# Parallel Streaming Frequency-Based Aggregates

Kanat Tangwongsan*
Computer Science Program
Mahidol University
International College
kanat.tan@mahidol.ac.th

Srikanta Tirthapura
Dept. of Electrical and
Computer Engineering
Iowa State University
snt@iastate.edu

Kun-Lung Wu
IBM T.J. Watson Research
Center, Yorktown Heights
klwu@us.ibm.com

## ABSTRACT

We present efficient parallel streaming algorithms for fundamental frequency-based aggregates in both the sliding window and the infinite window settings. In the sliding window setting, we give a parallel algorithm for maintaining a space-bounded block counter (SBBC). Using SBBC, we derive algorithms for basic counting, frequency estimation, and heavy hitters that perform no more work than their best sequential counterparts. In the infinite window setting, we present algorithms for frequency estimation, heavy hitters, and count-min sketch. For both the infinite window and sliding window settings, our parallel algorithms process a "minibatch" of items using linear work and polylog parallel depth. We also prove a lower bound showing that the work of the parallel algorithm is optimal in the case of heavy hitters and frequency estimation. To our knowledge, these are the first parallel algorithms for these problems that are provably work efficient and have low depth.

## Categories and Subject Descriptors

H.1.0 [**Information Systems**]: Models and Principles—*General*; F.2.0 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*General*

## Keywords

parallel streaming; stream processing; heavy hitter; basic counting

## 1. INTRODUCTION

Today's applications need to monitor massive volumes of data and derive insights from it in real-time. In such applications, it is natural to employ parallel computing to improve processing throughput. However, parallel stream monitoring needs fundamentally new algorithms, and there does not seem to have been much work in this direction so far. In this work, we consider the design and analysis of parallel algorithms for processing a high-velocity data stream on a shared-memory machine (e.g., a multicore machine).

Following the paradigm used by streaming systems such as Apache Spark [ZDL+13], we assume the model of a *discretized stream*. The system receives data from one or more sources and divides the resulting stream into "minibatches". The streaming algorithm processes a minibatch, potentially using parallel processing, and updates its state. After processing one minibatch, the system moves onto the next minibatch. This model of processing elements has the following advantages: (1) Each minibatch can potentially be processed in parallel without a sequential bottleneck, whether in data ingestion or in processing, and (2) the view of a user of the system is simple. Queries can be answered on data received until, and including the most recent mini-batch that was processed.

While there is a long line of research on sequential streaming algorithms for a variety of problems, and empirical and experimental work on parallel streaming algorithms, basic questions on algorithms for parallel stream processing remain open. This work initiates a systematic study of parallel streaming algorithms for fundamental aggregates, with a rigorous analysis of both the algorithms' cost and accuracy guarantees.

To process a stream in parallel, one approach is to use *independent per-processor data structures* (see Figure 1). Suppose there are $p$ processors; the stream $\mathcal{S}$ is partitioned into sub-streams $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_p$, one per processor. Each processor $i$ processes $\mathcal{S}_i$ and maintains a data structure $D_i$ local to processor $i$. The different data structures $D_i, i = 1 \ldots p$ are merged together periodically, or whenever a query is posed. For this approach to work, the data structures should be *mergeable* (see, e.g., [ACH+13]). Even if mergeable data structures exist for a given aggregate, there are inherent shortcomings of the independent data structure approach in our context. In particular, the approach does not take advantage of available shared memory, and the merging step can be a sequential bottleneck. See Section 5.4 for further discussion of the independent data structure approach for heavy hitter identification.

We take a different approach to processing a stream in parallel. Instead of a per-processor data structure, we use a *single shared data structure* (see Figure 1) that the processors update cooperatively update in parallel . Updates and queries can be interleaved, and our algorithms require no locking. This approach has the following advantages when compared with the independent data structure approach: (1) the *total workspace is smaller* than the independent data structure approach. For instance, in the case of approximate heavy hitter identification, a shared data structure takes the same total memory as a sequential algorithm such as [MG82], while the independent data structure approach requires memory $p$ times larger. (2) There is no need for a further merge step to combine different data structures. This eliminates a sequential bottleneck, and allows polylogarithmic depth parallel algorithms.

### 1.1 Our Contributions

We present efficient parallel algorithms for estimating fundamental frequency based aggregates on a stream in the infinite window

*Corresponding author.

*SPAA'14,* June 23–25, 2014, Prague, Czech Republic.
Copyright 2014 ACM 978-1-4503-2821-0/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612669.2612695.

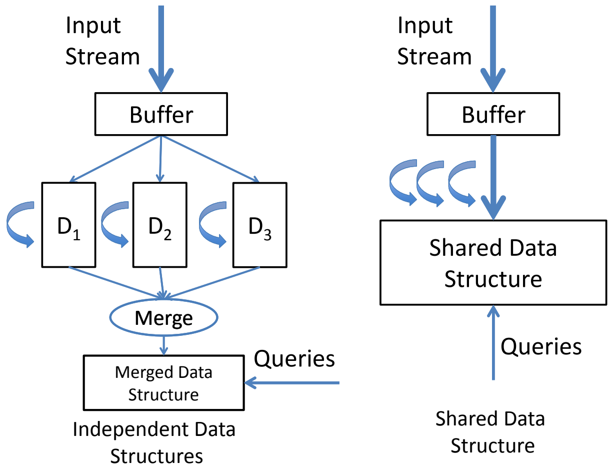

**Figure 1:** Two approaches to parallel streaming algorithms for a shared memory multiprocessor.

and the sliding window models. In the *infinite window* model, the aggregate is desired over the entire stream from the start, whereas in *sliding window* model, the aggregate is desired only over the most recent elements of the stream. We consider the count-based sliding window, defined as the $n$ most recent elements in the stream. We assume that $n$ is much larger than the size of a minibatch, and that it is undesirable to store the entire window in the main memory.

Our algorithms operate in the work-depth model assuming an underlying CRCW machine. As is standard, *work* is the total operation count of an algorithm, and *depth*, traditionally known as "parallel time", is the longest chain of dependencies within a parallel computation. Throughout, we measure space in terms of words, unless stated otherwise.

We derive algorithms for the following problems:

—*Basic Counting*: On a stream of bits, maintain the number of 1s within a count-based sliding window of size $n$. Though easy in the model of infinite window, this aggregate is non-trivial in the sliding window model, and is a fundamental problem; the work of Datar et al. [DGIM02] show how to reduce other aggregates on a sliding window, such as approximate histograms, hash tables, and $\ell_p$ norms of vectors to basic counting on a bit stream.

—*Sum*: This is a generalization of basic counting. On a stream of non-negative integers, maintain the sum of the elements within a sliding window. The maintenance of the mean of non-negative integers can be reduced to the sum.

—*Frequency Estimation*: Approximately track the frequency of items either within an infinite window or a sliding window.

—*Heavy Hitters*: Continuously track $\phi$-heavy hitters in a stream. This is a fundamental problem in stream monitoring, and has received widespread attention due to its importance in monitoring applications, including network monitoring [EV03, CH10]. We consider tracking heavy hitters in both the infinite window and sliding window settings.

—*Count-Min Sketch* [CM05]: While not an aggregate in itself, this data structure is a useful summary of frequency-based properties of a stream, and can be used for answering a variety of queries on the input stream, including point and range queries, quantiles, and heavy hitters.

Except for count-min sketch, our algorithms provides a notion of deterministic guarantee, namely an $\varepsilon$-*approximation*, where the estimate has worst-case relative error at most $\varepsilon$. However, our algorithms are randomized; hence, we state their work and depth bounds in expectation.

Our algorithms are work efficient; i.e., they perform no more work, up to constant factors, than their most efficient sequential counterparts. They also have low depth, which is polylogarithmic in the input size and in the workspace, for almost all cases. Their memory requirements are also of the same order as their sequential counterparts [1]. To our knowledge, these are the first parallel streaming algorithms that simultaneously have all these properties. Previous work on parallel and distributed streaming algorithms have mostly focused on the case of independent data structures and are unable to achieve a parallel depth that is smaller than the size of the data structure. In the case of frequency estimation and heavy hitters, we are able to, for the first time, break the barrier of $\Omega(1/\varepsilon)$ for the parallel depth.

At a high level, the difference between the infinite window case and the sliding-window case can be seen as keeping a regular counter versus keeping a counter that slides with the window. We design a synopsis data structure, called a *space-bounded block counter*, that maintains an approximate count of the number of 1s in a sliding window via deterministic sampling, while supporting parallel ingestion of a sequence of bits, which we use internally to process a minibatch. We use the parallel space-bounded block counter as a building block in algorithms for basic counting and sum over a sliding window, and in frequency estimation and heavy hitters.

## 1.2 Related Work

There have been several related works on parallel and distributed processing of a data stream, but they have usually not considered the (theoretical) efficiency of parallel algorithms for the problem.

Work on the distributed streaming model, both one-shot processing [GT01, GT04], as well as continuous monitoring [Cor13, CMYZ12, TW11, ABC09], considers computation of aggregates over the union of multiple distributed streams that are observed by different processors. A significant difference between distributed stream processing and parallel stream processing is that in the distributed case, there are physically different input streams that are observed and processed by geographically different processors. But in the parallel case, the stream is processed by multiple processors only to improve the throughput. In the distributed streams model, the focus is on minimizing the communication between processors, while in the shared-memory parallel case, the focus is more on the processing efficiency.

Das et al. [DAAE09] consider tracking frequent elements in a stream using a shared memory multicore machine. Their work uses shared data structures among multiple threads, and supports our approach of using shared data structures as opposed to independent per-processor data structures. However, unlike our work, they do not provide theoretical guarantees about the parallel performance of their algorithms. In particular, they do not have an analysis of the cost of their parallel algorithm. Cafaro and Tempesta [CT11] present a parallel algorithm for maintaining frequent elements in a stream; this algorithm uses the independent data structure approach, and employs a sequential merge step, so the depth of the algorithm is at least $\Omega(\frac{1}{\varepsilon^2})$, while our algorithm for infinite window has a depth that is polylogarithmic in $\frac{1}{\varepsilon}$. Other works on parallel streaming algorithms for frequent items includes [ZSZ+13], and our work differs in providing provable guarantees on the cost.

[1] The exception is that our algorithm for the Sum is a factor of $\log R$ worse in work and memory than the best sequential algorithm.

Sequential streaming algorithms for basic counting on a sliding window were first studied by [DGIM02], and there has been much follow up work since then, including [GT04, LT06a, XTB08, BO10]. We do not attempt a detailed survey of the prior sequential algorithms in this area, but to our knowledge, there has been no work on parallel algorithms with a provable guarantee on both performance and accuracy. There are many sequential streaming algorithms for finding frequent elements, including counter-based algorithms such as the Misra-Gries algorithm [MG82], Lossy Counting [MM02], and Space-Saving [MAE06], and sketch-based algorithms, such as Count-Sketch [CCFC02] and Count-Min sketch [CM05]. Frequent elements over sliding windows has been considered in [LT06b, HLT10].

## 2. PRELIMINARIES AND NOTATION

Throughout the paper, let $[n]$ denote the set $\{1, \dots, n\}$ and denote by $[\alpha, \beta]$ the interval $\{x \in \mathbb{R} \mid \alpha \leqslant x \leqslant \beta\}$. A sequence is written as $\langle x_1, x_2, \dots, x_{|X|} \rangle$, and for a sequence $X$, the $i$-th element is denoted by $X_i$ or $X[i]$. We say that an event happens with high probability (**whp**) if it happens with probability at least $1 - n^{-\Omega(1)}$.

A *stream* $\mathcal{S}$ is an infinite sequence of elements $e_1 e_2 e_3 \dots$, where $e_i$ belongs to a universe $\mathcal{U}$. When we want to emphasize the universe, we say that $\mathcal{S}$ is a $\mathcal{U}$-stream. A *stream segment* is a finite sequence of consecutive elements of a stream. For example, a minibatch is a stream segment. In many cases, we are only interested in prefixes of the infinite stream. Define $\mathcal{S}_t = e_1 e_2 \dots e_t$. On this stream, a *window* of size $n$, denoted by $\mathcal{W}_n(\mathcal{S}_t)$, is the segment of $\mathcal{S}_t$ consisting of the latest $n$ entries, i.e., $e_{t-n+1}, \dots, e_t$.

Often in this paper, we deal with $\{0, 1\}$-streams, which we also refer to as *binary streams*. A *compacted stream segment* (CSS) is an encoding of a segment of a binary stream where only the positions of the 1 bits and the length of the segment itself are recorded. In particular, for a binary stream segment $T$, the CSS of $T$, denoted by CSS$(T)$ is an ordered pair $(\ell, \mathsf{s})$, where $\ell$ is the length of the segment, and $\mathsf{s}$ is a sequence of length $\|T\|_0$, where $\mathsf{s}_i$ stores the position of the $i$-th 1 in the segment $T$. Here, the 0-th norm $\|\cdot\|_0$ denotes the number of non-zero entries in a stream segment. Given a binary stream, it is easy to construct its CSS using standard techniques [JáJ92]:

**Lemma 2.1** *The CSS of a binary stream segment $T$ can be computed in $O(n)$ work and $O(\log n)$ depth, where $n$ is the length of the segment $T$.*

We will also rely on the following result:

**Theorem 2.2 (Parallel Integer Sort [RR89])** *There is an algorithm* `intSort` *that takes a sequence of integer keys $a_1, a_2, \dots, a_n$, each a number between $0$ and $c \cdot n$, where $c = O(1)$, and produces a sorted sequence in $O(n)$ work and* polylog$(n)$ *depth.*

**Linear-Work Histogram:** Common to many of our algorithms is a routine for determining the frequencies of the elements in a stream segment, a problem which we call histogram construction. The following theorem shows how to do this for a stream segment of length $\mu$ in $O(\mu)$ work and $O(\text{polylog}(\mu))$ depth.

**Theorem 2.3** *There is an algorithm* `buildHist` *that takes a sequence $a_1 a_2 \dots a_\mu$, $a_i \in \mathcal{U}$ and produces a sequence $\langle (\texttt{elt} = \cdot, \texttt{freq} = \cdot) \rangle$ of distinct elements and their frequencies, reported in any order. Further, the algorithm takes $O(\mu)$ work and $O(\text{polylog}(\mu))$ depth.*

Proof. Let $h : \mathcal{U} \to \{1, \dots, R\}$, where $R = O(\mu)$, be a hash function. It suffices to use, for example, a $O(\log \mu)$-wise independent family. The algorithm proceeds as follows: First, it hashes each $a_i$ using the hash function $h(\cdot)$ and buckets elements with the same hash values together. This can be accomplished using `intSort` in $O(\mu)$ work and depth as the range of the hash function is $R = O(n)$.

Suppose the nonempty buckets are $B_1, \dots, B_t$, where $B_i$ contains the elements which hash to value $i$. We know that all elements of the same key are in the same bucket and we will process each of these buckets in parallel. That is, we call `collectBin`$(B_i)$ for each $i$, in parallel, and concatenate their results together.

```
def collectBin(B):
  (1) If $B$ is empty, return an empty sequence.
  (2) Pick an arbitrary element $e \in B$.
  (3) Let $n_e$ be the number of times $e$ occurs in $B$.
  (4) Let $B'$ be $B$ without any occurrences of $e$
  (5) return collectBin($B'$) ++ $\langle \texttt{elt} = e, \texttt{freq} = n_e \rangle$.
```

Correctness of this algorithm is straightforward and its performance depends essentially on how many distinct elements fall into the same bucket. More specifically, each pass through Steps 2–4 requires at most $O(|B_i|)$ work and $O(\log |B_i|) \leqslant O(\log \mu)$ depth. Hence, in terms of work, the total cost across all buckets is at most

$$W \leqslant \sum_{e\text{: unique elts}} r_e n_e,$$

where $n_e$ is the number of occurrences of item $e$ and $r_e$ is the number of unique elements in the bucket that $e$ hashes to. We conclude that $\mathbf{E}[W]$ is $O(\mu)$ because $\mathbf{E}[r_e] \leqslant \mu / R = O(1)$ and $\sum_e n_e = \mu$. As for depth, the overall depth is $D \leqslant \max_i \{r_i \log \mu\}$, where $r_i$ is the number of unique elements in $B_i$. Since $r_i \leqslant O(\log \mu)$ **whp**, by balls-and-bins analysis, we have $D \leqslant O(\log^2 \mu)$ **whp**. This concludes the proof. □

## 3. SPACE-BOUNDED BLOCK COUNTER

In this section, we describe a data structure for maintaining an approximate count of the number of 1s (i.e., 1-bits) in a sliding window. This data structure is an important building block of the algorithms presented later in this paper.

In a nutshell, for a parameter $\lambda$ that controls the accuracy-vs.-space tradeoff, the data structure consumes roughly $m/\lambda$ space and gives an additive-error guarantee of $\lambda$, where $m$ is the number of 1s in the sliding window. We achieve this tradeoff by devising a parallel variant of the deterministic sampling scheme of Lee and Ting [LT06b, LT06a]. In addition, this data structure has a means to limit the total space consumption to a preset limit $\sigma$.

### 3.1 $\gamma$-Snapshots

We begin by reviewing an elegant deterministic-sampling synopsis due to Lee and Ting [LT06b, LT06a] that forms the basis of our parallel counter data structure. Let $\mathcal{S}_t = \langle b_1, b_2, \dots, b_t \rangle$ be a stream and let $\omega_i$ be the position of the $i$-th 1 in the stream. In $\gamma$-snapshots, the stream is subdivided into equal-sized blocks with $\gamma$ elements each. That is, $b_1, \dots, b_\gamma$ is block $\mathbf{B}_1$. Then, $b_{\gamma+1}, \dots, b_{2\gamma}$ is block $\mathbf{B}_2$. In general, the range $b_{(k-1)\gamma-1}, \dots, b_{k\gamma}$ is block $\mathbf{B}_k$. We write $\beta(i)$ for the block number which $b_i$ belongs to.

To build a snapshot, the scheme specifies deterministically which of these blocks are sampled We provide a slightly different view from [LT06b] although it is equivalent to the original definition:

**Definition 3.1 ($\gamma$-Snapshot)** *Let $n$ be the sliding-window size. A $\gamma$-snapshot for the stream $\mathcal{S}_t = \langle b_1, b_2, \dots, b_t \rangle$, denoted by $\mathsf{SS}_{\gamma,n}(\mathcal{S}_t)$, is an ordered pair $(Q, \ell)$ where*

*(1) $Q = \{\beta(\omega_{\gamma i}) :$ block $\mathbf{B}_{\beta(\omega_{\gamma i})}$ overlaps with or falls in the window $b_{t-n+1}, \dots, b_t\}$; and*

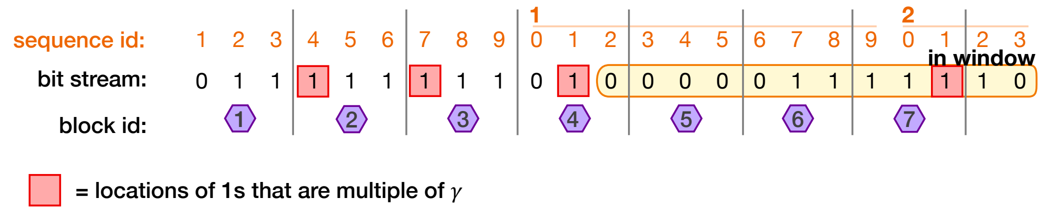

**Figure 2:** An illustration of $\gamma$-snapshot for a window of size 12 and $\gamma = 3$, showing the 1 bits located at multiple of $\gamma = 3$ indices. This results in $(Q = \{4, 7\}, \ell = 1)$.

*(2) if $p^* = \max\{\omega_{\gamma i} : \omega_{\gamma i} \leqslant t\}$, then $\ell$ is the number of 1s in the window after $p^*$ (i.e., how many 1s in $b_{\max(p^*+1,t-n+1)}, \dots, b_t$)*

Figure 2 shows $\gamma$-snapshot in action, showing the 1 bits located at multiple of $\gamma = 3$ indices, the block ids, and the resulting snapshot.

As Lee and Ting [LT06b] show, a $\gamma$-snapshot has the following guarantees:

**Lemma 3.2 ([LT06b])** *Let $\mathsf{SS}_{\gamma,n}(\mathcal{S}_t) = (Q, \ell)$. Then,*

$$\mathrm{val}(\mathsf{SS}_{\gamma,n}(\mathcal{S}_t)) \overset{\mathsf{def}}{=} \gamma|Q| + \ell$$

*satisfies*

$$m \leqslant \mathrm{val}(\mathsf{SS}_{\gamma,n}(\mathcal{S}_t)) \leqslant m + 2\gamma,$$

*where m the true number of 1s in the window $\mathcal{W}_n(\mathcal{S}_t) = b_{t-n+1}, \dots, b_t$. Furthermore, $\ell < \gamma$ and $|Q| \leqslant O(m/\gamma)$.*

We represent a snapshot $\mathsf{SS}_{\gamma,n}(\mathcal{S}_t) = (Q, \ell)$ as a sequence listing the elements of $Q$ and the number $\ell$. In this representation, $\mathrm{val}(\mathsf{SS}_{\gamma,n}(\mathcal{S}_t))$ takes $O(1)$ work to evaluate because val only needs to know the size of $Q$ and the value of $\ell$. Furthermore, shrinking the window is easy in this representation: given a snapshot for a size-$n$ window, we can easily construct a snapshot for a smaller-sized window $n'$, simply by filtering out samples that are "too old" for $n'$. We have the following:

**Lemma 3.3** *There is an algorithm `shrink`$(r)$ that takes as input a $\gamma$-snapshot $\mathsf{SS}_{\gamma,n}(\mathcal{S}_t) = (Q, \ell)$ and produces a $\gamma$-snapshot $\mathsf{SS}_{\gamma,n-r}(\mathcal{S}_t)$ in $O(|Q|)$ work and $O(\log |Q|)$ depth.*

## 3.2 Maintaining Space-Bounded Counter

We present the design and analysis of a parallel space-bounded block counter, a data structure that maintains internally a $\gamma$-snapshot while taking advantage of parallelism to support its operations. We describe an interface and the corresponding guarantees in the following theorem:

**Theorem 3.4** *Let $\sigma, \lambda > 0$ be parameters and n be the window size. There is a data structure $(\sigma, \lambda)$-space bounded block counter, or a $(\sigma, \lambda)$-SBBC$(n)$, supporting operations*

- `new`$()$ *– create a new instance*
- `advance`$(T)$ *– incorporate a minibatch encoded as a CSS into the data structure*
- `query`$()$ *– retrieve a $\gamma$-snapshot for the window. Specifically, the return value of* `query` *satisfies*
  - *If it returns OVERFLOWED, then $m \geqslant \sigma \cdot \lambda$;*
  - *Otherwise, it returns $\mathsf{SS}_{\lambda/2,n}(\mathcal{S}_t)$.*
- `decrement`$(r)$ *– changes the latest r 1s to 0, effectively decrementing the counter by r (valid when the counter is not "overflowed" and the counter value is positive)*

*where m is the actual number of 1s in the window. Furthermore, the space consumption is at most $O(\min\{\sigma, m/\lambda\})$. The operation* `advance` *takes $O(\min\{\sigma, m/\lambda\} + |T|/\lambda)$ work; and* `decrement` *takes $O(m/\lambda)$ work. The other operations take constant work. Every operation has at most polylog depth (polylog in the size of the data structure and the input).*

For a SBBC $\Gamma$, let the *value* of $\Gamma$ be defined as $\mathrm{val}(\Gamma.\mathtt{query}())$ or undefined when the counter "overflows." Then, as a direct consequence of Theorem 3.4, we have the following corollary:

**Corollary 3.5** *If $\hat{m}$ is the value of a counter, then $m \leqslant \hat{m} \leqslant m + \lambda$, where m is the actual number of 1s in $\mathcal{W}_n(\mathcal{S}_t)$.*

We prove this theorem by providing an implementation. An instance of $(\sigma, \lambda)$-SBBC$(n)$ maintains a $\frac{\lambda}{2}$-snapshot and certain demarcation information. It is kept as a tuple $(t, r, \mathsf{SS} = \mathsf{SS}_{\lambda/2,r}(\mathcal{S}_t))$. The main component of this tuple is the $\frac{\lambda}{2}$-snapshot $\mathsf{SS}$. But we need the other two components track the coverage of the snapshot. Specifically, the numbers $t$ and $r$ indicate that the snapshot $\mathsf{SS}$ is $\mathsf{SS}_{\lambda/2,r}(\mathcal{S}_t)$—i.e., representing the window $\mathcal{W}_r(\mathcal{S}_t) = \{b_{t-r+1}, \dots, b_t\}$.

Importantly, even though we instruct the data structure to track a window of size $n$, it is possible that due to the capacity limit $\sigma$, the snapshot cannot cover the whole size-$n$ window; it has been truncated to a smaller window size, indicated by $r$.

As such, the operation `new` is straightforward. The operation `query` returns OVERFLOWED if $r < n$ or otherwise, returns $\mathsf{SS}$. The operation `decrement`$(r)$ updates the snapshot $\mathsf{SS}$ as follows: Suppose the snapshot is $(Q, \ell)$. If $r < \ell$, update $\mathsf{SS}$ to $(Q, \ell - r)$. Otherwise, update $\mathsf{SS}$ to $(Q', \ell')$, where $Q'$ drops the latest (i.e., the largest) $\lceil \frac{r-\ell}{\lambda} \rceil$ elements from $Q$ and $\ell' = \lambda \cdot \lceil \frac{r-\ell}{\lambda} \rceil - r$.

We devote the rest of this section to `advance` and analyzing the data structure's properties. Before we describe an implementation of `advance`, notice that in the definition of $\gamma$-snapshot, we are only interested in the positions of every $\gamma$-th one (i.e., $\omega_1, \omega_\gamma, \omega_{2\gamma}, \dots$). Therefore, if the snapshot maintained is $(Q, \ell)$, then the next 1 that would be of interest (be a multiple of $\gamma$) is the $(\frac{\lambda}{2} - \ell)$-th 1 in the stream segment being added. We implement `advance`$(T)$, where $T = (T_\ell, T_s)$ is the new stream segment encoded as CSS, by first extending $(Q, \ell)$ to a snapshot for a window of size $r + T_\ell$ then shrinking it back to a window of size $n$ or smaller (depending on the size limit $\sigma$).

Hence, to implement `advance`$(T)$, we will look at $T_s[\frac{i \cdot \lambda}{2} - \ell]$ for $i = 1, 2, \dots$. Then, We use $\beta(\cdot)$ to compute the block id's and append these to $Q$, adjusting $\ell$ accordingly. Subsequently, we use `shrink` to make the window size $n$ or some smaller $r$ if the number of entries still is at least $2\sigma + 1$.

**Work and Depth Analysis:** Clearly, both `new` and `query` take constant work and depth. The operation `decrement` can be implemented by going over the sequence $Q$ of length at most $O(m/\lambda)$, so the work is $O(m/\lambda)$ and the depth is $\log(m/\lambda)$. Moreover, the operation `advance` looks at every $\lambda/2$-th element of $T$ to determine the block id's to append to $Q$, taking $O(T_\ell/\lambda)$ work. Note that the sequence $Q$ before shrinking has length at most $\Lambda = \min\{\sigma, m/\lambda\} + T_\ell/\lambda$, so shrinking takes at most $O(\Lambda)$ work. All these operations require at most $O(\log(T_\ell + \min\{\sigma, m/\lambda\}))$ depth.

Finally, we show that if `query` returns OVERFLOWED, the actual number of 1s in the window $\mathcal{W}_n(\mathcal{S}_t)$ is at least $\sigma \cdot \lambda$. In the `advance` operation, the snapshot is trimmed to a window size smaller than $n$ if the sequence $Q$ for the window of size $n$ has at least $2\sigma + 1$ elements. By Lemma 3.2, we know that $m \geqslant \gamma(2\sigma + 1) - 2\gamma = \lambda \cdot \sigma$.

# 4. BASIC COUNTING AND SUM

As the first application of SBBC, we consider basic counting, which asks for a space-efficient data structure to estimate, at any moment, the number of 1s in either a sliding window of interest, or the stream observed thus far. While trivial in the case of infinite window, the problem is more involved in the case of sliding windows, and has been extensively studied with many applications. We present a parallel streaming algorithm for basic counting on a sliding window. For a sliding window of size $n$, this problem seeks an approximation with relative error at most $\varepsilon$ to the number of 1s in the sliding window $\mathcal{W}_n(\mathcal{S}_t)$. More specifically, we prove the following theorem:

**Theorem 4.1** *Let $\varepsilon > 0$ and $n$ be a fixed window-size. There is a parallel data structure for basic counting on a sliding window with a relative error at most $\varepsilon$ requiring $S = O(\varepsilon^{-1} \log n)$ space such that incorporating a minibatch of length $\mu$ takes $O(S + \mu)$ work and $O(\text{polylog}(S, \mu))$ depth.*

This means that if the minibatch is at least $\Omega(S)$ long, which is likely, the average per-element update cost is $O(1)$.

To prove this theorem, we will rely on the space-bounded block counters from Section 3; however, such a counter provides an additive guarantee whereas the basic counting problem seeks a relative guarantee. We need a bit more work because these types of guarantees behave differently for different ranges of values: an SBBC precise enough to $\varepsilon$-estimate a small count of 1s would consume too much space than we can afford, and a small-footprint SBBC sufficient to $\varepsilon$-estimate a large count of 1s would be too coarse for tracking a small count of 1s.

For this, we keep $O(\log n)$ different counters to cover the whole range of possible values (an integer between 0 and $n$) using a geometric scale, an idea also used in prior algorithms. Further, we set an upper bound on how much space each counter can consume. This not only controls the total space consumption but also serves to provide a coarse lower bound on the number of 1s in the window (by determining which counters have exceeded the space quota). Then, to incorporate a minibatch, we advance all the SBCCs simultaneously. To answer a query for the count, we simply look for the most precise SBBC that has not overflowed and return its estimate. We formalize this idea as follows:

Proof of Theorem 4.1. Fix $\varepsilon > 0$ and $n > 0$. Further, let $k = \min\{i \mid \varepsilon n/2^i < 1\}$. Following Lee and Ting [LT06a], we maintain $k + 1$ SBBCs $\Gamma_0, \Gamma_1, \ldots, \Gamma_k$ such that $\Gamma_i$ is a $(\sigma_i, \lambda_i)$-SBBC($n$), where $\sigma_i = 2/\varepsilon$ and $\lambda_i = \varepsilon n/2^i$. Hence, it follows directly from Theorem 3.4 that the total space requirement is $k \cdot \frac{2}{\varepsilon} = O(\varepsilon^{-1} \log n)$. Next, we consider the cost of handling a minibatch and the accuracy of our estimate.

When a minibatch $T$, $|T| = \mu$, arrives, we incorporate it by calling `advance`$(T)$ on all $k$ SBBCs in parallel. Because by Theorem 3.4, the depth of `advance` is polylogarithmic, the depth of incorporating a minibatch remains polylogarithmic. Furthermore, since the work of `advance` on $\Gamma_i$ is $O(\sigma_i + \mu/\lambda_i)$, the total work across $k$ SBCCs is

$$\sum_{i=0}^{k} \left( \frac{2}{\varepsilon} + \frac{\mu}{(n/2^i) \cdot \varepsilon} \right) \leqslant O(\varepsilon^{-1} \log n + \mu) = O(S + \mu),$$

where we note that $\sum_{i=0}^{k} \frac{1}{(n/2^i)\cdot\varepsilon} \leqslant \sum_{j \geqslant 0} 1/2^j \leqslant 2$.

To analyze the accuracy of the estimate, we first derive a lower bound on the number of 1s in the sliding window. Let $i^*$ be the largest (i.e., the finest) index $i$ where $\Gamma_i$ does not return OVERFLOWED. We note that this $i^*$ exists because the number of 1s in the window is at most $n$ which never "overflows" $\Gamma_0$ as $\sigma_0 \lambda_0 = 2n > n$. Then, $\Gamma_{i^*+1}$ returns OVERFLOWED and by the property of `query` (Theorem 3.4), the number of 1s in the window $m$ satisfies $m \geqslant \sigma_{i^*+1}\lambda_{i^*+1} = \frac{2}{\varepsilon} \cdot \frac{\varepsilon n}{2^{i^*+1}} = n/2^{i^*}$.

In addition, querying $\Gamma_{i^*}$ returns a snapshot SS where

$$m \leqslant \text{val}(\mathsf{SS}) \leqslant m + \lambda_{i^*} \leqslant m + \tfrac{\varepsilon n}{2^{i^*}}$$

by Corollary 3.5. But $m \geqslant n/2^{i^*}$, so $m \leqslant \text{val}(\mathsf{SS}) \leqslant m + \varepsilon m = (1 + \varepsilon)m$, which concludes the proof. □

## 4.1 Sum

We discuss a direct application of the basic counting data structure. Given a stream $\mathcal{S}$ of non-negative integers (i.e., $b_i \in \{0, 1, \ldots R\}$), the *sum problem* is to maintain the sum of the most recent $n$ items in the stream. This problem has been considered in the past [DGIM02, GT04], for which an algorithm for the single stream case is known: Gibbons and Tirthapura [GT04] present an $\varepsilon$-approximation to the sum using space $O(\varepsilon^{-1} \log n)$ words and constant processing time per item [2].

For incoming value $x, 0 \leqslant x \leqslant R$, let $x^i$ denote the $i$-th least significant bit in the binary representation of $x$, i.e. $x^0$ is the least significant bit, $x^1$ is the bit to the left of $x^0$, and so on. We maintain multiple basic counting data structures (Theorem 4.1), $D^i$ for $i = 0, \ldots, \log R$, where $D^i$ counts the number of 1s among the $x^i$s for all $x$ within the sliding window. A minibatch $B = s_1, s_2, \ldots, s_\mu$ of length $\mu$ is processed as follows. In parallel, we compute the binary sequences $B^1, B^2, \ldots, B^{\log R}$, where $B^i = \langle s_1^i, s_2^i, \ldots, s_\mu^i \rangle$. For $i = 1, \ldots, \log R$, in parallel, sequence $B^i$ is inserted into basic counter $D^i$. Since we are able to estimate the total number of 1-bits in each bit position $i$ to an $\varepsilon$-relative error using counter $D^i$, we are able to estimate the sum also to within an $\varepsilon$-relative error, by computing the weighted sum of the basic counts of the different data structures, where the count from $D_i$ is assigned a weight of $2^i$.

The total depth of this algorithm is of the same order as the depth of the basic counter, since the only additional operations are to extract the bit $s_j^i$ from element $s_j$, which can be done in $O(1)$ time, and to finally add the results of all data structures $D^i$s, which can be done in parallel in $O(\log \log R)$ depth. The total work of the above algorithm is $O(\log R)$ times the work of the basic counter, and the total workspace used is also $O(\log R)$ times the workspace of the parallel basic counter.

**Theorem 4.2** *Let $\varepsilon > 0$ be given and $n$ be a fixed window-size. There is a parallel data structure for continuously maintaining the sum of non-negative integers chosen from $\{0, \ldots, R\}$ on a sliding window with relative error at most $\varepsilon$ requiring $S = O(\varepsilon^{-1} \log n \log R)$ space such that incorporating a minibatch of length $\mu$ takes $O((S + \mu) \log R)$ work and $O(\text{polylog}(S, \mu))$ depth.*

[2] The space bound assumes that $\log n = \Theta(\log R)$

# 5. FREQUENCY ESTIMATION AND HEAVY HITTERS

We now consider two related problems, *frequency estimation*, and *heavy hitters*. Suppose the input is a stream of elements, where each element is an item identifier. Let $N$ denote the number of elements, and $f_e$ the number of occurrences of item $e$ in the stream so far. Given a parameter $\varepsilon, 0 < \varepsilon < 1$, the frequency estimation task is to maintain for every item $e$, an estimate $\hat{f}_e$ such that $f_e - \varepsilon N \leqslant \hat{f}_e \leqslant f_e$.

In the related problem of identifying heavy hitters from a stream, there are two parameters $\phi, 0 < \phi < 1$, a threshold for frequency, and $\varepsilon, 0 < \varepsilon < \phi$, a threshold for error, and the task is to output all elements $e$ such that $f_e \geqslant \phi N$, and output no element $e$ such that $f_e \leqslant (\phi - \varepsilon)N$. If a streaming algorithm can solve frequency estimation, then it can solve heavy hitter identification—the algorithm can simply output every element $e$ such that $\hat{f}_e \geqslant (\phi - \varepsilon)N$, and it will satisfy the conditions required for heavy hitter identification. In the remainder of this section, we focus on frequency estimation; all our results are applicable to heavy hitter identification.

## 5.1 Misra-Gries Summary

We begin by reviewing an algorithm for frequency estimation commonly known as the Misra-Gries (MG) algorithm [MG82], though it has been rediscovered at least twice [DLOM02, KSP03] since Misra and Gries proposed it in 1982. For a parameter $\varepsilon > 0$ that controls the space consumption and accuracy, the MG algorithm maintains a summary, which we will refer to as a MG summary, where up to $S = \lceil 1/\varepsilon \rceil$ items along with their counters are maintained. When a stream element $e$ arrives, MG performs the steps in Algorithm 1 to update the summary.

**Algorithm 1:** Sequential Misra-Gries

```
def update($e$):
if $e$ is an item in the summary then
    Increment the counter for $e$
else
    if the summary contains less than $S$ items then
        Add $e$ to the summary with a count of 1
    else
        Decrement all counters maintained by 1 and remove all
        counters that reach 0.
```

From this description, the space requirement of a MG summary is $O(1/\varepsilon)$ because it never keeps more than $S$ items and counters. Let $C_e$ be the value of the counter for item $e$ (it is 0 if item $e$ is not maintained in the summary). Further, let $f_e$ be the true frequency of item $e$ in the stream. It was shown that $C_e$ is a good estimate for $f_e$; we reproduce a proof of this lemma as the reasoning here will form the basis for the analysis of our parallel algorithms.

**Lemma 5.1** ([MG82, DLOM02, KSP03]) *For any item $e \in \mathcal{U}$, the estimate satisfies $f_e - \varepsilon m \leqslant C_e \leqslant f_e$, where $m$ is the length of the stream observed so far.*

Proof. Recall that we set $S = \lceil 1/\varepsilon \rceil$. We will focus on showing that $f_e - m/S \leqslant C_e$ as the relation $C_e \leqslant f_e$ follows directly from the description.

To prove this, we establish a bound on the number of times the counter for an element $e$ can possibly be decremented. If this number is $\tau$, we have that $C_e \geqslant f_e - \tau$. We give a bound on $\tau$ by considering that every time a counter is decremented, a total of at least $S$ counters, corresponding to unique items, are decremented. Viewed differently, when update($e$) decrements counters, it (virtually) deletes $S$ unique items from the stream: $e$ itself and each of the $S$ items maintained in the summary. Because there have been $m$ items in the stream so far, we know that $S\tau \leqslant m$. Hence, we have $\tau \leqslant m/S$, proving that $C_e \geqslant f_e - m/S \geqslant f_e - \varepsilon m$. □

## 5.2 Parallel Infinite Window

We describe a parallel streaming algorithm for frequency estimation based on the Misra-Gries algorithm. We prove the following theorem:

**Theorem 5.2** *Let $\varepsilon > 0$. There is a parallel algorithm for frequency estimation requiring $O(\varepsilon^{-1})$ space such that incorporating any minibatch of size $\mu$ takes $O(\varepsilon^{-1} + \mu)$ work and $O(\text{polylog}(\varepsilon^{-1}, \mu))$ depth and for any item e, it can provide an estimate $\hat{f}_e \in [f_e - \varepsilon m, f_e]$, where $f_e$ is the true frequency of e in the stream and $m = \sum_e f_e$ is the length of the stream thus far.*

Set $S = \lceil 1/\varepsilon \rceil$. Like in the sequential setting, our parallel algorithm keeps a selected set of $S$ elements and their corresponding frequency estimates. This is kept as a sequence $\langle (e_i, \hat{f}_i) \rangle_{i=1}^{S}$, where $\hat{f}_i$ is the frequency estimate of the element $e_i$. To process a minibatch, we show how to augment a MG summary with the minibatch in a way that results in a new MG summary. At the heart of our algorithm is a parallel routine MGaugment that takes as input a MG summary and a histogram of frequencies in the minibatch, and outputs a MG summary of the combined data.

Conceptually, MGaugment combines together the corresponding counters from input MG summary and the histogram. This, however, can lead to more than $S$ counters in the result. The algorithm then systematically decrements certain counters so that at most $S$ of them remain. To meet the accuracy guarantees, we make sure that each time a counter is decremented, at least $S$ unique counters are decremented together, though this has to be performed implicitly to achieve the parallelism we intended for. We formalize this idea as follows:

**Lemma 5.3** *Let $F = \langle (e_i, \hat{f}_i) \rangle_{i=1}^{S}$ be a MG summary and $H = \langle (e'_i, f'_i) \rangle_{i=1}^{p}$ be a histogram. There is an algorithm MGaugment that takes F and H, and produces a MG summary of size S combining together F and H in $O(S + p)$ work and $O(\log(S + p))$ depth. Moreover, the resulting summary still satisfies $C_e \in [f_e - \varepsilon m, f_e]$.*

Proof. We first form a sequence $H' = \langle \texttt{elt} = \cdot, \texttt{freq} = \cdot \rangle$, adding up the corresponding frequencies–i.e., $f_i$ from $F$ and $f'_i$ from $H$. This step can be accomplished in $O(S + p)$ work and $O(\log^2(S + p))$ depth using, for example, a hash table of size $O(S + p)$.

Then, we find an integer $\varphi$ such that *at most* $S$ items in $H'$ have $\texttt{freq} \geqslant \varphi$. Although $H'$ is arbitrarily ordered, computing $\varphi$ can be done in $O(|H'|)$ work and $O(\log^2 |H'|)$ depth using a variant of quick select, which can find any element of a given rank. Following that, the algorithm subtracts $\varphi$ from all the frequencies in $H'$ and returns as its output the frequencies that remain positive.

By definition of $\varphi$, it is easy to see that the output is a summary with at most $S$ items. We will now show that it has the promised accuracy guarantees. To do this, we start by observing that our process—subtracting $\varphi$ from all frequencies and retaining only the positive ones—has the same net effect as performing $\varphi$ batches ($i = 1, \ldots, \varphi$) of decrementing where the $i$-th batch decrements precisely the counters for all items whose $\texttt{freq} \geqslant i$.

Now since $i \leqslant \varphi$, we know that each batch decrements at least $S$ unique counters. Therefore, using the reasoning in Lemma 5.1, we know that the frequency in the output summary is at most $m/S$ smaller than the true frequency. Since $C_e$ is clearly at most $f_e$, we conclude that $f_e \geqslant C_e \geqslant f_e - m/S \geqslant f_e - \varepsilon m$, completing the proof. □

Using this lemma, we can derive an algorithm for Theorem 5.2 by first running `buildHist` (Theorem 2.3) on the minibatch and feeding the result to `MGaugment`. For a minibatch of length $\mu$, `buildHist` takes $O(\mu)$ work and polylog depth, so the total depth remains polylog and the total work to process this minibatch is $O(\mu + S + p) \leqslant O(\mu + S)$ because the number of distinct elements in a minibatch is at most the size of the minibatch—i.e., $p \leqslant \mu$. This proves Theorem 5.2.

## 5.3 Parallel Sliding Window

We now turn to the sliding window case, beginning with a basic parallel algorithm that meets neither the space nor work bound. Following that, we improve upon the basic algorithm to achieve the promised bounds. The main result for the sliding window case is as follows:

**Theorem 5.4** *Let $\varepsilon > 0$. Let $n \geqslant 1$ be the sliding-window size. There is a parallel algorithm for sliding-window frequency estimation requiring $O(\varepsilon^{-1})$ space such that incorporating any minibatch of size $\mu$ takes $O(\varepsilon^{-1} + \mu)$ work and $O(\varepsilon^{-1} + \mathrm{polylog}(\mu))$ depth and for any item e, it can provide an estimate $\hat{f}_e \in [f_e - \varepsilon n, f_e]$, where $f_e$ is the true frequency of e in the sliding window.*

### 5.3.1 The Basic Parallel Algorithm

We present a basic algorithm for the sliding-window case, which is a direct application of SBBC. Let $\varepsilon > 0$ be given. We set $S = \lceil 1/\varepsilon \rceil$. The algorithm is very simple: it keeps an SBBC for every item. Let $\Gamma_e$ be the $(\infty, n/S)$-SBBC$(n)$ for item $e$. Let $\mathcal{B}$ be a collection of SBBCs that the algorithm maintains. Then, we can support operations for incorporating a minibatch and querying for a frequency estimate as follows.

To query for item $e$'s frequency, we return the estimate from $\Gamma_e$ if $\Gamma_e \in \mathcal{B}$; otherwise, we return 0. To process a minibatch $T$, $\mu = |T|$, we perform the following steps:

**(1)** For each item $e$ present in $T$ or $\mathcal{B}$, create a CSS $\chi^{(e)}$ for the binary sequence $\langle \mathbf{1}_{\{T_j = e\}} : j = 1, \ldots, |T| \rangle$.

**(2)** For each item $e$ present in $T$ or $\mathcal{B}$, create $\Gamma_e$ if it does not already exist in $\mathcal{B}$ and then call `advance` with $\chi^{(e)}$.

Step 1 can be implemented in $O(\mu \log \mu)$ work and $O(\log \mu)$ depth by first marking each element of $T$ with its position and then using a parallel sort routine to gather identical items together. Then, Step 2 performs at most $O(\sum_e [\frac{n_e}{n/S} + n'_e]) = O(S + \mu) = O(\varepsilon^{-1} + \mu)$ work, where $n_e$ is the true frequency of $e$ in the sliding window before adding $T$ and $n'_e$ is the frequency of $e$ in $T$. Hence, the cost of processing a minibatch $T$ is $O(\varepsilon^{-1} + \mu \log \mu)$ work and $O(\mathrm{polylog}(\varepsilon^{-1}, \mu))$ depth.

We provide an estimate for an item $e$ by reporting

$$\hat{f}_e = \mathrm{val}(\Gamma_e.\texttt{query}()) - n/S.$$

As a direct consequence of Theorem 3.4, we know that for any item $e$, our estimate $\hat{f}_e$ satisfies $f_e - n/S \leqslant \hat{f}_e \leqslant f_e$, where $f_e$ is the true frequency in the window after adding $T$. This implies $f_e - \varepsilon n \leqslant \hat{f}_e \leqslant f_e$, as promised. Moreover, the SBBCs combined require $O(|\mathcal{B}|) + \sum_e O(\frac{f_e}{n/S}) = O(|\mathcal{B}| + \varepsilon^{-1})$ space, where we have used the fact that $\sum_e f_e = n$.

We summarize the guarantees of the basic algorithm in the following theorem:

**Theorem 5.5** *Let $\varepsilon > 0$. Let $n \geqslant 1$ be the sliding-window size. There is a parallel algorithm for sliding-window frequency estimation such that incorporating any minibatch of size $\mu$ takes $O(\varepsilon^{-1} + \mu \log \mu)$ work and $O(\mathrm{polylog}(\varepsilon^{-1}, \mu))$ depth and for any item e, it can provide an estimate $\hat{f}_e \in [f_e - \varepsilon n, f_e]$, where $f_e$ is the true frequency of e in the sliding window. Further, this algorithm requires $O(|\mathcal{B}| + \varepsilon^{-1})$ space, where $\mathcal{B}$ is the collection of SBBCs it keeps.*

We note that $\mathcal{B}$ may be as large as $\Omega(n)$, much larger than that of the best sequential algorithm for this problem, which uses only $O(\varepsilon^{-1})$ space.

### 5.3.2 First Improvement: Space

The basic parallel algorithm has two significant drawbacks: First, as already noted, it can consume more space than the best sequential algorithm, especially when there are many unique items. Second, it performs more work than the best sequential counterpart (not work efficient). In this section, we improve upon the space requirement and in the section that follows, we will improve on the work bound.

Like in the sequential case (e.g., [LT06b]), the basic idea is to track the frequencies of a selected few items approximately. To ensure that the estimate is accurate, we use the `decrement` operation of SBBCs to mimic the decrement actions in the Misra-Gries algorithm, where the counters whose count drops to zero are discarded. In essence, our algorithm is a parallelization of Lee and Ting's algorithm [LT06b].

We assume without loss of generality (WLOG) that *the minibatch is smaller than the window size n.* Otherwise, we could throw away the state and start over by looking at the most recent $n$ items in the minibatch, resetting any error accumulated.

Let $\varepsilon > 0$. Fix $S = 8/\varepsilon$ and $\lambda = \varepsilon n/4$. Our updated algorithm keeps track of $(\infty, \lambda)$-SBBC$(n)$ counters, similar to the basic algorithm (but with less error), but we will make sure that the collection $\mathcal{B}$ never grows larger than $S$. We accomplish this by adding a pruning step after Step 2 of the basic algorithm. The new step decrements a number of counters so that at most $S$ of them remain positive. We arrive at the algorithm presented in Algorithm 2.

**Algorithm 2:** A space-efficient algorithm for windowed frequency estimation.

1. For each item $e$ present in $T$ or $\mathcal{B}$, create a CSS $\chi^{(e)}$ for the binary sequence $\langle \mathbf{1}_{\{T_j = e\}} : j = 1, \ldots, |T| \rangle$.
2. For each item $e$ present in $T$ or $\mathcal{B}$, create $\Gamma_e$ if it does not already exist in $\mathcal{B}$ and then call `advance` with $\chi^{(e)}$.
3. Prune $\mathcal{B}$ by
   (a) Compute a value $\varphi$ such that *at most $S$* counters in $\mathcal{B}$ have values[3]at least $\varphi$.
   (b) For each $\Gamma_e \in \mathcal{B}$ whose value is at least $\varphi$, **in parallel**, call $\Gamma_e$.`decrement`$(\varphi)$, dropping a counter if its value goes to 0. All the other counters will just be deleted.

We analyze the updated algorithm:

**Claim 5.6** *The space-efficient version requires $O(\varepsilon^{-1})$ space. Furthermore, processing a length-$\mu$ minibatch takes $O(\varepsilon^{-1} + \mu \log \mu)$ work and $O(\mathrm{polylog}(\varepsilon^{-1}, \mu))$ depth.*

Proof. After this pruning step, the size of $\mathcal{B}$ is clearly at most $S$, so the space consumption of the updated algorithm is $O(|\mathcal{B}|) + \sum_e O(\frac{f_e}{\lambda}) = O(\varepsilon^{-1})$ space, where, again, $f_e$ is the frequency of $e$ in the window after adding $T$ and $\sum f_e = n$. Furthermore, let $\mathcal{B}'$ be

[3] Remember that the value of a counter $\Gamma$ is $\mathrm{val}(\Gamma.\texttt{query}())$.

the collection $\mathcal{B}$ before pruning, so the work in Step 3 is

$$|\mathcal{B}'| + \sum_{e:\Gamma_e \in \mathcal{B}} \frac{f_e}{\lambda} \leqslant |\mathcal{B}'| + O(\varepsilon^{-1}) = O(\mu + \varepsilon^{-1})$$

by Theorem 3.4; the depth is polylog in $\mu$ and $\varepsilon^{-1}$. The work and depth of Steps 1 and 2 are unchanged: $O(\varepsilon^{-1} + \mu \log \mu)$ work and $O(\text{polylog}(\varepsilon^{-1}, \mu))$ depth, concluding the proof. □

It remains to show that the estimate is still accurate:

**Claim 5.7** *The space-efficient version provides estimates such that for any item e, our estimate $\tilde{f}_e \in [f_e - \varepsilon n, f_e]$, where $f_e$ is the true frequency of e in the window.*

Proof. If we maintained all the counters (without decrementing or discarding), we would look up the counter $\Gamma_e$ and returns an estimate $\hat{f}_e = \text{val}(\Gamma_e.\texttt{query}()) - \lambda$ satisfying $f_e - \lambda \leqslant \hat{f} \leqslant f_e$.

The space efficient algorithm, however, decremented and removed several counters. To analyze this portion of error, we follow the line of reasoning of Lemma 5.1. Note that our decrement process can be conceptually broken down into $\varphi$ batches, where each batch decrements at least $S$ counters by exactly 1 (cf. Lemma 5.3).

Suppose the last item in the minibatch is $b_t$. We consider the counters' states *right before* the minibatch containing item $b_{t-n+1}$ arrived. Let $\Psi_0$ be the total number of 1's these counters maintained (i.e., $\Psi_0$ is the sum of all $\text{val}(\Gamma_e.\texttt{query}())$ at that point). Furthermore, let $\Psi_1$ be the number of 1's added to counters created or advanced after that point. We establish that $\Psi_0 \leqslant (\sum_e f'_e) + S \cdot \lambda \leqslant 3n$ because there can be at most $S$ counters (due to pruning). Also, we know that $\Psi_1 \leqslant 2n$ because there are $n$ elements between $b_{t-n+1}$ and $b_t$ and the minibatch containing $b_{t-n+1}$ itself has size at most $n$, as we had assumed WLOG. Hence, we can upper bound the error due to counter decrementing for any element $e$ as $(\Psi_0 + \Psi_1)/S \leqslant 5n/S = \frac{5}{8}\varepsilon n < \frac{3}{4}\varepsilon n$. Hence, the total error is at most $\frac{3}{4}\varepsilon n + \lambda = \varepsilon n$. □

Therefore, with the space improvement, we have:

**Theorem 5.8** *Let $\varepsilon > 0$. Let $n \geqslant 1$ be the sliding-window size. There is a parallel algorithm for sliding-window frequency estimation such that incorporating any minibatch of size μ takes $O(\varepsilon^{-1} + \mu \log \mu)$ work and $O(\text{polylog}(\varepsilon^{-1}, \mu))$ depth and for any item e, it can provide an estimate $\hat{f}_e \in [f_e - \varepsilon n, f_e]$, where $f_e$ is the true frequency of e in the sliding window. Further, this algorithm requires $O(\varepsilon^{-1})$ space.*

### *5.3.3 Second Improvement: Work*

With the space improvement, the space bound now matches that of the best sequential algorithm. We now turn to the work aspect. The previous algorithm (Algorithm 2) would have been work efficient if we had been able to construct all CSSs $\chi^{(e)}$ in linear work in parallel. We, however, do not know how to accomplish that. Instead, we observe that if we can predict which counters will survive after the pruning step, we will not have to construct sequences for counters that will be thrown away in the end. Building on this idea, we devise an algorithm for determining which counters will be kept. Then, we give a linear-work parallel algorithm that constructs CSS $\chi^{(e)}$ for the items that will be kept, though this latter algorithm has depth $O(\varepsilon^{-1})$, which is linear in the number of counters to be kept.

**Predicting Survivors:** We describe `predict`, an algorithm for computing the set $K$ of items that will be retained after the pruning step, as well as the pruning "cut off" $\varphi$. Before we give the description, keep in mind that we have assumed WLOG that each minibatch is smaller than the window size $n$. First, we construct a histogram $H$ for the minibatch; this requires at most $O(n)$ work and polylog depth. Then, we read off the values of the existing SBBCs after "shrinking" the window to evict elements too old for the new window (using $\text{val}(\texttt{shrink}(\Gamma.\texttt{query}))$). It is easy to see that adding the corresponding counts together gives us the counts if we were to perform Steps 1–2 in the previous algorithm.

With these counts calculated, we can easily compute the cutoff $\varphi$ in Step (3a) of the previous algorithm (see Lemma 5.3 for an algorithm). Using $\varphi$, we know how much the surviving counters must be decremented by and which counters will be retained.

All these can be performed in $O(1/\varepsilon + \mu)$ work: `buildHist` takes linear work; the work performed by `shrink` across all counters is $\sum_{e:\Gamma_e \in \mathcal{B}} f_e/\lambda = O(1/\varepsilon)$; and computing $\varphi$ takes at most $O(1/\varepsilon + \mu)$ work. Further, all these operations have $\text{polylog}(\mu, 1/\varepsilon)$ depth.

**Constructing Selected Sequences:** Given a set $K$ of items that we will retain after processing the minibatch, we describe an algorithm to construct the CSS $\chi^{(e)}$ for all $e \in K$ simultaneously:

**Lemma 5.9** *There is an algorithm* $\texttt{sift}(T, K)$ *that takes a stream segment $T = \langle a_1, a_2, \ldots, a_{|T|} \rangle$ and an index set K, and produces $|K|$ sequences $\{\chi^{(\kappa)}\}_{\kappa \in K}$ such that $\chi_i^{(\kappa)}$ is the CSS representing the binary sequence $\langle \mathbf{1}_{\{a_j = e\}} : j = 1, \ldots, |T| \rangle$. Moreover,* `sift` *runs in $O(|T| + |K|)$ work and $O(|K| + \log(|K| + |T|))$ depth.*

Proof. We describe an algorithm consisting of two steps: First, it produces a subsequence $s'$ of $T$ containing only elements that are in $K$, where each element of $s'$ is tagged with its original position in $T$. For $i = 1, \ldots, |s'|$, let $p_i$ denote the position of the element $s'_i$ in $T$. This can be accomplished in $O(|T|)$ work and $O(\log |T|)$ depth using standard techniques [JáJ92]. Then, we derive the sequences for CSS $\chi^{(\kappa)}$'s as follows.

Let us observe that given the sequences $s'$ and $p$, we can easily derive the sequences $\{\chi^{(\kappa)}\}_{\kappa \in K}$ in $O(|s'|)$ work and $O(|s'|)$ depth using sequential radix sort, which is stable. To parallelize this process, we divide $s'$ (and $p$) into $|s'|/|K|$ equal-sized pieces (each is $|K|$ long) and run the sequential algorithm on these pieces *in parallel*. Each of these pieces requires $O(|K|)$ work and $O(|K|)$ depth, resulting in $O(\frac{|s'|}{|K|} \cdot |K|) = O(|s'|)$ work and $O(|K|)$ depth for the all pieces combined. At this point, each piece has been converted into $|K|$ sequences, one for each $\kappa \in K$. To generate the final output, we concatenate sequences of the same $\kappa$ together, preserving order. For each $\kappa$, while the depth is bounded by $O(\log |s'|)$, the work is $O(|s'|/|K| + \ell_\kappa)$, where $\ell_\kappa$ is the length of the final sequence $\chi^{(\kappa)}$. The $|s'|/|K|$ term is due to a prefix-sum operation to determine the position in the output, and the $\ell_\kappa$ reflects the cost to write a sequence of that length. Since there are $|K|$ values of $\kappa$, we have that the work for concatenation is $O(|K|\frac{|s'|}{|K|} + \sum_{\kappa \in K} \ell_\kappa) = O(|s'| + |K|) \leqslant O(|s| + |K|)$, which concludes the proof. □

Using the output from the routine `predict` for predicting survivors (i.e., $K$ and $\varphi$), we call $\texttt{sift}(T, K)$ to generate $\chi^{(e)}$ for all $e \in K$. Following that, we update the counters by calling `advance` with $\chi^{(e)}$ (we create it if it did not exist) and subsequently calling `decrement` with $\varphi$ for all $e \in K$. Finally, we delete all existing counters that do not exist in $K$.

Because this algorithm simulates the effects of the previous algorithm, the space bound and accuracy guarantees follow from the previous analysis. It remains to analyze the work and depth of this algorithm. We have shown that `predict` takes $O(1/\varepsilon + \mu)$ work and $\text{polylog}(1/\varepsilon, \mu)$ depth. Also, the set $K$ has size at most $O(1/\varepsilon)$. So then, $\texttt{sift}(T, K)$ takes $O(\mu + 1/\varepsilon)$ work and $O(1/\varepsilon + \log(\mu))$ depth. Finally, the combined work performed by `advance` and `decrement` is at most $O(1/\varepsilon)$, as analyzed before. This proves Theorem 5.4.

### *5.3.4 Lower Bound*

We show a lower bound on the work required to identify all heavy hitters from a stream in the infinite window case, proving work optimality of our algorithm.

**Lemma 5.10** *On a stream of total length $N$, any deterministic algorithm that outputs all items with frequency $\phi N$ or greater, and no item with frequency less than $(\phi - \epsilon)N$ must have work $\Omega(N)$.*

Proof. We argue that every deterministic algorithm that has the above property must examine $\Omega(N)$ elements of the stream. We prove this using contradiction; suppose there was a deterministic algorithm $\mathcal{A}$ with the above property that examined less than $(1 - \phi)N$ stream elements.

Consider an input $I_1$ where the algorithm output the set $S_1$. We construct another input $I_2$ as follows. Let $y$ be an item that does not belong to $S_1$; such an element $y$ must exist since the algorithm cannot output all elements as a part of $S_1$, and every element that is output must have a minimum frequency. $I_2$ is identical to $I_1$, except that in every position of $I_1$ that was not examined by $\mathcal{A}$, $I_2$ contains $y$. The frequency of $y$ in $I_2$ is at least $\phi n$, and hence $y$ must be output by $\mathcal{A}$ upon observing input $I_2$. However, the behavior of $\mathcal{A}$ on input $I_2$ must be identical to its behavior on input $I_1$, since every input element examined by $\mathcal{A}$ is identical in both cases. Hence, $\mathcal{A}$ will not output $y$ on input $I_2$, which contradicts the assumption that $\mathcal{A}$ correctly outputs all items with a frequency $\phi n$ or greater. Hence, the algorithm $\mathcal{A}$ must examine at least $(1 - \phi)N$ elements of the input, and this takes $\Omega(N)$ time. □

Note that the above lower bound does not hold for randomized algorithms which allow a (small) probability of not satisfying the requirements of heavy hitter identification. Indeed, there are algorithms for identifying heavy hitters by examining only a random sample whose size is much smaller than the length of the stream [MM02, EV03].

We also note that the same argument in Lemma 5.10, of the algorithm needing to examine a majority of the input, also applies to frequency estimation; hence, the linear lower bound on work also applies to frequency estimation. The following corollary follows by observing that if $\mu = \Omega(1/\varepsilon)$, then Theorem 5.2 gives a $O(\mu)$ work bound:

**Corollary 5.11** *Our parallel algorithm for frequency estimation and heavy hitter identification for infinite window (Theorem 5.2) is work-optimal if the batch size $\mu$ is $\Omega(\frac{1}{\varepsilon})$.*

## 5.4 Comparison with Independent Data Structure Approach

We compare our algorithm for frequency estimation and heavy hitters with a parallel algorithm based on the independent data structure approach. As described in [ACH+13], there is a mergeable data structure for approximate heavy hitter identification that takes space $O(1/\varepsilon)$ per processor. Suppose there are $p$ processors; the input stream $\mathcal{S}$ is partitioned into sub-streams $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_p$, one per processor. Processor $i$ processes $\mathcal{S}_i$ and maintains a local data structure $D_i$. Since the $D_i$s are mergeable, it is possible to construct the data structure $D$ for $\mathcal{S}$ given only $D_1, D_2, \ldots, D_p$, without access to the individual streams $\mathcal{S}_1, \mathcal{S}_2, \ldots$. Hence, the algorithm can simply send all the $D_i$s to a single processor who merges them to construct $D$, which can be used to answer queries about frequency estimation and heavy hitters on $\mathcal{S}$.

The total memory taken by the parallel algorithm across all processors is $O(p/\varepsilon)$. Note that this is a factor of $p$ worse than the sequential algorithm, as well as $p$ times worse that our parallel algorithm.

Further, merging two data structures $D_i$ and $D_j$ from processors $i$ and $j$ is a sequential operation, and the time taken to merge all $p$ summaries at query time can be $O(p/\varepsilon)$, if done at a single processor. Even if merging is done by organizing the processors into a $\log p$-deep hierarchy, the depth of this parallel algorithm is $\Omega(\varepsilon^{-1} \log p)$. With the approach of independent data structures, it seems hard to overcome this bottleneck, and achieve work-optimality and depth that is polylog in $\varepsilon^{-1}$.

## 6. COUNT-MIN SKETCH

In this section, we describe another application of the techniques developed so far in this paper. We devise a parallel version of the Count-Min (CM) sketch [CM05]. In the sequential setting, CM sketch has proved to be a versatile summary for frequency-based properties of a stream that can be used for answering a variety of queries on the input stream, including point and range queries, quantiles, and heavy hitters, among others.

Unlike the other algorithms considered in this work, the CM sketch gives a notion of probabilistic guarantee, namely an $(\varepsilon, \delta)$-approximation, where the quantity being reported has relatively error at most $\varepsilon$ with probability at least $1 - \delta$.

For $\varepsilon, \delta > 0$, the CM sketch algorithm maintains a 2-d array $A$ with $d = \lceil \ln(1/\delta) \rceil$ rows and $w = \lceil e/\varepsilon \rceil$ columns, together with $d$ hash functions $h_1, h_2, \ldots, h_d$ chosen from a family of pair-wise independent hash functions. The array $A$ is initially all 0. The sequential CM algorithm answers a query about an item $e$ by reporting

$$a_e = \min \{A[i, h_i(e)] \mid i = 1, \ldots, d\}.$$

Moreover, when an element $e$ shows up in the stream, it updates $A$ by going through $i = 1, \ldots, d$, adding 1 to $A[i, h_i(e)]$. As Cormode and Muthukrishnan show, this gives the guarantee that with probability at least $1 - \delta$, $a_e \leqslant f_e + \varepsilon m$, where $f_e$ is the true frequency of item $e$ and $m$ is the length of the stream so far.

**A Parallel Implementation:** Instead of updating $A$ each time a new stream element shows up, we observe that if the same item $e$ shows up $k$ times, they will all be hashed to the same locations and the count at that location will be incremented by $k$. Like before, we work in minibatches. When a minibatch $T$ arrives, we use the `buildHist` algorithm to compute a histogram of frequencies $H = \langle(\mathtt{elt} = \cdot, \mathtt{freq} = \cdot)\rangle$. For each $(\mathtt{elt}, \mathtt{freq}) \in H$, we increment $A[i, h_e(\mathtt{elt})]$ by `freq` for $i = 1, \ldots, d$ simultaneously in parallel.

Simultaneously incrementing these counters, however, requires some care since some of the locations may be shared by different items. To increment them in parallel, we gather for each row, the frequencies that hash to the same column. This can be done in $O(\mu + w)$ work and $\text{polylog}(\max\{\mu, w\})$ depth using parallel integer sort [RR89], where we note that the hash values are $\{1, \ldots, w\}$. Hence, processing a minibatch of size $\mu$ takes $O(\mu + (\mu + w)d)$ work and $O(\text{polylog}(\mu, w))$ depth—or if we assume $\mu = \Omega(w)$, then we have $O(\log(1/\delta))$ work per item on average.

To answer a query about $e$, we report $a_e = \min\{A[i, h_i(e)] \mid i = 1, \ldots, d\}$ as before, but we compute min in parallel using a reduce operation (which for length-$p$ data has $O(p)$ work and $O(\log p)$ depth in the length of the data. Hence, a query costs $O(d) = O(\ln(1/\delta))$ work and $O(\log\log(1/\delta))$ depth.

We summarize the results as follows:

**Theorem 6.1** *There is a data structure for maintaining the count-min sketch in space $O(\varepsilon^{-1} \log(1/\delta))$ such that incorporating a minibatch of size $\mu$ takes $O(\log(\frac{1}{\delta}) \max\{\mu, \frac{1}{\varepsilon}\})$ work and $O(\text{polylog}(\mu, \frac{1}{\varepsilon}))$ depth, and a query about any element takes $O(\log(1/\delta))$ work and $O(\log\log(1/\delta))$ depth.*

# 7. CONCLUSION

We have presented parallel algorithms for maintaining frequency-based aggregates on a high-velocity stream. The aggregates considered include heavy hitters, basic counting, frequency estimation, sum, and the count-min sketch. These algorithms perform linear-work (i.e., constant work per element on average) and have low depth. These are the first parallel algorithms for the problems that are provably work-optimal and low-depth.

## Acknowledgments

This research was in part sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program, Agreement Number W911NF-12-C-0028. Tirthapura was partially supported by the National Science Foundation, through grant numbers 0834743 and 0831903. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Defense Advanced Research Projects Agency, National Science Foundation, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

## References

[ABC09] Chrisil Arackaparambil, Joshua Brody, and Amit Chakrabarti. Functional monitoring without monotonicity. In *ICALP '09*, pages 95–106, 2009.

[ACH$^+$13] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Trans. Database Syst.*, 38(4), 2013.

[BO10] Vladimir Braverman and Rafail Ostrovsky. Effective computations on sliding windows. *SIAM J. Comput.*, 39(6):2113–2131, 2010.

[CCFC02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[CH10] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *VLDB J.*, 19(1):3–20, 2010.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[CMYZ12] Graham Cormode, S. Muthukrishnan, Ke Yi, and Qin Zhang. Continuous sampling from distributed streams. *J. ACM*, 59(2), 2012.

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

[CT11] Massimo Cafaro and Piergiulio Tempesta. Finding frequent items in parallel. *Concurrency and Computation: Practice and Experience*, 23(15):1774–1788, 2011.

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

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6), 2002.

[DLOM02] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *ESA*, pages 348–360, 2002.

[EV03] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[GT01] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA'01*, pages 281–291, 2001.

[GT04] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. *Theory of Computing Systems*, 37:457–478, 2004.

[HLT10] Regant Y. S. Hung, Lap-Kei Lee, and Hing-Fung Ting. Finding frequent items over sliding windows with constant update time. *Inf. Process. Lett.*, 110(7):257–260, 2010.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[LT06a] Lap-Kei Lee and H. F. Ting. Maintaining significant stream statistics over sliding windows. In *SODA*, pages 724–732, 2006.

[LT06b] Lap-Kei Lee and H. F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *PODS'06*, pages 290–297, 2006.

[MAE06] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An integrated efficient solution for computing frequent and top-*k* elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, 2006.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[RR89] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[TW11] Srikanta Tirthapura and David P. Woodruff. Optimal random sampling from distributed streams revisited. In *Proc. International Symposium on Distributed Computing (DISC)*, pages 283–297, 2011.

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

[ZDL$^+$13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.

[ZSZ$^+$13] Yu Zhang, Yue Sun, Jianzhong Zhang, Jingdong Xu, and Ying Wu. An efficient framework for parallel and continuous frequent item monitoring. *Concurrency and Computation: Practice and Experience*, 2013.