Stream Sampling over Windows with Worst-Case Optimality and ℓ-Overlap Independence

Yufei Tao · Xiaocheng Hu · Miao Qiao

Accepted by VLDB Journal in March 2017

Abstract Sampling provides fundamental support to numerous applications that cannot afford to materialize all the objects arriving at a rapid speed. Existing stream sampling algorithms guarantee small space and query overhead, but all require worst-case update time proportional to the number of samples. This creates a performance issue when a large sample set is required. In this paper, we propose a new sampling algorithm that is optimal simultaneously in all the three aspects: space, query time, and update time. In particular, the algorithm handles an update in $O(1)$ worst-case time with a very small hidden constant. Our algorithm also ensures a strong independence guarantee: the sample sets of all the queries are mutually independent as long as the overlap between two query windows is small.

1 Introduction

Stream sampling has attracted considerable research attention [2,3,9–11,13,15,16], due to its usefulness in numerous domains, e.g., statistical estimation [8], graph processing [14], computational geometry [7], data mining [5], network monitoring [12], privacy protection [4], to mention just a few. Its importance is even more prominent today as the continuously increasing update volume makes it a tough challenge to enable real-time processing in many stream applications.

Y. Tao
University of Queensland
Australia
E-mail: taoyf@itee.uq.edu.au

X. Hu
Chinese University of Hong Kong
Hong Kong
E-mail: immoonancient@gmail.com

M. Qiao
Massey University
New Zealand
E-mail: m.qiao@massey.ac.nz

A stream is an unbounded sequence of elements e_1, e_2, e_3, ..., where e_i ($i \geq 1$) denotes the i-th element. Let n be the number of elements received so far. Let r be a system parameter that denotes the sample size. A *sampling query* uses an integer $w \geq 1$ to indicate a *window* that is the set of w most recent elements, namely, $\{e_{n-w+1}, e_{n-w+2}, ..., e_n\}$. It returns r independent elements, each of which is taken uniformly at random from the window. An example is *"take 10,000 random tweets from the 1 million most recent tweets"*.

We permit a sampling query to supply an arbitrary window length $w \in [1, n]$, namely, the sampling can be carried out on any *granularity* of recency. As two extremes, a query with $w = 1$ essentially extracts the newest element, whereas a query with $w = n$ samples from the entire stream since the beginning of time.

There are three main challenges:

1. How to ensure query correctness using a structure whose size is far less than n.
2. How to minimize the cost of (i) updating the structure upon receiving a new element, and (ii) answering a query.
3. How to guarantee ℓ-*overlap independence*: for any set of queries where the windows of any two queries share at most ℓ common elements, the query results must be mutually independent.

The last challenge deserves some extra explanation. The ℓ-overlap independence property is vital to data analysis that requires the results of multiple queries. This is common in *sliding-window continuous aggregation* [1], of which we show a general form:

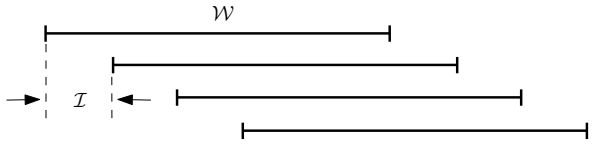


Fig. 1 Continuous aggregation

"*execute the following after every $\mathcal{I}$ elements:*

estimate the number of elements satisfying a predicate $\mathcal{P}$ from the $\mathcal{W}$ most recent elements

display a running average, which is the mean of all the estimates so far"

where $\mathcal{I}$ and $\mathcal{W}$ are integer parameters, and $\mathcal{P}$ is an arbitrary `WHERE` condition as in standard SQL. A stream system registers a large number of such requests, each of which has its own $\mathcal{I}$, $\mathcal{W}$, and $\mathcal{P}$. For each request, after every $\mathcal{I}$ elements, the system

(i) performs a sampling query with a window length $\mathcal{W}$;
(ii) estimates the number of qualifying elements from the samples.

See Figure 1 for the windows of 4 queries. If the estimates obtained from these queries are mutually independent, the quality of the running average quickly improves thanks to the central limit theorem. Ensuring the independence is precisely the goal of ℓ-overlap independence (in the above example, $\ell = \mathcal{W} - \mathcal{I}$).

1.1 Motivation

The previous research has focused exclusively on a special form of ℓ-overlap independence with $\ell = 0$, which we call *disjoint independence* (in other words, independence is guaranteed only if the windows are *tumbling*, instead of *sliding*). For $\ell = 0$, the best known algorithm [11] uses $O(r \log(n/r))$ space—which is asymptotically optimal—and answers a sampling query in $O(r \log(n/r))$ time. Given a new element, the structure can be updated in $O(1)$ amortized time but $O(r \log(n/r))$ worst-case time.

This work is motivated by several observations:

- The aforementioned structure is not suitable when the sample size r is large (which is necessary to perform estimates for selective predicates, as shown in the experiments), rendering $O(r \log(n/r))$ worst-case update time excessively expensive.
- The lack of ℓ-overlap independence with $\ell > 0$ forces a stream system to discard confidence intervals for the type of queries in Figure 1: calculating such intervals anyway is statistically *incorrect* without the independence guarantee.

1.2 Our Contributions

In this paper, we address all the above issues:

- For disjoint independence, we present an algorithm with optimal worst-case guarantees in all aspects: $O(r \log(n/r))$ space, $O(r)$ query time, and $O(1)$ update time.
- For ℓ-overlap independence, we present an algorithm with the optimal worst-case guarantees again in all aspects: $O(\ell + r \log(n/r))$ space, $O(r)$ query time, and $O(1)$ update time. Note that the space consumption increases linearly with ℓ. We prove that this is compulsory, namely, $\Omega(\ell + r \log(n/r))$ is a matching space lower bound.

The proposed algorithms are suitable for inclusion in the existing stream systems to provide the underneath sampling support. The worst-case $O(1)$ update time is a desired feature in update-intensive applications. Furthermore, the hidden constant is extremely small: only 3 random numbers need to be generated for every update.

The paper contains an extensive experimental evaluation to confirm the efficiency and effectiveness of the proposed solutions. For efficiency, we show that our algorithm has ultra-stable performance in all settings. In particular, the time to process *every* element is essentially undetectable. For effectiveness, we present the first experiments in the literature that demonstrate the usefulness of ℓ-overlap independence with $\ell > 0$, and conversely, the harmfulness of having only disjoint independence.

1.3 Paper Organization

Section 2 formally defines the problem studied. Section 3 presents preliminary results and reviews related work. Section 4 discusses how to adapt existing solutions to tackle our problem. Sections 5 and 6 describe the new algorithms and prove their theoretical guarantees. Section 7 presents an empirical evaluation. Section 8 concludes the paper with a summary of findings.

2 Problem Definition

We now formulate our stream sampling problem. For the reader's convenience, some notations that already appeared in Section 1 will be restated below. The data stream, as mentioned, is an unbounded sequence of elements $e_1, e_2, e_3, ...$ We say that e_i has *sequence number* i, and indicate so by $seq(e_i) = i$. Denote by n the number of elements already received. Given integers x, y satisfying $x \leq y \leq n$, define a *window*—denoted as $win([x, y])$—to be the set of elements whose sequence numbers are in $[x, y]$.

For a set S of elements, a *with-replacement* (WR) *sample set* R with size $r \geq 1$ has r elements, each of which is independently taken uniformly at random from S. In other words, R has $|S|^r$ possibilities, each occurring with the same likelihood.

A *sampling query* q specifies an integer parameter w that defines a window $win([n-w+1, n])$, which we abbreviate as $win(q)$. It returns a WR sample set R of $win(q)$ with size r, where r is a system parameter identical for all queries. R is a random variable with w^r possibilities. Note that both w and n are query dependent: the former is chosen freely by the query, while the latter depends on the query's issuance time.

Let integer $\ell \geq 0$ be a system parameter, such that two queries q_1, q_2 are said to be *ℓ-overlapping* if their windows share at most ℓ common elements, namely, $|win(q_1) \cap win(q_2)| \leq \ell$. For example, a query with $win([5, 20])$ is ℓ-overlapping with another query with $win([16, 80])$ for any $\ell \geq 5$. A set Q of queries is *ℓ-overlapping* if any two queries in Q are ℓ-overlapping.

The requirement of *ℓ-overlap independence* imposes a constraint that needs to hold on any ℓ-overlapping set Q of queries. Suppose, without loss of generality, that $Q = \{q_1, q_2, ..., q_g\}$. Let $w_1, w_2, ..., w_g$ be the parameters of these queries, and R_1, R_2, ..., R_g be their sample sets returned by a system. It is required that $R_1, ..., R_g$ must be mutually independent. To phrase this mathematically, let us fix query q_i for an arbitrary $i \in [1, g]$. Recall that its sample set R_i ought to have w_i^r possibilities, each happening with probability $1/w_i^r$. With ℓ-overlap independence, this must still be true, even if *conditioned on* the sample sets $R_1, ..., R_{i-1}, R_{i+1}, ..., R_g$ of all the other queries:

$$\mathbf{Pr}[R_i = \text{any possibility} \mid R_1, ..., R_{i-1}, R_{i+1}, ..., R_g] = 1/w_i^r.$$

We want to design a data structure to answer all queries correctly while ensuring ℓ-overlap independence. The structure needs to consume small space at all times, and processes every query and update efficiently. The special case of 0-overlap independence (i.e., disjoint independence) has been considered previously. Our objective is to support any value of ℓ. We will refer to the above problem as *Arbitrary-Window stream sampling with ℓ-Overlap Independence* (ℓ-AWOI).

Our discussion will concentrate on $n > 2r$. The special case of $n \leq 2r$ can be dealt with by simply keeping all the (at most $2r$) received elements in $O(r)$ space, and answering each query in a straightforward manner.

3 Preliminary

This section explains basic results on stream sampling, and discusses the previous work most relevant to ours.

3.1 Sampling without Replacement (WoR)

Let S be a set of elements. A *WoR sample set* of S with size r is a subset $T \subseteq S$ that equals any of the $\binom{|S|}{r}$ size-r subsets of S with the same probability. We follow the convention that if $r \geq |S|$, then $T = S$. It is known that a WoR sample set can be efficiently converted to a WR sample set:

Lemma 1 ([11]) *Given a size-r WoR sample set T of S, we can obtain a size-r WR sample set R of S in $O(r)$ time.*

3.2 Merging of WR Samples

The following states that two WR sample sets can be merged in a progressive manner:

Lemma 2 *Let S_1 and S_2 be disjoint sets. Let R_1 and R_2 be a size-r WR sample set of S_1 and S_2, respectively. If the sizes $|S_1|$ and $|S_2|$ are known, we can obtain from R_1 and R_2 a size-r WR sample set R of $S_1 \cup S_2$ in $O(r)$ time. Even better, the algorithm is **progressive** in the sense that the time to generate k samples in R is $O(k)$ for any $k \in [1, r]$.*

Proof Let $n_1 = |S_1|$ and $n_2 = |S_2|$. We obtain R by repeating the following for each $i \in [1, r]$:

1. Generate a random integer $x \in [1, n_1 + n_2]$.
2. If $x \leq n_1$, then add $R_1[i]$ (the i-th element of R_1) to R; otherwise, add $R_2[i]$ to R.

$O(1)$ time is spent on every i. □

3.3 Two-Window Sampling

Consider two consecutive windows $B_1 = win([\alpha, \beta])$ and $B_2 = win([\beta + 1, \gamma])$. Define $n_1 = |B_1| = \beta - \alpha + 1$ and $n_2 = |B_2| = \gamma - \beta$.

Let $V = win([\gamma - v + 1, \gamma])$ be a query window of length v satisfying

$$n_2 < v \leq n_1 + n_2 \quad (1)$$

$$v \geq n_1. \quad (2)$$

Note that, by (1), V fully covers B_2, but is covered by $B_1 \cup B_2$; see Figure 2.

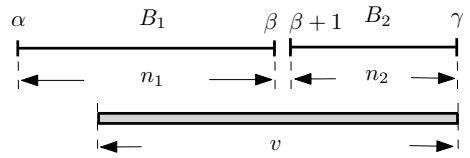


Fig. 2 Two-window sampling

Suppose that we have prepared a size-r WR sample set R_1 of B_1, and a size-r WR sample set R_2 of B_2. Each sample

e is associated with its sequence number $seq(e)$, so that we can determine if $e \in V$ by checking if $seq(e) \in [\gamma - v + 1, \gamma]$. The goal of the *two-window sampling problem* [3] is to obtain a size-r WR sample set R of V.

Lemma 3 ([3]) *The two-window sampling problem can be solved in $O(r)$ time.*

3.4 Fixed-w Stream Sampling

If we impose two restrictions on the sampling problem defined in Section 2:

- All queries must have the same parameter w, which becomes another system parameter;
- $\ell = 0$, namely, only disjoint independence is needed

then our problem degenerates into a special version that has been well understood [2, 3, 16]. For $w = \infty$ (i.e., each query samples from the entire stream), the *reservoir algorithm* of Vitter [16] gives an optimal structure of $O(r)$ space that answers a query in $O(r)$ time, and handles an update in $O(1)$ time. Braverman et al. [3] showed that the same optimal performance can also be achieved for any finite integer w.

3.5 Time-Based Stream Sampling (TSS)

In some scenarios, each stream element e_i ($i \geq 1$) is tagged with its *arrival time* $time(e_i)$, such that $i < j$ implies $time(e_i) \leq time(e_j)$ (equality may hold). If t_{now} is the arrival time of the last element, a *time window* $timewin([x, y])$ is the set of stream elements e with $time(e) \in [x, y]$. Accordingly, a *time-based sampling query* returns a size-r WoR sample set from the query window $timewin(q) = timewin([t_{now} - \lambda, t_{now}])$.

Such queries have been well studied [3, 10, 11] in the scenario where (i) integers r and λ are fixed for all queries, and (ii) the system must return mutually independent sample sets for any set of queries with disjoint windows (i.e., disjoint independence). Gemulla and Lehner [10] proved that the space consumption of any solution must be $\Omega(r \log(n/r))$. On the upper bound side, Braverman et al. [3] described a structure that uses $O(r \log n)$ space, answers a query in $O(r)$ time, and handles an update in $O(r \log n)$ time. Hu et al. [11] gave an improved structure that uses $O(r \log(n/r))$ space, answers a query in $O(r)$ time, and handles an update in $O(1)$ amortized time. The worst-case update time of the structure in [11] is $O(r \log(n/r))$.

4 How to Adapt Known Techniques for ℓ-AWOI

4.1 The Case of $\ell = 0$

We will first reveal several inherent connections between TSS (see Section 3) and the ℓ-AWOI problem with $\ell = 0$.

4.1.1 Connection 1: Upper Bound

Suppose that $\mathcal{A}_1$ is an *arbitrary* algorithm for TSS. We will show that it is possible to solve the 0-AWOI problem by using $\mathcal{A}_1$ as a black box.

Recall that the stream to 0-AWOI is a sequence of elements $e_1, e_2, e_3, ...$ which do not have explicit arrival time. In order to utilize $\mathcal{A}_1$, we manually set the arrival time $time(e_i)$ to $seq(e_i) = i$. Let us refer to the resulting "timestamped" stream as the *augmented stream*.

We feed the augmented stream to $\mathcal{A}_1$, which is parameterized for WoR queries demanding $r + 1$ samples with λ set to a gigantic integer far greater than the length of any realistic stream (e.g., $\lambda = 2^{100}$). Let $\mathcal{J}$ be the structure maintained by $\mathcal{A}_1$ (the details of $\mathcal{J}$ are unknown to us).

Let q be a (0-AWOI) query with parameter w. Recall that $win(q) = \{e_{n-w+1}, e_{n-w+2}, ..., e_n\}$. We answer the query as follows:

1. Make a copy of $\mathcal{J}$ to $\mathcal{J}'$.
2. Update $\mathcal{J}'$ with $\mathcal{A}_1$ by informing it the "arrival" of a dummy element $e_\triangle$ with $time(e_\triangle) = n + \lambda - w + 1$.
3. Issue a TSS query to $\mathcal{A}_1$ on the updated $\mathcal{J}'$. Let T be the set of $r+1$ WoR samples fetched. Obtain $T' = T \setminus \{e_\triangle\}$. As the samples in T are distinct, $|T'|$ is either r or $r + 1$ depending on whether $e_\triangle \in T$.
4. We apply Lemma 1 to convert T' to a WR sample set R of the same size. If $|R| = r$, we directly return R as the result for q. Otherwise, we arbitrarily remove an element from R, and return the remaining set.
5. Discard $\mathcal{J}'$. Note that $\mathcal{J}$ is never touched in the query processing. The copying is needed because we cannot afford to alter $\mathcal{J}$ with the dummy element.

The TSS query q' issued to $\mathcal{A}_1$ has a window $timewin(q') = timewin([n-w+1, n+\lambda-w+1])$ on the augmented stream, which by our construction contains precisely the same tuples as $win(q) = win([n-w+1, n])$ on the original stream, plus $e_\triangle$. The query correctness is ensured by kicking $e_\triangle$ out of T (if it is there) at Line 3.

The reduction works with any TSS algorithm $\mathcal{A}_1$. To get concrete performance bounds, let us set $\mathcal{A}_1$ to the state-of-the-art algorithm in [11], which uses $O(r \log(n/r))$ space, answers a query in $O(r)$ time, and processes an update in $O(r \log(n/r))$ time (see Section 3). Line 1 thus takes $O(r \log(n/r))$ time (i.e., same as the space of $\mathcal{J}$). Line 2 also entails $O(r \log(n/r))$ time because only a single update is performed. Lines 3 and 4 require $O(r)$ time, while

$O(1)$ time for Line 5. The overall query time is therefore $O(r \log(n/r))$.

The reduction inherits the same update cost as $\mathcal{A}_1$, namely, $O(r \log(n/r))$. We will reduce this dramatically to $O(1)$ using a new approach later.

4.1.2 Connection 2: Lower Bound

As mentioned earlier, Gemulla and Lehner [10] proved a space lower bound $\Omega(r \log(n/r))$ for the WoR version of the TSS problem. In Appendix 1, we adapt their argument to establish the same lower bound for any algorithm solving our problem under $\ell = 0$.

4.2 The Case of $\ell > 0$

For the TSS problem, to ensure disjoint independence, Braverman et al. [3] proposed using different structures to answer queries with disjoint windows. Can we enforce ℓ-overlap independence for ℓ-AWOI with $\ell > 0$ by keeping several independent structures? The answer is yes. To explain, let us first assume that the length w of every query window satisfies:

- $w \geq 2(\ell + 1)$, and
- w is a multiple of $\ell + 1$ (but this does not mean all queries must have the same length—their lengths can be at different multiples of $\ell + 1$).

In this case, we can maintain $\ell + 1$ independent structures, each of which solves the problem of Section 2 under disjoint independence (e.g., the solution in Section 4.1 will do). A query with $win([n - w + 1, n])$ can be processed as follows:

1. Compute $i = (n - w + 1) \bmod (\ell + 1)$.
2. Answer the query using the $(i + 1)$-th structure.

ℓ-overlap independence follows from the next two facts on any ℓ-overlapping queries q_1 and q_2:

- If $win(q_1) \cap win(q_2) \neq \emptyset$, the two queries are always answered by different structures, and hence, have independent sample sets.
- Otherwise, the two queries have independent sample sets either because they are answered by different structures, or if answered by the same structure, because the structure ensures disjoint independence.

The query time remains as $O(r \log(n/r))$.

The drawback of this approach is that, the introduction of $\ell + 1$ structures blows up the space cost and update overhead by a factor of $\ell + 1$: both reaching a prohibitive quadratic term $O(r\ell \log(n/r))$ in the worst case.

It is possible to extend the above method to queries with arbitrary lengths, but at the tradeoff of pushing the space even higher. We will not elaborate further down this line because the next section will present a much better approach that reduces the space to $O(r \log(n/r) + \ell)$.

5 Improved ℓ-Overlap Independence

In this section, we present a new reduction from ℓ-overlap independence to disjoint independence, which avoids the drawback encountered in Section 4.2.

Let $\mathcal{A}_2$ be an algorithm that optimally solves the 0-AWOI problem. That is, $\mathcal{A}_2$ uses $O(r \log(n/r))$ space, answers a query in $O(r)$ time, and handles an update in $O(1)$ time—such $\mathcal{A}_2$ does not exist yet, but we will propose one in Section 6. Next, we show how to deploy $\mathcal{A}_2$ as a black box to settle ℓ-AWOI for any $\ell > 0$.

5.1 Algorithm

We keep the ℓ most recent elements in a *buffer* P, which is a first-in-first-out queue because the arrival of a new element forces the oldest element out of P. The stream traffic leaving P is fed directly to $\mathcal{A}_2$. In other words, $\mathcal{A}_2$ sees only a sub-stream, which includes all but the ℓ newest elements. This completes the description of our structure. As $\mathcal{A}_2$ needs $O(r \log(n/r))$ space, the overall space is $O(\ell + r \log(n/r))$. It is easy to implement an update in $O(1)$ time, by using an array of size ℓ to manage P.

This simple structure is already powerful enough to answer any query q. Let w be the parameter of q. If $w \leq \ell$, $win(q)$ contains the w newest elements in the buffer P, making it straightforward to take r WR samples from $win(q)$ in $O(r)$ time.

In the more interesting case where $w > \ell$, define $w' = w - \ell$. Note that $win(q)$ includes the entire P, and also the newest w' elements in the sub-stream fed to $\mathcal{A}_2$. Hence, each sample for q has probability w'/w to come from the sub-stream, and $1 - w'/w$ from P. We take a coin with head probability $1 - w'/w$, toss it r times, and observe the number x of times that it comes up heads. Then:

1. Sample WR x elements from P in $O(x)$ time, which constitute a set X.
2. Issue a sampling query on $\mathcal{A}_2$ with parameter w'. From the returned sample set, sample WR $r - x$ elements, which constitute a set Y. The step takes $O(r)$ time.

The final sample set R is simply the union of X and Y. The total query time is $O(r)$.

5.2 Correctness

Let $Q = \{q_1, q_2, ..., q_g\}$ be an arbitrary ℓ-overlapping set of queries. As explained, each query result is the union of two sets X, Y (Y can be empty). Denote by X_i, Y_i the corresponding X, Y of q_i ($i \in [1, g]$).

Proposition 1 *Both statements are true:*

- *For each $i \in [1, g]$, X_i is independent of any joint variable made of any subset of $\{X_1, Y_1, ..., X_{i-1}, Y_{i-1}, X_{i+1}, Y_{i+1}, ..., X_g, Y_g\}$.*
- *For each $i \in [1, g]$, $Y_1, Y_2, ..., Y_i$ are mutually independent, when conditioned on $X_1, X_2, ..., X_i$.*

Proof The first statement is obvious because X_i is determined by random choices that do not affect any variable in the subset.

To prove the second, for each $q_j \in Q$ where $1 \leq j \leq i$, define q'_j as the query we issue to $\mathcal{A}_2$ (q'_j may be nil). $\{q_1, ..., q_i\}$ is ℓ-overlapping because it is a subset of Q. This implies that $\{q'_1, q'_2, ..., q'_i\}$ is 0-overlapping. By the disjoint independence guarantee of $\mathcal{A}_2$, the results of $q'_1, q'_2, ..., q'_i$ are mutually independent. Then, the statement follows from the fact that Y_j depends only on the result of q'_j, and random choices exclusively made for its computation. □

We are now ready to prove:

Lemma 4 *Our reduction in Section 5 guarantees ℓ-overlap independence.*

Proof The lemma can be established with basic probability theory, but the derivation is lengthy and tedious. We present a neater proof with an information theoretic argument by resorting to entropy and mutual information.

Let $Q = \{q_1, q_2, ..., q_g\}$, X_i ($1 \in [1, g]$), and Y_i be defined as explained earlier. Define $R_i = X_i \cup Y_i$. It suffices to prove:

$$H(R_1, R_2, ..., R_g) = \sum_{i=1}^{g} H(R_i) \tag{3}$$

where $H(.)$ denotes entropy. By the chain rule on entropy, we know:

$$H(R_1, R_2, ..., R_g) = \sum_{i=1}^{g} H(R_i \mid R_1, R_2, ..., R_{i-1}).$$

We will show that

$$H(R_i \mid R_1, R_2, ..., R_{i-1}) = H(R_i) \tag{4}$$

which will establish (3), and complete the proof. This is equivalent to showing that R_i is independent of the joint variable $(R_1, ..., R_{i-1})$. To prove this, we look at their mutual information $I(.)$:

$$\begin{aligned}
& I(R_i; R_1, ..., R_{i-1}) \\
= \; & I(X_i, Y_i; X_1, Y_1, ..., X_{i-1}, Y_{i-1}) \\
= \; & I(X_i, Y_i; X_1) + I(X_i, Y_i; Y_1, X_2, Y_2..., X_{i-1}, Y_{i-1} \mid X_1) \\
& \text{(by chain rule on mutual information)} \\
= \; & I(X_i, Y_i; Y_1, X_2, Y_2, ..., X_{i-1}, Y_{i-1} \mid X_1) \\
& \text{(by Proposition 1)} \qquad (5) \\
= \; & I(X_i, Y_i; Y_1, Y_2, X_3, Y_3, ..., X_{i-1}, Y_{i-1} \mid X_1, X_2) \\
= \; & ... \\
= \; & I(X_i, Y_i; Y_1, Y_2..., Y_{i-1} \mid X_1, X_2, ..., X_{i-1}) \\
= \; & I(X_i; Y_1, Y_2..., Y_{i-1} \mid X_1, X_2, ..., X_{i-1}) \\
& + I(Y_i; Y_1, Y_2..., Y_{i-1} \mid X_1, X_2, ..., X_i) \\
= \; & 0 \text{ (by Proposition 1)}
\end{aligned}$$

which validates (4). □

5.3 Lower Bound

We already showed (in Appendix 1) that $\Omega(r \log(n/r))$ is a space lower bound. In fact, $\Omega(\ell)$ is also a space lower bound because *any algorithm must store the ℓ newest elements*. To see this, imagine, before the next element comes, *repeatedly* issuing queries with the same $w = \ell$. The set of such queries is ℓ-overlapping. Hence, their sample sets must be mutually independent. This, in turn, implies that by issuing enough queries and a union of their results, we must end up seeing all the ℓ newest elements, none of which can therefore be discarded. We thus have obtained a space lower bound of $\Omega(\ell + r \log(n/r))$.

6 An Optimal Structure

This section will establish the main result of our paper:

Theorem 1 *For the ℓ-AWOI problem, there is a structure of $O(\ell + r \log(n/r))$ space that can be maintained in $O(1)$ time per update, and answers a query in $O(r)$ time. All complexities hold in the worst case.*

By the reduction of Section 5, it suffices to consider the problem only for $\ell = 0$. Therefore, we will concentrate on that special instance in the rest of the section and prove:

Theorem 2 *For the 0-AWOI problem, there is a structure of $O(r \log(n/r))$ space that can be maintained in $O(1)$ time per update, and answers a query in $O(r)$ time. All complexities hold in the worst case.*

6.1 Structure

We view the stream from $h(n)$ levels, where

$$h(n) = \lfloor 1 + \log_2(n/r) \rfloor.$$

At the i-th level where $1 \leq i \leq h(n)$, we partition the sequence numbers 1, 2, ..., n into intervals[1] of the same length $2^{i-1}r$. The set of elements in each interval is called a *bucket*.

[1] Such intervals are sometime termed "dyadic intervals", and are also used by the algorithms in [3,10].

In other words, the first bucket at level i is $win([1, 2^{i-1}r])$, the second $win([2^{i-1}r + 1, 2^i r])$, and so on. Figure 3 illustrates the buckets at 5 levels.

A bucket $win([x, y])$ is said to be *z-complete* for any integer z satisfying $y \leq z$. There are $\lfloor z/(2^{i-1}r) \rfloor$ z-complete buckets at level i. We refer to the window

$$win\left(\left[\left\lfloor \frac{z}{2^{i-1}r} \right\rfloor \cdot 2^{i-1}r + 1, z\right]\right)$$

as the *z-residue* at level i (it contains the elements that (i) have sequence numbers at most z, but (ii) are not covered by z-complete buckets). The z-residue is empty if z is a multiple of $2^{i-1}r$.

Our structure has an *anchor number* Z_{anc} that is always a positive multiple of r. It separates the stream into two parts:

- *Head*: the set of elements with sequence number from 1 to Z_{anc}.
- *Tail*: the set of elements with sequence number from $Z_{anc} + 1$ to n.

We store information about these two parts differently, as explained below.

6.1.1 Head Component

For each level $i \in [1, h(Z_{anc})]$, we refer to the last two Z_{anc}-complete buckets (if they exist) as the *canonical buckets*. Figure 3 illustrates such buckets with bold segments. Our structure keeps for each canonical bucket:

- If at level 1, all the elements therein;
- Otherwise, a size-r WR sample set of the elements therein.

The structure also stores a size-r WR sample set of the Z_{anc}-residue at each level i.

6.1.2 Tail

Our structure simply keeps all the elements in the tail, whose length will be represented as l_{tail}. We enforce the following invariant at all times:

$$l_{tail} \leq 2r \cdot h(Z_{anc}). \tag{6}$$

6.1.3 Space

For each level $i \leq h(Z_{anc})$, we store at most $3r$ elements. Therefore, the total space consumption is

$$O(r \cdot h(Z_{anc}) + l_{tail}) = O(r \log(n/r)).$$

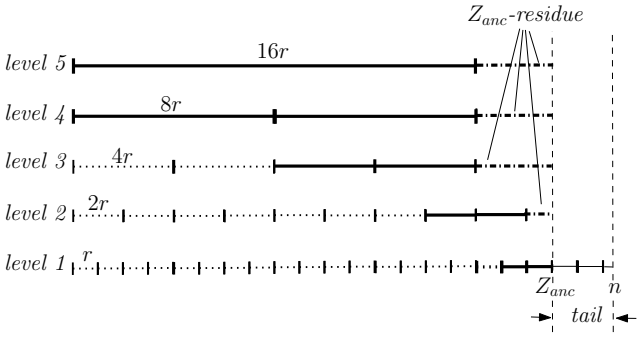


Fig. 3 Illustration of our structure (thick solid lines represent canonical buckets)

6.2 Query

Let q be a query with parameter w. If $w \leq l_{tail}$, $win(q)$ is a subset of the tail, where all the elements are directly retained, making it straightforward to take r WR samples from $win(q)$.

Next, we consider $w > l_{tail}$. Define:

$$v = w - l_{tail}.$$

The case where $v \leq r$ is also trivial because, once again, all the elements in $win(q)$ are available: they are in the tail and the most recent level-1 bucket at Z.

The subsequent discussion focuses on $v > r$. We chop $win(q) = [n - w + 1, n]$ into two disjoint windows: $V_1 = win([n - w + 1, Z_{anc}])$ and $V_2 = win([Z_{anc} + 1, n])$. Note that V_2 is exactly the tail. We will first obtain a sample set R^* on V_1, and then merge it with samples from the tail to produce the final sample set.

6.2.1 Computing R^*

This can be achieved by two-window sampling. Let i be the maximum integer satisfying

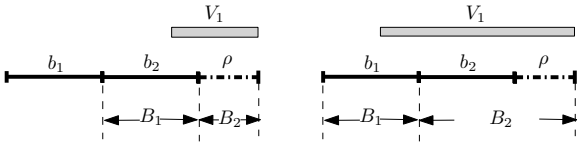
$$2^{i-1}r < v.$$

By definition, $2^i r \geq v$.

Denote by b_1, b_2 the two level-i canonical buckets (with b_2 as the more recent one), and ρ the level-i Z_{anc}-residue. Thus, $|b_1| = |b_2| = 2^{i-1}r$ and $|\rho| < 2^{i-1}r$. Let R_{b_1}, R_{b_2}, R_ρ be the WR sample sets in our structure for b_1, b_2, ρ, respectively. In the special case where $i = 1$, R_{b_1} can be directly produced from b_1 in $O(r)$ time on the fly (because b_1 is stored entirely), and similarly for R_{b_2}.

$2^{i-1}r < v$ indicates that ρ does not cover V_1, whereas $v \leq 2^i r$ indicates that V_1 must be covered by $b_1 \cup b_2 \cup \rho$. We formulate a two-window instance by distinguishing two possibilities:

- *Case 1: V_1 is covered by $b_2 \cup \rho$.* See Figure 4a. Set $B_1 = b_2$ and $B_2 = \rho$; accordingly, set $R_1 = R_{b_2}$ and $R_2 = R_\rho$.



(a) Case 1 (b) Case 2

Fig. 4 Answering a query

– *Case 2: V_1 is not covered by $b_2 \cup \rho$.* See Figure 4b. Set $B_1 = b_1$ and $B_2 = b_2 \cup \rho$. Accordingly, set $R_1 = R_{b_1}$, and obtain R_2 by applying Lemma 2 to merge R_{b_2} and R_ρ.

B_1, B_2, R_1, R_2 and V_1 define a two-window sampling instance, noticing that $n_1 = |B_1|$, $n_2 = |B_2|$, and v satisfy (1) and (2). Solving it with Lemma 3 gives R^*. The time required is $O(r)$.

6.2.2 The Final Samples

First, generate a size-r WR sample set R^{**} of the tail in $O(r)$ time. Then, apply Lemma 2 to merge R^* and R^{**} into R, which is returned as the final query result.

6.3 The Update Framework

Recall that the anchor number Z_{anc} needs to satisfy (6) at all times. An incoming element is directly appended to the tail, increasing l_{tail} by 1. This pushes up the left hand side of (6) by 1. Eventually, (6) will be invalidated such that the structure must be replaced by a new one.

To achieve $O(1)$ worst-case update time, we must start building the new structure well before the invalidation of (6). We achieve the purpose by doing the construction *progressively*, namely, dividing it into *pieces* of work, where each piece takes only $O(1)$ time. At most one piece is performed upon receiving a new element.

Specifically, we commence the construction of the next structure when—defined as the *triggering moment*—the l_{tail} of the current structure satisfies

$$l_{tail} = r \cdot h(Z_{anc}) + 1. \quad (7)$$

The construction must have finished when l_{tail} reaches $2r \cdot h(Z_{anc})$. Therefore, the number of work pieces is set to $2r \cdot h(Z_{anc}) - r \cdot h(Z_{anc}) = r \cdot h(Z_{anc})$.

The rest of the section is devoted to implementing the above strategy and proving its correctness, in particular, why it ensures disjoint independence.

6.4 Building the New Structure

At the triggering moment of the current structure, we choose the new structure's anchor time Z_{anc}^{new} as:

$$Z_{anc}^{new} = Z_{anc} + r \cdot h(Z_{anc}).$$

Notice that, at this moment, n equals $Z_{anc} + l_{tail} = Z_{anc} + r \cdot h(Z_{anc}) + 1$. In other words, $Z_{anc}^{new} = n - 1$ currently, namely, the new structure contains the last stream element in its tail. The rest of the tail is trivial to build: every time a new element arrives, append it in constant time.

Next, we will focus on explaining how to build the head component of the new structure.

6.4.1 An $O(r \cdot h(Z_{anc}))$-Time Algorithm for the Head

The head component concerns only the elements with sequence number from 1 to Z_{anc}^{new}—that have *already* arrived. By the description in Section 6.1, it suffices to obtain a WR sample set for (i) each canonical bucket, and (ii) the Z_{anc}^{new}-residue at each level. We will achieve the purpose using only the information from the current structure.

WR Sample Sets of Canonical Buckets. The definition below will be useful:

Definition 1 (Materializable Bucket) A bucket $w([x, y])$ at level $i \in [1, h(Z_{new})]$ is *materializable* if $Z_{anc} < y \leq Z_{anc}^{new}$.

In other words, a materializable bucket is Z_{anc}^{new}-complete but not Z_{anc}-complete. Figure 5b illustrates all the materializable buckets using double lines.

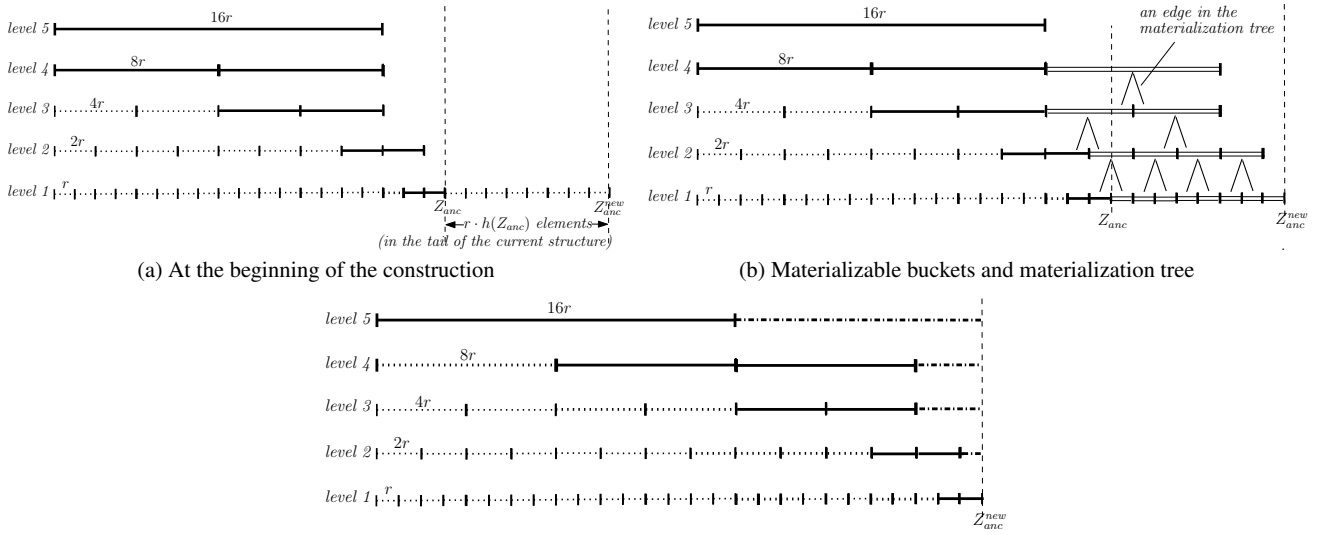
Lemma 5 *For $i \geq 2$, a level-i materializable bucket b is always the union of two level-$(i-1)$ buckets b_1, b_2 satisfying one of the following:*

– *both b_1, b_2 are materializable, or*
– *one of them is a canonical bucket of the current structure, and the other is materializable.*

Proof Suppose that b_2 is more recent than b_1. From the fact that b (being materializable) is Z_{anc}^{new}-complete but not Z_{anc}-complete, it is easy to verify that b_2 must also be Z_{anc}^{new}-complete but not Z_{anc}-complete. Hence, b_2 is materializable. The lemma then follows from the observation that, at any level, a materializable bucket must be preceded by another materializable bucket or a canonical bucket of the current structure. □

The lemma suggests that the materializable buckets define a binary *materialization tree* as follows. The root of the tree is the materializable bucket at the highest level[2]. Recursively, if a node is a materializable bucket b of level at least 2, its child nodes are the two buckets b_1, b_2 as stated in Lemma 5. Figure 5b illustrates the tree by using edges to link up a parent bucket with its children. The root of the tree is the materializable bucket at level 4, whereas a leaf is either

[2] There cannot be two materializable buckets sharing the same highest level; otherwise, there would be a materializable bucket at an even higher level.



(a) At the beginning of the construction

(b) Materializable buckets and materialization tree

(c) Canonical buckets and residue of the new structure

Fig. 5 Illustration of the new structure's construction

a level-1 materializable bucket, or a canonical bucket of the current structure.

Lemma 6 *The materialization tree contains at most* $2h(Z_{anc})$ *leaves and at most* $2h(Z_{anc})$ *internal nodes.*

Proof There are $r \cdot h(Z_{anc})/r = h(Z_{anc})$ materializable buckets at level 1, all of which can be leaves in the materialization tree. On the other hand, each of the $h(Z_{anc})$ levels can contribute at most 1 leaf (which must be the most recent canonical bucket at this level in the current structure) to the tree. Hence, the materialization tree has at most $2h(Z_{anc})$ leaves, and therefore, at most $2h(Z_{anc}) - 1$ internal nodes. □

Lemma 7 *We can compute a size-*r *WR sample set of every materializable bucket by applying Lemma 2 at most* $2h(Z_{anc})$ *times in total.*

Proof Let bucket b be an internal node in the materialization tree with child nodes b_1, b_2. Having prepared the WR sample sets of b_1, b_2, we can merge those sets to obtain a WR sample set of b using Lemma 2.

This implies that we can produce the sample sets of all the materializable buckets by carrying out the merging in a bottom up manner. Lemma 7 then follows from the two facts below:

- The WR samples of every leaf bucket in the materialization tree either are directly available from the current structure (this is the case if the bucket is at level 2 or above, and hence, a canonical bucket of the current structure) or can be obtained in $O(1)$ time per sample (this is the case if the bucket is at level 1, and hence, has all its elements explicitly captured).
- There are at most $2h(Z_{anc})$ internal nodes according to Lemma 6.

□

The simple observation below clarifies the relevance of materializable buckets to the canonical buckets of the new structure:

Proposition 2 *Every canonical bucket of the new structure is either a canonical bucket of the current structure, or a materializable bucket.*

It then follows that the size-r WR sample sets of all the canonical buckets in the new structure can be obtained by at most $2h(Z_{anc})$ applications of Lemma 2, namely, one application for each internal node in the materialization tree. Therefore, all those applications take up $O(r \cdot h(Z_{anc}))$ time in total.

WR Sample Sets for the Z_{anc}^{new}-Residue. A similar bottom-up merging idea can also be applied to create the WR sample set of the Z_{anc}^{new}-residue at each level $i \in [2, h(Z_{new})]$. Let s be the length of the residue[3]. If $s = 0$, there is no residue at level i. Otherwise, $s \geq r$ because the residue must include at least one level-1 bucket. On the other hand, recall that $s < 2^{i-1}r$ by definition of residue. Thus, it is possible to define $i^* \in [1, i-1]$ as the maximum integer satisfying $2^{i^*-1} \leq s/r$.

Lemma 8 *The* Z_{anc}^{new}*-residue at level* i *is the union of (i) the most recent level-*i^* *canonical bucket of the new structure, and (ii) the* Z_{anc}^{new}*-residue at level* i^**.*

Proof Let $\rho = win([j+1, Z_{anc}^{new}])$ be the Z_{anc}^{new}-residue at level i, where j is the largest multiple of $2^{i-1}r$ not exceeding Z_{anc}^{new}. Since j is a multiple of $2^{i-1}r$, it is also a multiple of $2^{i^*-1}r$. Therefore, there is a level-i^* bucket b starting at

[3] The value of s can be calculated in $O(1)$ time as the difference between Z_{anc}^{new} and the largest multiple of $2^{i-1}r$ at most Z_{anc}^{new}.

Algorithm 1: CONSTRUCTING THE HEAD COMPONENT OF THE NEW STRUCTURE (NON-PROGRESSIVELY)

```
/* first compute the WR sample sets of the
   materializable buckets at level ≥ 2   */
for each materializable bucket b at level ≥ 2 in bottom-up order do
    b1, b2 ← the children of b in the materialization tree;
    apply Lemma 2 to produce a WR sample set of b by
    merging the WR sample sets of b1, b2;
/* The WR sample sets of all the canonical
   buckets in the new structure are now
   ready; we proceed to compute the WR
   sample set of the Z_anc^new-residue at each
   level ≥ 2                             */
for i = 2 to h(Z_anc^new) do
    ρ ← the Z_anc^new-residue at level i;
    i* ← maximum integer j satisfying 2^(j-1) ≤ s/r, where s
    is the length of the level-i residue;
    b* ← most recent level-i* canonical bucket of the new
    structure;
    ρ* ← the Z_anc^new-residue at level i*;
    apply Lemma 2 to produce a WR sample set of ρ by
    merging the WR sample sets of b* and ρ*;
```

$j+1$. Bucket b must be the most recent canonical bucket of the new structure at level i^*. Otherwise, level i^* has another bucket b' that is Z_{anc}^{new}-complete, and is more recent than b. This implies that ρ must fully contain a canonical bucket of the new structure at level i^*+1, violating the definition of i^*. The lemma then follows from the fact that the portion of ρ outside b is precisely the level-i^* Z_{anc}^{new}-residue. □

To illustrate the lemma, let us examine Figure 5c (where the Z_{anc}^{new}-residue residue of each level is indicated using dash-dot lines). Consider the Z_{anc}^{new}-residue at level, for instance, $i=3$, which has length $s=3r$ (the residue contains 3 level-1 buckets), making $i^*=2$. Lemma 8 states that the Z_{anc}^{new}-residue at level $i=3$ is the union of a level-2 canonical bucket of the new structure and the Z_{anc}^{new}-residue at level 2. This is indeed the case as we can see in Figure 5c. Note that the figure also shows all the canonical buckets (in solid lines) in the new structure.

Corollary 1 *We can compute a size-r WR sample set of the Z_{anc}^{new}-residue at all levels $i \in [2, h(Z_{new})]$ by applying Lemma 2 at most $h(Z_{anc})$ times in total.*

Proof Lemma 8 suggests that we can do so by applying Lemma 2 in ascending order of i. □

By Corollary 1, all the residue's WR sample sets in the new structure can be created in $O(r \cdot h(Z_{anc}))$ time. Algorithm 1 summarizes the above discussion.

Algorithm 2: CONSTRUCTING THE NEW STRUCTURE (PROGRESSIVELY)

Input: e: the incoming element
1 append e to the tail of the new structure;
2 perform one piece of work in Algorithm 1; specifically, generate 3 samples in the progressive execution of the algorithm in Lemma 2;

6.4.2 Making the Algorithm Progressive

It is clear from Lemma 7 and Corollary 1 that, our construction algorithm essentially invokes Lemma 2 at most $3h(Z_{anc})$ times. Recall that one application of Lemma 2 produces r samples. Hence, all the at most $3h(Z_{anc})$ applications generate no more than $3r \cdot h(Z_{anc})$ samples. The *progressive* nature of Lemma 2 allows us to precisely control the progression of the whole generation by simply counting how many samples are churned out. We do so by dividing the construction algorithm into $r \cdot h(Z_{anc})$ disjoint pieces (as demanded in Section 6.3), each of which yields exactly 3 samples.

The progressive version of our construction algorithm is described in Algorithm 2. Given a new element e, we first append e to the tail of the new structure (as mentioned before), and then carry out a single piece of work as just defined.

The construction finishes after $r \cdot h(Z_{anc})$ elements, all of which have entered the tail of the new structure, which therefore has a length $l_{tail}^{new} = r \cdot h(Z_{anc})$. Since Z_{anc}^{new} is strictly greater than Z_{anc}, it follows that $l_{tail}^{new} \leq r \cdot h(Z_{anc}^{new})$. In other words, the new structure has not yet reached its triggering moment (see Inequality 7).

Remark. As mentioned, when $n < 2r$, we retain all the incoming elements. When $n = 2r$, the elements kept so far can be regarded as a structure satisfying our definition in Section 6.1 with anchor number $Z_{anc} = r$ and a tail of length $l_{tail} = n - Z_{anc} = r$. Then, this structure reaches its triggering moment at receiving the next incoming element, and kicks off the progressive construction of the next structure.

6.5 Proof of Disjoint Independence

This subsection is dedicated to proving that our algorithm guarantees disjoint independence. This will complete the whole proof of Theorem 2.

It suffices to consider $r=1$ because the algorithm essentially runs r independent threads in parallel, each of which produces one sample for a query.

As explained in Section 6.2, we obtain the sample set R of a query q by merging R^* and R^{**}. When $r=1$, the effect of merging is simple: R equals either R^* or R^{**}. If $R = R^{**}$, then R depends on random numbers generated exclusively

for q, in which case independence is obvious. The subsequent discussion considers only queries for which $R = R^*$; call such queries *non-trivial*.

Let $Q = \{q_1, q_2, ..., q_g\}$ be a 0-overlapping set of non-trivial queries. Suppose that $win(q_i) = win([x_i, y_i])$, for $i \in [1, g]$. Without loss of generality, assume that $[x_1, y_1]$, $[x_2, y_2]$, ..., $[x_n, y_n]$ are in ascending order. Let $R_1, R_2, ..., R_g$ be their results returned by our algorithm. To prove that these results are mutually independent, by chain rule on entropy, we only need to prove that, for every $j \in [2, g]$, R_j is independent of the joint variable $(R_1, R_2, ..., R_{j-1})$.

Our update algorithm creates a series of anchor numbers. For each $i \in [1, j]$, denote by Z_i the largest anchor number Z satisfying $Z \leq y_i$. Note that q_i being non-trivial implies $x_i \leq Z_i$. It thus follows that $Z_i \in [x_i, y_i]$, which in turn indicates that $Z_1, Z_2, ..., Z_j$ must be distinct (because $[x_1, y_1], ..., [x_j, y_j]$ are disjoint), namely:

$$Z_1 < Z_2 < ... < Z_{j-1} < x_j \leq Z_j \leq y_j. \quad (8)$$

Lemma 9 *R_j is a function of, i.e., uniquely determined by*

- *the random numbers generated by our update algorithm for $n \geq Z_{j-1} + 1$, and*
- *the random numbers generated exclusively for q_j by our query algorithm.*

Proof Consider any bucket $b = win([x, y])$ at level 2 or above, for which our structure stores a singleton WR sample set R_b (when $r = 1$). Let Z be any positive integer. Define $R_b(Z)$ to be the set of information below:

- *Whether* $R_b[1]$ (the only sample in R_b) has a sequence number at least $Z + 1$;
- If yes to the above, the element of $R_b[1]$.

Note that $R_b(Z)$ does not care about the concrete element of $R_b[1]$ if its sequence number is at most Z. Also, if $Z+1 > y$, then apparently $R_b(Z)$ is empty.

The proof of the following lemma can be found in Appendix 2.

Lemma 10 *$R_b(Z)$ is a function of the random numbers generated by our update algorithm for $n \geq Z + 1$.*

The sample set R_j of q_j is answered by two-window sampling (Lemma 3), which operates on two windows B_1 and B_2, with (singleton) sample sets R_{B_1} and R_{B_2}, respectively. R_j is completely determined by $R_{B_1}(Z_{j-1})$ and R_{B_2}, and a random number generated for q_j. Next, we claim that both $R_{B_1}(Z_{j-1})$ and R_{B_2} are functions of the random numbers stated in Lemma 9.

The claim is true about R_{B_2} because B_2 is completely within $win(q_j)$, where all the elements are strictly after $n = Z_{j-1}$; see (8). B_1 is always a bucket. If B_1 is at level 1, its elements are entirely retained; the claim is true because $R_{B_1}(Z_{j-1})$ depends solely on a random number generated for q_j. If B_1 is at a higher level, the claim is true due to Lemma 10. □

Lemma 9, together with (8), shows that R_j is independent of $(R_1, R_2, ..., R_{j-1})$, thus completing the proof of disjoint independence.

7 Experiments

This section is organized in two parts with a focus on *efficiency* and *effectiveness*, respectively. Specifically, Section 7.1 will first evaluate the space, query, and update cost of the proposed algorithms, and then, Section 7.2 will demonstrate the usefulness of ℓ-overlap independence in statistical estimation. In both scenarios, we will use the current state of the art for benchmarking. All experiments were performed on a machine equipped with a 3GHz dual-core CPU and 16GB memory. The operating system was Linux (Ubuntu 14.04).

7.1 Efficiency of Stream Sampling

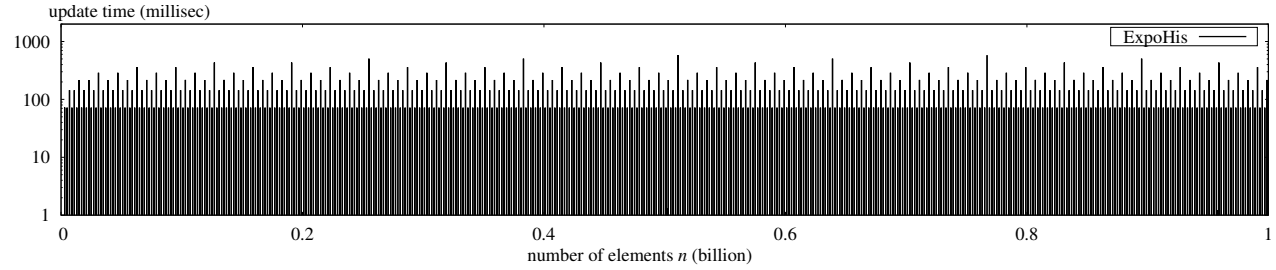
In each experiment, a stream was a sequence of 10^9 elements, the i-th ($1 \leq i \leq 10^9$) of which arrived at *timestamp* i. Each element was stored in a word of 32 bits. The element contents are irrelevant; in other words, all the results below are valid regardless of the bits within the elements.

A query *workload* consisted of 100 queries that were evenly dispersed throughout the history. Specifically, the i-th query ($1 \leq i \leq 100$) was issued right after $n = 10^7 \cdot i$ elements had been received. Each query specified a parameter w, which was generated uniformly at random from r to the current value of n.

We compared the following methods:

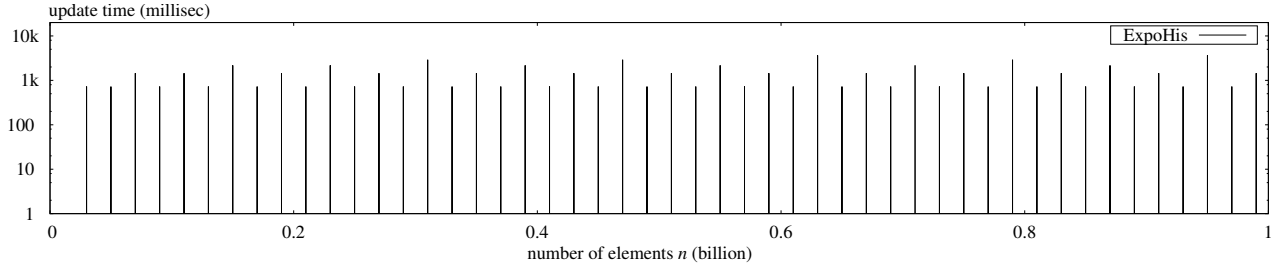
- OPTIMAL: Our algorithm as in Theorem 2 for disjoint independence $\ell = 0$, or in Theorem 1 for general ℓ-overlap independence with $\ell > 0$.
- EXPOHIS: This method integrates the best time-based sampling algorithm [11] and the approach in Section 4.1, representing the state of the art for disjoint independence. It consumes $O(r \log(n/r))$ space, answers a query in $O(r \log(n/r))$ time, and handles an update in $O(r \log(n/r))$ time. The name follows from the fact that the method leverages an *exponential histogram* [6] as its underlying structure.
- EXPOHIS$^+$: It extends EXPOHIS with the approach in Section 4.2 to support ℓ-overlap independence of $\ell > 0$. It uses $O(r\ell \log(n/r))$ space, answers a query in $O(r \log(n/r))$ time, and handles an update in $O(r\ell \log(n/r))$ time.

We inspected the influence of three parameters r, ℓ, and n on the space, query, and update performance of each method.



update time of OPTIMAL: < 0.05 millisecond *throughout* the stream

(a) $r = 1$ million



update time of OPTIMAL: < 0.05 millisecond *throughout* the stream

(b) $r = 10$ million

Fig. 6 Update time on individual elements (disjoint independence)

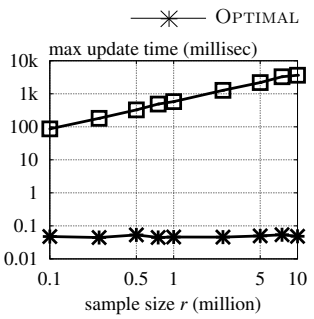


Fig. 7 Maximum update time vs. r (disjoint independence)

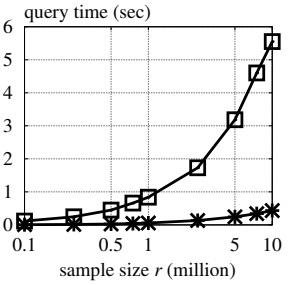


Fig. 8 Query time vs. r (disjoint independence)

The value of r was varied from 10^5 to 10^7, that of ℓ from 0 to 10^7, and n from 1 to 10^9. Recall that a solution to the disjoint independence problem is the basis of a method settling ℓ-overlap independence with $\ell > 0$. Hence, we will first study the former in Section 7.1.1, and then the latter in Section 7.1.2.

7.1.1 Disjoint Independence

The competing methods in this subsection are OPTIMAL and EXPOHIS. Figures 6a and 6b give their cost in processing each stream element, for $r = 10^6$ and 10^7, respectively. These results reveal the characteristic behavior of EXPOHIS, namely, *"spiky" update overhead*. While the method handles most elements efficiently, it must undergo expensive "overhauls" periodically. By comparing the two figures, one can see that, as r grows, although overhauls are less frequent, their cost (a.k.a. the height of a spike) becomes substantially higher. In practice, every spike has the effect of stalling the stream, thus forcing delays in all the higher-level applications. This is undesirable in a stream system, echoing the motivation of this work. OPTIMAL, in contrast, exhibited ultra-stable performance: its update cost was nearly unmeasurable on *every* element—less than 50 microseconds (10^{-6}).

We repeated the above on multiple values of r, and measured the *maximum* per-element update time of the two methods in each experiment. The results are presented in Figure 7. As expected, the worst update cost of EXPOHIS increased rapidly with r, whereas that of OPTIMAL was not affected by r.

Figure 8 plots, as a function of r, each method's average per-query time in answering all the queries in a workload issued on a stream. Recall that EXPOHIS has an extra multiplicative logarithmic factor in its query complexity compared to OPTIMAL, which explains the large gap between the two curves in the figure.

Setting r to 1 million, Figure 9 shows how the space consumption of each method changed as the stream elements arrived. The overall trend for both methods is identical, and matches precisely their space complexity $O(r \log(n/r))$. The "oscillating behavior" of EXPOHIS is a typical phenomenon of exponential histograms. Figure 10 plots the maximum space usage of the two methods as a function of r, confirming a consistent 15% saving for OPTIMAL.

Remark. It should be mentioned that there are only insignificant differences in the average per-update cost between OPTIMAL and EXPOHIS: less than 0.1 microseconds for both methods. This is not surprising because (i) they both achieve $O(1)$ update time after amortization, and (ii) most elements require very simple processing that involves only writing a memory cell and generating a single random number. The

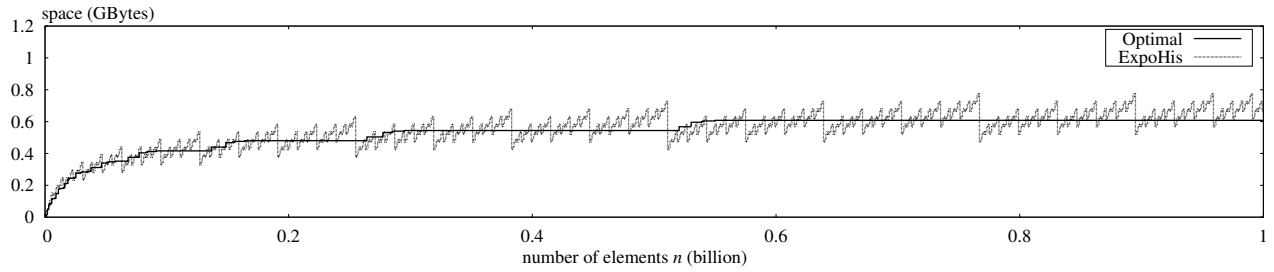


Fig. 9 Space growth with n ($r = 1$ million, disjoint independence)

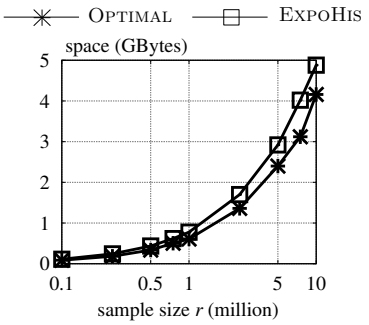


Fig. 10 Maximum space vs. r (disjoint independence)

"spikes" of EXPOHIS account for a small percentage of the overall running time. Thus, our algorithms make sense only in applications where (i) the sample size is massive, and (ii) one would like to avoid periodic stalling of the stream completely for real time processing.

As another remark, it was difficult to measure precisely the time of every update for our algorithms because it was too low. We suspect that it should be at the order of 0.1 microseconds. But since there were 10^9 updates, it was almost impossible to avoid a system-level context switch happening in one of these updates. In other words, essentially we ended up measuring the context-switch time. This explains why we claimed only "less than 50 microseconds", which we believe is a much weaker claim than our real performance guarantee.

7.1.2 ℓ-Overlap Independence

We separate the subsequent experiments of *very small* ℓ and from those of *arbitrarily large* ℓ, because as will be clear EXPOHIS$^+$ is applicable only to the former, while OPTIMAL is the only practical solution to the latter. Focus will be placed on update and space cost, because the query time of each method is the same as that of its disjoint-independence counterpart.

$\ell \leq 18$. Recall that EXPOHIS$^+$ has quadratic update and space overhead: both $O(r\ell \log(n/r))$—which should limit its usage only to low values of ℓ. The next few experiments aim to verify this intuition.

Setting $r = 1$ million and $\ell = 18$, Figures 11 and 12 give, for both OPTIMAL and EXPOHIS$^+$, the per-element update time and space consumption as the stream progressed. EXPOHIS$^+$, which put together $\ell + 1 = 19$ instances of EXPOHIS (as explained in Section 4.2), saw both types of overhead surge to 19 times that of EXPOHIS. The consequence is severe: EXPOHIS$^+$ required up to nearly 10 seconds to perform a single update, and occupied almost all the 16GB of memory on our machine. OPTIMAL, on the other hand, exhibited exactly the same update and space efficiency as disjoint independence.

Again setting r to 1 million, Figure 13 (14, resp.) makes explicit the growth of the maximum per-element update time (space usage, resp.) of the two methods as ℓ changed from 0 to 18. Evidently, EXPOHIS$^+$ scaled poorly with this parameter; it simply failed for $\ell \geq 19$ on our machine.

Gigantic ℓ. Next, we considered scenarios completely beyond the functionality of EXPOHIS$^+$: ℓ-overlap independence with huge ℓ. OPTIMAL, hence, became the only surviving method.

We repeated the experiments of Figures 13 and 14, but this time varying ℓ all the way to 10 million. The results are given in Figures 15 and 16. OPTIMAL, once again, demonstrated excellent performance in all situations. In particular, its update cost on *every* element was still below 50 microseconds, i.e., *no increase at all*. Its space consumption remained almost flat for all values of ℓ. This is not surprising: recall that the space complexity of OPTIMAL is $O(r \log(n/r)+\ell)$—the ℓ term is well dominated by the term $r \log(n/r)$, even for $\ell = 10^7$. The property is crucial for applications such as continuous aggregation as evaluated in the next subsection.

7.2 Usefulness of ℓ-Overlap Independence

We will demonstrate the effectiveness of ℓ-overlap independence in the continuous aggregate scenario mentioned in the introduction. Specifically, a *continuous query* specifies a predicate $\mathcal{P}$, a window length $\mathcal{W}$, and a re-evaluation interval $\mathcal{I}$. It estimates (using samples) the number of elements satisfying $\mathcal{P}$ in the window that includes the $\mathcal{W}$ most recent elements, and repeats this after the window has slid down by $\mathcal{I}$ elements (i.e., one estimate every $\mathcal{I}$ elements). It displays the running average of all the estimates so far. Intuitively, as long as the data distribution inside the window remains roughly the same, the accuracy of the running average should

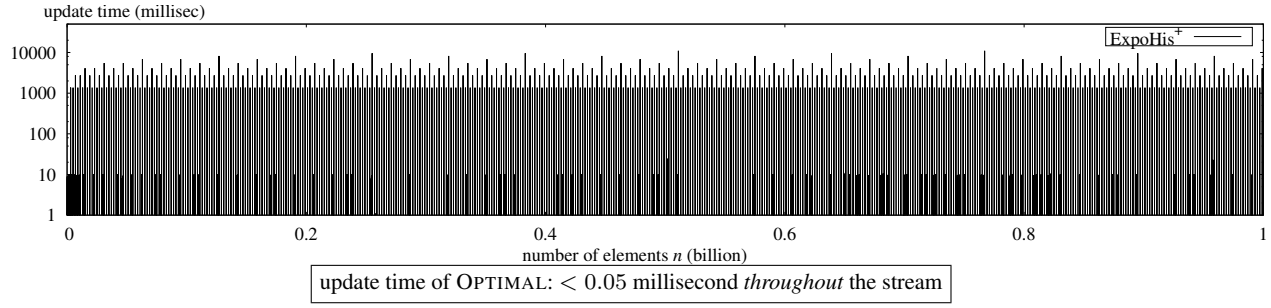


Fig. 11 Update time on individual elements ($r = 1$ million, 18-overlap independence)

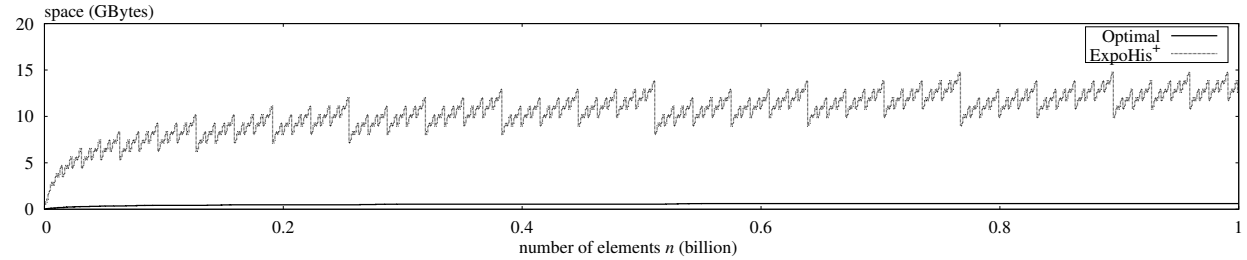


Fig. 12 Space growth with n ($r = 1$ million, 18-overlap independence)

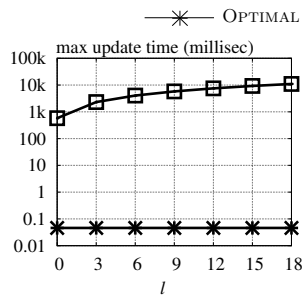


Fig. 13 Maximum update time vs. small ℓ ($r = 1$ million, ℓ-overlap independence)

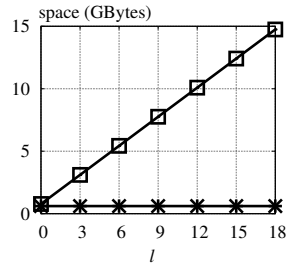


Fig. 14 Maximum space vs. small ℓ (r = 1 million, ℓ-overlap independence)

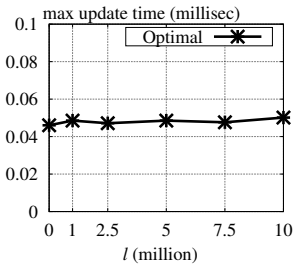


Fig. 15 Maximum update time vs. huge ℓ (r = 1 million, ℓ-overlap independence)

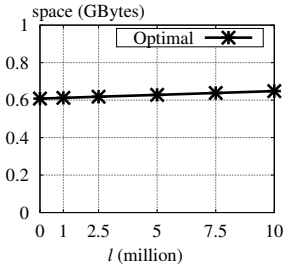


Fig. 16 Maximum space vs. huge ℓ (r = 1 million, ℓ-overlap independence)

improve over time. We will show that this is true only if the system guarantees ℓ-overlap independence with $\ell = \mathcal{W} - \mathcal{I}$ (c.f. Figure 1).

Once $\mathcal{P}$ is determined, the stream can be regarded as a bit sequence, namely, 1 (or 0, resp.) if the corresponding element satisfies (or does not satisfy, resp.) $\mathcal{P}$. When viewed this way, the continuous query essentially estimates the number of 1's in the current window. It does so by extracting r WR samples from the underlying system, counting the number x of 1 samples, and calculating the estimate as $est = x \cdot (\mathcal{W}/r)$.

In each subsequent experiment, the input stream is a sequence of 0's and 1's, with the property that there is roughly the same number of 1's in any window with the same length, as long as the length is large. Such a stream was generated as follows. Among the first F (where F is a parameter) elements in the stream, we set an arbitrary element to 1, while the rest to 0. Suppose that the element was the i-th (for some $i \leq F$). Then, we set to 1 all the elements with sequence numbers $i + j \cdot F$, for any integer $j \geq 1$, and to 0 all the other elements. In this way, any window of length $\mathcal{W} \gg F$ covers roughly $\mathcal{W}/F$ 1's. As we will see, this provides *ground truth* for assessing the estimation errors of continuous queries.

We compared:

- *Disjoint*: The underlying system, which provides the stream sampling functionality, guarantees only disjoint independence for continuous queries. The (by far) most popular algorithm achieving the purpose is an algorithm due to Braverman et al. (see Section 2.1 of [3]). Note that the algorithm was designed for *fixed-length* sliding windows, such that a separate instance of the algorithm is needed for each specific length. This increases its space consumption and update overhead as many times as the number of lengths to be supported. Nevertheless, since our objective in this subsection is effectiveness (as opposed to efficiency, as in Section 7.1), we favored [3] by pretending that its space and update cost was not an issue.
- *ℓ-Overlap*: The underlying system guarantees ℓ-overlap independence, using our algorithm of Theorem 1.

In all the following experiments, the number r of samples was set to a million, and the value of ℓ was set to 10

million. The continuous query was issued after the stream had received 10^9 elements. Parameter $\mathcal{I}$ was varied in a range from 10 (i.e., one estimate every 10 elements) to 10^5. The value of $\mathcal{W}$ was set accordingly to $\mathcal{I} + \ell$.

7.2.1 Results

Recall that the estimate obtained from one window is $est = x \cdot (\mathcal{W}/r)$, where x is the number of 1-samples. It is rudimentary to verify that this is an unbiased estimate of the true number of 1's in the window with a standard deviation of $(W/F)\sqrt{F/r}$, given that $F \gg 1$. This formula will be helpful in comprehending the key phenomenon that we will reveal shortly: *when the predicate $\mathcal{P}$ is selective, performing the estimate using only one window is not reliable, making it crucial to improve the accuracy by leveraging multiple independent estimates to compute an average.* The importance of ℓ-overlap independence lies exactly in its ability to guarantee the required independence.

Let us start with a tough challenge: $F = 10^6$, corresponding to the case where the query counts the number of "outlier elements" that come up in the stream once every 1 million arrivals. Plugging in $r = 10^6$, the above formula indicates a standard deviation of $\mathcal{W}/F$, namely, 100% standard error with respect to the ground truth $\mathcal{W}/F$! Fortunately, ℓ-overlap independence rescues this by bringing down the error quickly through averaging independent estimates. Figure 17a illustrates this for $\mathcal{I} = 10^3$ using a representative continuous query that issued 100 estimates consecutively. The y-axis shows the running averages of *ℓ-independent* and *disjoint*, as a function of the stream length (for $\mathcal{I} = 10^3$, the period of 100 estimates covers the arrival of nearly 10^5 elements). Moreover, for *ℓ-independent*, we indicate its confidence interval starting from the 30th estimate based on the central limit theorem[4]. The ground truth here is $\mathcal{W}/F \approx 10$. As expected, *ℓ-overlap* yielded an increasingly accurate running average, which after 100 estimates was very close to the ground truth. In contrast, *disjoint* was not able to improve itself at all, and still gave a gigantic error at the end. Note that *no confidence intervals can be calculated* for *disjoint* in a statistically correct manner due to the lack of confidence in its estimates.

Next, we made the job a lot easier, by decreasing F substantially to 10^4, i.e., the predicate is 100 times less selective than before. The standard deviation formula now evaluates to $\frac{1}{10}(W/F)$, i.e., 10% error. We repeated the experiment of Figure 17a. The results are shown in Figure 17b (here the ground truth is $\mathcal{W}/F \approx 1000$). Both *$\ell$-overlap* and *disjoint* were much more accurate in their first estimates. Even so, *ℓ-overlap* was able to improve its accuracy even further, whereas *disjoint* made no improvement at all.

The next evaluation demonstrates that the above is not a phenomenon on isolated queries, but indeed holds in general. After fixing the values of F and $\mathcal{I}$, we repeated the previous experiment for 30 continuous queries. This produced, for each query, a sequence of 100 estimates, for each of which we calculated its absolute relative error on the ground truth[5]. Then, for each $i \in [1, 100]$, we measured the mean of the i-th absolute relative errors of all 30 queries. Figures 18a-18f present the 100 means of *ℓ-overlap* and *disjoint* as a function of the stream length, for various combinations of F and $\mathcal{I}$. In all cases, the precision of *ℓ-overlap* improved dramatically after only a few number of queries, whereas similar improvement was absent from *disjoint*. Note that *disjoint* exhibited fluctuation when $\mathcal{I} = 10^5$. To explain, first note that the larger $\mathcal{I}$ is, the "more independent" the samples of *disjoint* tend to be. The value of $\mathcal{I} = 10^5$, intuitively, put *disjoint* in a "mixture" state where each window's sample set contains some samples that were independent from the other windows, but also other samples that were not. Its behavior exhibited in Figures 18c and 18f was the consequence of such a state.

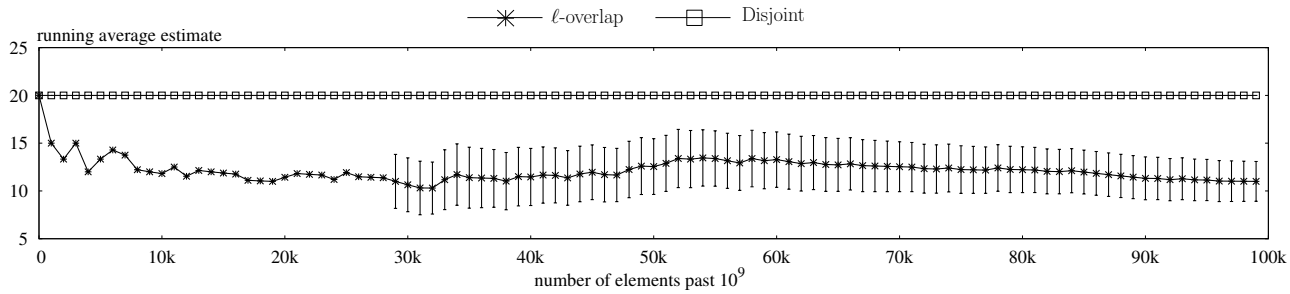
8 Conclusions and Future Work

Stream sampling is important to applications that require processing an unbounded sequence of elements. The existing solutions are inadequate because they (i) incur excessive time processing an incoming element, and (ii) can ensure only weak independence on the sample sets returned to different queries. In this paper, we give a new algorithm that is worst-case optimal in three aspects: space, query time, and update time. Furthermore, it ensures ℓ-overlap independence, which is the strongest independence guarantee in the literature so far. Besides being a solid theoretical technique, the algorithm has excellent practical efficiency as well, by processing each element in microseconds. It can therefore serve as a robust and reliable data feeder at the bottom level of modern stream systems.

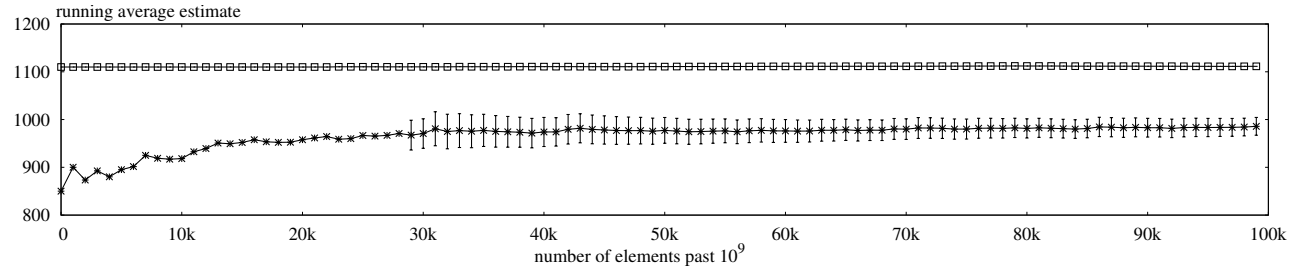
From a high level, our update algorithm is a method of *de-amortization*, namely, how to turn an algorithm that is fast on average into an algorithm with asymptotically the same worst case performance. Our techniques, in summary, are based on two key ideas. First, introduce a buffer at the tail that has the *same space* complexity as the main structure. Second, we break the periodic construction algorithm into pieces, each of which generates a *single* random number, and takes $O(1)$ time. This idea circumvents the pitfall of

[4] Salkind noted in the book entitled *Statistics for People Who (Think They) Hate Statistics* that most researchers suggest that the number of repeats should be no less than 30 before the theorem can be applied.

[5] If the ground true is act, then the absolute relative error is $|est - act|/act$.

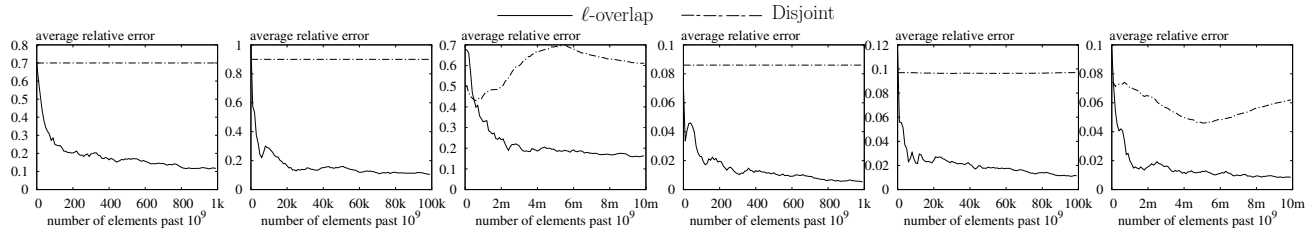


(a) $F = 10^6$, one estimate every $\mathcal{I} = 10^3$ elements, 100 estimates in total (true answer: 10)



(b) $F = 10^4$, one estimate every $\mathcal{I} = 10^3$ elements, 100 estimates in total (true answer: 1000)

Fig. 17 Running average estimates and confidence intervals (where appropriate)



(a) F=10^6, $\mathcal{I}$=10 (b) F=10^6, $\mathcal{I}$=10^3 (c) F=10^6, $\mathcal{I}$=10^5 (d) F=10^4, $\mathcal{I}$=10 (e) F=10^4, $\mathcal{I}$=10^3 (f) F=10^4, $\mathcal{I}$=10^5

Fig. 18 Absolute Relative error of the running average estimates

suspending and resuming the algorithm in a "forced" manner at the operating system's level.

In this paper, we have strived to simplify the algorithmic procedure. We leave it as an open problem whether the procedure can be made substantially simpler. For this purpose, we suspect that one may need to depart from the two ideas aforementioned—a direction that appears elusive to us at this stage.

We have concentrated on producing samples whose randomness is theoretically sound. Another reasonable direction would be to study whether we could trade theoretical randomness for implementation convenience. For instance, a reviewer of this paper mentioned the following idea to leverage EXPOHIS for approximate (disjoint-independence) sampling. Create two threads of the algorithm: one responsible for answering sampling queries, while the other thread responsible for building a new structure (using the algorithm of EXPOHIS). The query thread operates on a slightly outdated structure that ignores some recent elements. As soon as the maintenance thread finishes, the query thread replaces its old structure with the new one. Then, the maintenance thread starts to work on the incoming elements to construct the next structure (there are the elements invisible to the query thread). The query thread would provide fast response time, because the operating system automatically interleaves between the two threads. We suspect that in practice this method could be feasible and would turn out high-quality samples, although they are not provably random.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments, suggestions for improving the paper, the very interesting interaction. The review process was one of the best that we have ever experienced.

References

1. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
2. B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.
3. V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. *JCSS*, 78(1):260–272, 2012.
4. K. Chaudhuri and N. Mishra. When random sampling preserves privacy. In *CRYPTO*, pages 198–213, 2006.

5. Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. *Knowl. Inf. Syst.*, 10(3):265–294, 2006.
6. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. of Comp.*, 31(6):1794–1813, 2002.
7. G. Frahling, P. Indyk, and C. Sohler. Sampling in dynamic data streams and applications. *Int. J. Comput. Geometry Appl.*, 18(1/2):3–28, 2008.
8. W. A. Fuller. *Sampling Statistics*. Wiley, 2009.
9. R. Gemulla and W. Lehner. Deferred maintenance of disk-based random samples. In *EDBT*, pages 423–441, 2006.
10. R. Gemulla and W. Lehner. Sampling time-based sliding windows in bounded space. In *SIGMOD*, pages 379–392, 2008.
11. X. Hu, M. Qiao, and Y. Tao. External memory stream sampling. In *PODS*, pages 229–239, 2015.
12. A. Lall, V. Sekar, M. Ogihara, J. J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. In *SIGMETRICS*, pages 145–156, 2006.
13. S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1):67–90, 2010.
14. A. Pavan, K. Tangwongsan, S. Tirthapura, and K. Wu. Counting and sampling triangles from a graph stream. *PVLDB*, 6(14):1870–1881, 2013.
15. A. Pol, C. M. Jermaine, and S. Arumugam. Maintaining very large random samples using the geometric file. *VLDB J.*, 17(5):997–1018, 2008.
16. J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

Appendix 1: Space Lower Bound for Our Problem of Section 2 under Disjoint Independence

We will need the following mathematical fact:

Lemma 11 *Let x, y be any positive real values satisfying $x \geq y$ and $x \geq 1$. Then $1 - (1 - 1/x)^y = \Omega(y/x)$.*

Proof It is fundamental to verify that, for any real value z, $1 + z \leq e^z$, and for any real value $z \in [0, 1]$, $e^{-z} \leq 1 - (1 - 1/e)z$. Therefore:

$$1 - (1 - 1/x)^y \geq 1 - e^{-y/x}$$
$$\geq 1 - \left(1 - \left(1 - \frac{1}{e}\right)\frac{y}{x}\right) = \left(1 - \frac{1}{e}\right)\frac{y}{x}.$$

□

Let $\mathcal{A}_3$ be an algorithm solving our problem under $\ell = 0$. Suppose that $n \geq r$ stream elements have been received. Consider the i-th element e_i where $i \in [1, n - r]$. Define a random variable E_i to be 1 if e_i is retained by $\mathcal{A}_3$ at this moment, or 0 otherwise. Motivated by Gemulla and Lehner, we look at the query with parameter $w = n-i+1$. As each WR sample of the query picks e_i with probability $1/w$, e_i is picked by at least one of its r samples with probability $1 - (1 - 1/w)^r$. It thus follows that

$$\mathbf{Pr}[E_i = 1] \geq 1 - \left(1 - \frac{1}{n - i + 1}\right)^r$$
$$\text{(by Lemma 11)} = \Omega\left(\frac{r}{n - i}\right).$$

Hence, the expected space used by $\mathcal{A}_3$ is at least

$$\sum_{i=1}^{n-r} \mathbf{E}[E_i] = \sum_{i=1}^{n-r} \Omega\left(\frac{r}{n - i}\right) = \Omega(r \log(n/r)).$$

The worst-case space of $\mathcal{A}_3$ cannot be smaller, and thus, must also be $\Omega(r \log(n/r))$.

Appendix 2: Proof of Lemma 10

The lemma is trivial if $Z + 1 > y$; next, we assume $Z + 1 \leq y$.

Consider first $i = 2$. Let b_1, b_2 be the level-1 buckets covered by b. All the elements in b_1, b_2 are directly retained. The lemma holds on b because Lemma 2 obtains $R_b[1]$ with a single random number generated after the entire b_2 has been received, i.e., at or after $n = y \geq Z + 1$.

Consider $i = j \geq 3$. Redefine b_1, b_2 as the level-$(i - 1)$ buckets covered by b, whose size-1 sample sets are R_{b_1}, R_{b_2}, respectively. Inductively assume that the lemma holds on $R_{b_1}(Z)$ and $R_{b_2}(Z)$. Lemma 2 generates a random number, at or after $n = y$, to decide whether $R_b[1]$ equals $R_{b_1}[1]$ or $R_{b_2}[1]$. Hence, given the number, $R_b(Z)$ is fully determined by $R_{b_1}(Z)$ and $R_{b_2}(Z)$.