf{Pr}[Y = 0] &= \sum_{j=\alpha-\gamma+1}^{\alpha} \mathbf{Pr}[Y = 0 \mid i = j] \cdot \mathbf{Pr}[i = j] \\ &= \sum_{j=\alpha-\gamma+1}^{\alpha} \frac{\alpha\beta'}{(\beta' + \alpha - j)(\beta' + \alpha - j + 1)} \cdot \frac{1}{\alpha} \\ &= \sum_{j=0}^{\gamma-1} \frac{\beta'}{(\beta' + j)(\beta' + j + 1)} \\ &= \beta' \cdot \sum_{j=0}^{\gamma-1} \left(\frac{1}{\beta' + j} - \frac{1}{\beta' + j + 1} \right) \\ &= \beta' \cdot \left(\frac{1}{\beta'} - \frac{1}{\beta' + \gamma} \right) = \frac{\gamma}{\beta' + \gamma}. \end{aligned}$$

□

5.2 Algorithm for Problem 2

This subsection explains our solution to Problem 2.

Merging WoR Sample sets. We will often need to solve a subproblem defined as follows. Let P_1 and P_2 be two disjoint sets of elements, whose cardinalities are known. Let S_1 and S_2 be size-R WoR sample sets of P_1 and P_2, respectively. The goal is to obtain a size-R sample set S of $P_1 \cup P_2$. Assuming that S_1 and S_2 are stored in arrays, next we describe an algorithm that achieves this goal with $O(R/B)$ I/Os.

First, we obtain the number s_1 (or s_2) of samples to take from S_1 (or S_2, resp.). For this purpose, we first set $s_1 = s_2 = 0$, $n_1 = |P_1|$ and $n_2 = |P_2|$. Then, repeat the following two step R times: Cast a coin with head probability $\frac{n_1}{n_1+n_2}$—if the coin heads, increase s_1 and decrease n_1 each by 1; otherwise, increase s_2 and decrease n_2 each by 1. After s_1 and s_2 are ready, we simply take a size-s_1 sample set S_1' from S_1, and a size-s_2 sample set S_2' from S_2 using offline sampling (Section 2.1). Then, we return S in an array that concatenates S_1' and S_2'.

Structure. We tackle Problem 2 by dividing a suffix of the stream into a list of subsequences $U_1, U_2, ..., U_m, buf$ that satisfy all the following conditions:

$P1$: These subsequences are disjoint and consecutive: The first element of U_{i+1} succeeds the last element of U_i $(1 \leq i \leq m-1)$, and the first element of *buf* succeeds the last element of U_m. We refer to each of $U_1, ..., U_m$ as a *bucket*.

$P2$: The length $|buf|$ of *buf* can be anywhere between 0 and $2R$. The length of a bucket, however, must be $2^j R$ for some integer $j \geq 0$. Moreover, $|U_i| \geq |U_{i'}|$ if $i < i'$.

$P3$: If $|buf| < R$, then $m = 0$ (i.e., no bucket exists).

$P4$: If $|U_1| = 2^j R$ for some $j \geq 1$, then for each $j' \in [0, j]$, there is at least one but at most two buckets with length $2^{j'} R$.

$P5$: U_1 must contain at least one alive element.

The above conditions imply $m = O(\log(n/R))$ where n is the number of alive elements currently.

A bucket U of length $2^j R$ with $j \geq 1$ defines a *left* and a *right sub-bucket* whose lengths are both $2^{j-1} R$. These buckets form a partition of U with the left sub-bucket covering the older elements in U. A bucket of length R does not define any sub-buckets.

We maintain a structure that stores the following information.

- For each bucket U, store its length, and the oldest and newest elements covered. They are the *boundary elements* of U. Store the same for each sub-bucket of U.
- For each bucket U, store 3 independent size-R WoR sample sets of the elements covered by U. If U has sub-buckets, for each sub-bucket U', store 3 independent size-R WoR sample sets of the elements covered by U'. In other words, there are up to 9 WoR sample sets associated with U and its sub-buckets. All these sample sets are independent from *buf*, and from the WoR sample sets of all other buckets and their sub-buckets.
- Define a *super-bucket* $U^\star$ as the subsequence that concatenates $U_2, U_3, ..., U_m$ and *buf* (i.e., excluding U_1). Store 3 independent size-R WoR sample sets of the elements covered by $U^\star$. These sample sets are independent from those of U_1 and its sub-buckets.

It is clear from the above description that the space of our structure is $O(\frac{R}{B} \log \frac{n}{R})$.

Producing a Sample Set. We first explain how to obtain a size-R WoR sample set S from our structure. If $m = 0$ or $|U_1| = R$, then the current sliding window has $O(R)$ elements, in which case we can simply run offline WoR sampling to compute S in $O(R/B)$ I/Os.

Now consider that U_1 has length $2^j R$ for some $j \geq 1$. By Properties $P3$ and $P4$, the super-bucket $U^\star$ must have length at least $R + \sum_{k=0}^{j-1} 2^k R = 2^j R$. Denote by U_1^{left} and U_1^{right} the left and right sub-buckets of U_1, respectively. Next, we identify two subsequences V_1 and V_2 by distinguishing two cases:

- *Case 1: the oldest alive element is in* U_1^{left}. Set $V_1 = U_1^{left}$, and V_2 to the subsequence formed by concatenating U_1^{right} and $U^\star$. Obtain a WoR sample set of V_2 by merging a WoR sample set of U_1^{right} with that of $U^\star$ in $O(R/B)$ I/Os.
- *Case 2: the oldest alive element is in* U_1^{right}. Set $V_1 = U_1^{right}$ and $V_2 = U^\star$.

In both cases, it is ensured that $2|V_1| \leq |V_2|$. We then produce the target S by performing two-bucket sampling on V_1 and V_2 in $O(R/B)$ I/Os.

We compute a size-R WR sample set from S by running our algorithm in Section 2.2 for converting a WoR sample set to a WR one. There is one subtle issue that deserves clarification. Recall that our conversion algorithm requires generating R binary random variables X of the form: X takes 1 with probability c/n, and 0 otherwise, where c is an integer in $[0, R-1]$, and n is the number of alive elements currently. The subtlety is that we do not know the value of n. This issue can be resolved by resorting to Lemma 3, where the values of α, β, and γ satisfy $\alpha = |V_1|$ and $\beta = |V_2|$. To generate an X, we first perform a Bernoulli trial with $c_1 = c_2 = 0$ to obtain a binary random variable Y. If $Y = 0$, we immediately set $X = 0$. Consider now $Y = 1$; we toss a coin with head probability c/α, and set X to 1 if the coin heads, or to 0 otherwise. It is easy to verify that $Y = 1$ with probability $c/(\beta + \gamma) = c/n$.

The above discussion also explains why we chose to maintain 3 independent WoR sample sets for each bucket: the two-bucket sampling requires 2 sample sets of V_1 (see Section 5.1), one from V_2, while the WoR-to-WR conversion demands another one from V_1.

Maintenance. It remains to explain how to update our structure upon the arrival of a new element e at time t_{now}. If U_1 does not exist, we discard the elements in *buf* that have expired. Otherwise we first discard all the buckets in which all the elements have expired, and then, recompute the WoR sample sets of $U^\star$ by merging those of the current $U_2, U_3, ..., U_m$ and *buf*. If the new U_1 has size $2^j R$, the cost of merging is $O(2^j R/B)$. We amortize this cost on the elements in the old (just discarded) U_1, each of which bears only $O(1/B)$ I/Os.

Now we update the WoR sample sets of $U^\star$ with e using the algorithm of [7] in Section 2.3, which requires $O(1/B)$ I/Os per element. Next, add e into *buf*. If $|buf| < 2R$, our update algorithm finishes. Otherwise, we remove the first R elements of *buf*, and make them into a new bucket of length R. If at this moment the number of buckets with length R is at most 2, our update algorithm finishes.

In general, when there are 3 buckets with the same length $2^j R$ for some $j \geq 0$, we fix it by combining the oldest two—denoted as U_1' and U_2'—of those buckets into a new bucket U with length $2^{j+1}R$. Specifically, the WoR sample set of U is obtained by merging those of U_1' and U_2', while U_1' and U_2' now serve as the sub-buckets of U. The sub-buckets of U_1' and U_2' can now be discarded. The merging may result in 3 buckets with length $2^{j+1}R$, which is fixed in the same manner. Standard analysis of the exponential histogram [5] shows that all the merging increases the amortized cost of each element by $O(1/B)$.

Finally, if the above merging results in a new U_1, we discard the WoR sample sets of $U^\star$, and recompute them from the WoR sample sets of $U_2, U_3, ..., U_m$, and *buf* in $O(2^j R/B)$ I/Os, assuming $|U_1| = 2^j R$. We charge the cost on the $\Omega(2^j R/B)$ elements that were newly added to U_1, so that each of them bears $O(1/B)$ I/Os.

In summary, our maintenance algorithm performs $O(1/B)$ amortized I/Os per element. We thus have completed the whole proof of Theorem 2.

5.3 Lower Bounds

Next, we will show that our algorithm in Section 5.2 is optimal by establishing Theorem 3. As required by the theorem, we assume that $N \geq cR$ and $R \geq cM$ with c being a sufficiently large constant. Our proof is similar to the one in Section 4.

Our discussion will focus on a data stream where the timestamp of the i-th ($i \geq 1$) element is $\lfloor i/R \rfloor (\tau + 1)$. That is, the first R elements are associated with timestamp 0, the next R with timestamp $\tau+1$, still the next R with timestamp $2(\tau + 1)$, and so on.

Define an *epoch* $j \geq 1$ to be the sequence of elements with timestamp $(j - 1)(\tau + 1)$ ($j = 0, 1, 2, ...$). In other words, each epoch has size R; and the total number of epochs is $O(N/R)$. Note that a WoR or WR sample set issued at the end of epoch j must contain elements only in this epoch.

Element Processing Cost. We now prove Statement 1 of Theorem 3. We will argue that any algorithm $\mathcal{A}$ must perform $\Omega(R/B)$ expected I/Os in each epoch, and hence, $\Omega(N/B)$ expected I/Os for all epochs.

Consider first WoR sampling. At the end of epoch j ($= 1, 2, ...$), $\mathcal{A}$ must be keeping all the R elements in this epoch (because it must return all these elements for a sample request issued at this moment). Therefore, for $R \geq 3M$, $\mathcal{A}$ must have written $\Omega(R/B)$ I/Os during the epoch.

For WR sampling, imagine issuing a sample request at the end of epoch j. By Lemma 3 of [10], when R is large enough, with at least half probability this request fetches $\Omega(R)$ distinct elements in epoch j. This means that, with at least half probability, $\mathcal{A}$ must have written $\Omega(R/B)$ I/Os during the epoch when c is sufficiently large.

WR Reporting. Next we prove Statement 2 of Theorem 3, namely, if $\mathcal{A}$ spends $o(perm(R) \cdot \frac{N}{R})$ expected I/Os processing N stream elements, it must incur $\Omega(perm(R))$ I/Os returning a WR sample set.

Our hard *workload* includes the aforementioned stream and a WR sample request at the end of each epoch. We claim that $\mathcal{A}$ must incur $\Omega(perm(R) \cdot \frac{N}{R})$ expected I/Os to process the entire workload, which is sufficient to establish Statement 2 (because there are $O(N/R)$ sample requests).

Let H be the expected number of I/Os needed to process epoch j ($= 1, 2, ...$) and the sample request at the end of epoch j. As mentioned earlier, with at least half probability, the WR sample set Q returned by $\mathcal{A}$ has at least $\Omega(R)$ distinct elements, all of which must be in epoch j. The sequence of those elements in Q is a random $\Omega(R)$-permutation of the elements in epoch j. When c is sufficiently large, by Lemma 1 we know $H = \Omega(perm(R))$. It thus follows that $\Omega(perm(R) \cdot \frac{N}{R})$ expected I/Os are needed to process the entire workload.

ACKNOWLEDGEMENTS

This work was supported in part by Grants GRF 4168/13 and GRF 142072/14 from HKRGC.

6. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.

[4] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. *JCSS*, 78(1):260–272, 2012.

[5] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. of Comp.*, 31(6):1794–1813, 2002.

[6] C. Fan, M. Muller, and I. Rezucha. Development of sampling plans by using sequential (item-by-item) selection techniques and digital computers. *Am. Stat. Assn. J.*, 57:387–402, 1962.

[7] R. Gemulla and W. Lehner. Deferred maintenance of disk-based random samples. In *EDBT*, pages 423–441, 2006.

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

[9] J. Gustedt. Efficient sampling of random permutations. *Journal of Discrete Algorithms*, 6(1):125–139, 2008.

[10] X. Hu, M. Qiao, and Y. Tao. Independent range sampling. In *PODS*, pages 246–255, 2014.

[11] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. *SIAM J. of Comp.*, 34(6):1443–1463, 2005.

[12] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *SIGMOD*, pages 299–310, 2004.

[13] T. Jones. A note on sampling a tape-file. *CACM*, 5(6):343, 1962.

[14] Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting. *SIAM J. of Comp.*, 36(2):394–410, 2006.

[15] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *PVLDB*, 1(1):970–983, 2008.

[16] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1):67–90, 2010.

[17] A. Pavan, K. Tangwongsan, S. Tirthapura, and K. Wu. Counting and sampling triangles from a graph stream. *PVLDB*, 6(14):1870–1881, 2013.

[18] A. Pol, C. M. Jermaine, and S. Arumugam. Maintaining very large random samples using the geometric file. *VLDB J.*, 17(5):997–1018, 2008.

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

[20] K. Yi. Dynamic indexability and the optimality of B-trees. *JACM*, 59(4), 2012.

Correctness of the Algorithm in Section 2.1

We will prove the algorithm's correctness by induction on R. The case of $R = 1$ is obvious. Next, assuming that the algorithm is correct for $R = k$, we will prove the same for $R = k + 1$. Specifically, given an arbitrary sequence $V \in P^{k+1}$, we want to prove that $\mathbf{Pr}[Q = V] = 1/n^{k+1}$ (Q is the output of our algorithm).

Denote by $Q_{\leq k}$ (or $A_{\leq k}$) the prefix of Q (or A) including its first k elements. From our inductive assumption, we know that $\mathbf{Pr}[Q_{\leq k} = V_{\leq k}] = 1/n^k$ (observe that $Q_{\leq k}$ is produced in the same way as running our algorithm for $R = k$). Next, we will show that $\mathbf{Pr}[Q[k+1] = V[k+1] \mid Q_{\leq k} = V_{\leq k}] = 1/n$, which will then complete the proof.

Let $J^\star$ be the value of J right before the generation of $A[k+1]$. Define $T_{\leq J^\star}$ as the length-$J^\star$ prefix of T (recall that T is an array storing a random permutation of its elements). We now distinguishes two cases:

Case 1: $V[k+1]$ *appears in* $V_{\leq k}$. In this case, $Q[k+1] = V[k+1]$ if and only if $A[k+1]$ equals the position of $V[k+1]$ in $T_{\leq J^\star}$, which happens with probability $1/n$.

Case 2: $V[k+1]$ *does not appear in* $V_{\leq k}$. In this case, $Q[k+1] = V[k+1]$ if and only if (i) $A[k+1] = J^\star + 1$, which happens with probability $\frac{n-J^\star}{n}$, and (ii) $V[k+1] = T[J^\star+1]$, which happens with probability $\frac{1}{n-J^\star}$. The two independent conditions happen simultaneously with probability $1/n$.