ABSTRACT

We revisit the classic basic counting problem in the distributed streaming model that was studied by Gibbons and Tirthapura (GT). In the solution for maintaining an $(\epsilon, \delta)$-estimate, as what GT’s method does, we make the following new contributions: (1) For a bit stream of size $n$, where each bit has a probability at least $\gamma$ to be 1, we exponentially reduced the average total processing time from GT’s $\Theta(n \log(1/\delta))$ to $O(1/(\gamma^2))(\log^2 n \log(1/\delta))$, thus providing the first sublinear-time streaming algorithm for this problem. (2) In addition to an overall much faster processing speed, our method provides a new tradeoff that a lower accuracy demand (a larger value for $\epsilon$) promises a faster processing speed, whereas GT’s processing speed is $\Theta(n \log(1/\delta))$ in any case and for any $\epsilon$. (3) The worst-case total time cost of our method matches GT’s $\Theta(n \log(1/\delta))$, which is necessary but rarely occurs in our method. (4) The space usage overhead in our method is a lower order term compared with GT’s space usage and occurs only $O(\log n)$ times during the stream processing and is too negligible to be detected by the OS in practice. We further validate these solid theoretical results with experiments on both real-world and synthetic data, showing that our method is faster than GT’s by a factor of several to several hundreds depending on the stream size and accuracy demands, without any detectable space usage overhead. Our method is based on a faster sampling technique that we design for boosting GT’s method and we believe this technique can be of other independent interest.

1. INTRODUCTION

Advances in modern science and technology have given rise to massive data (or so-called big data). Some of the data naturally arrives as streams. Examples include network data packets passing through a router, environmental data collected by sensor networks, and search requests received by search engines. In many cases, such massive streaming data need to be monitored in a real-time fashion. Such data processing requirements make conventional methods such as storing them in a relational database and issuing SQL queries thereafter infeasible, and thus brings up the phenomenon of data stream processing [2, 20]. In data stream processing, the workspace is often orders of magnitude smaller than the stream size, requiring the data be processed in one pass.

However, most streaming algorithms need to look at every data element at least once [1, 4, 7, 8, 9, 11, 16] (see [2, 20] for many other example references). In some cases where extremely fast paced streaming data is involved, even a single glance at every stream element can be unaffordable. For example, a typical OC48 link transfers 2.5 Gbits per second and AT&T backbone networks carry over 15 petabytes of data traffic on an average business day. Deploying a streaming algorithm for monitoring purpose to process every data element in such massive data streams is very computationally expensive and can greatly hurt the performance of the system. The goal of sublinear-time algorithms is to solve computational problems without having to look at every input data element. However, in sublinear time algorithm design, the input data is often stored statically [12, 17, 22, 23], meaning we can visit any part of the input data at any time if needed.

In this paper, we demonstrate that designing a sublinear-time algorithm for streaming data is also possible, without losing accuracy guarantee compared with linear-time streaming algorithms. In particular, we proposed the first streaming algorithm for the distributed basic counting problem using time sublinear of the stream size in the average case. To our best knowledge, the best prior result [14] for solving this problem has to visit every stream element at least once and thus needs a time cost at least linear of the stream size in any case.

1.1 Distributed basic counting

Alice and Bob (called processors) are processing in parallel two geographically distributed bit streams $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, respectively. In each stream, the $i$th bit is received before the $j$th bit, if $i < j$. Upon receiving a query, the referee, who is located on another remote site, wants to know the number of 1-bits in the union of the two bit streams.

$$U(A, B) = \sum_{i=1}^{n} (a_i \lor b_i)$$

where $\lor$ is the bit-wise logical OR operator and $n$ is the number of bits that Alice and Bob have both received when the query arrives at the referee. Note that both streams evolve...
over time and thus the stream size $n$ and the value of $U$ monotonically increase over time. The constraints and challenges in the computation of $U$ are: (1) no direct communication between processors is allowed since there are no direct connection between Alice and Bob, (2) use small workspace on the processors as well as on the referee, and (3) use small communication cost on the links connecting the processors and the referee.

The problem can be generalized to $k$ streams $R_1, R_2, \ldots, R_k$, processed by $k$ processors respectively, for some constant $k \geq 2$. For $j = 1, 2, \ldots, k$, we write the stream $R_j$ in the form of $\{r_{j,1}, r_{j,2}, \ldots, r_{j,n}\}$. Upon receiving a query, the referee wants to know the number of 1-bits in the bit-wise OR of the $k$ streams:

$$U(R_1, R_2, \ldots, R_k) = \sum_{i=1}^{n} (r_{1,i} \lor r_{2,i} \lor \ldots \lor r_{k,i}) \quad (1)$$

The same constraints and challenges for the 2-stream case hold in this general setting. Figure 1 shows the system setting that is assumed in the distributed basic counting. Because our method for the 2-stream case can be easily extended for the general setting, we will focus on the 2-stream case in our presentation. The extension for the general setting will be presented in the end.

We refer readers to [14] for a detailed discussion on the extensive applications of the distributed basic counting in large-scale data aggregation and monitoring.

1.2 Prior work

A naive method for the referee to maintain the knowledge of $U$ is to get Alice and Bob to continuously forward their stream elements to the referee. The referee will then simply do a straightforward calculation of $U$ in one pass of the two streams, using $O(\log n)$ bits of workspace at both the processors and the referee. However, this approach introduces a high communication cost between the processors and the referee, which is prohibited in many applications such as network monitoring. To reduce the communication cost, Gibbons and Tirthapura proposed a communication-efficient distributed computing scheme [14], where Alice and Bob each maintains a small-space data structure (a.k.a. sketch) without communicating neither to each other nor to the referee over the course of stream processing. When the query arrives, the referee will first notify Alice and Bob to send their sketches to the referee. The referee will then retrieve the knowledge of $U$ from the sketches. However, under this distributed computing setting, $O(n)$ bits of communication cost is necessary to get the exact value of $U$ even for randomized algorithms [19]. It is also shown that $O(\sqrt{n})$ bits of workspace is necessary at each processor even for an approximate answer for $U$ with a relative error bound, if Alice and Bob sample their streams independently [14]. In order to achieve a solution of both workspace and communication efficiency, Gibbons and Tirthapura proposed the coordinated adaptive sampling (a.k.a. distinct sampling) technique that uses only $O((1/\epsilon^2) \log(1/\delta) \log n)$ bits of workspace at each processor and the referee and $O((1/\epsilon^2) \log(1/\delta) \log n)$ bits of communication cost per link and per query. By using these sublinear (of stream size) space and communication cost, their technique guarantees an $(\epsilon, \delta)$-estimate for $U$ [14]. Their algorithm can be trivially extended to provide an $(\epsilon, \delta)$-estimate of $U$ over multiple streams with the same aforementioned workspace and communication cost.

**Definition 1.1.** Given the parameters $\epsilon$ and $\delta$, $0 < \epsilon, \delta < 1$, the $(\epsilon, \delta)$-estimate of a nonnegative variable $X$ is a random variable $\hat{X}$, such that:

$$\Pr \left[ |\hat{X} - X| \leq \epsilon X \right] \geq 1 - \delta$$

In particular, the $(\epsilon, 0)$-estimate is also called $\epsilon$-estimate.

**Definition 1.2.** ($\gamma$-random bit stream.). A bit stream $\{a_1, a_2, \ldots, a_n\}$ is a $\gamma$-random bit stream, if all the bits in the stream are mutually independent and $\Pr[a_i = 1] \geq \gamma$, for $i = 1, 2, \ldots, n$, where $0 < \gamma < 1$ is a constant.

The notion of $\gamma$-random bit stream reveals the sparseness and randomness of a bit stream. It requires the minimum statistical characteristic of a stream and thus can fit to capture the distribution of many real-world streams. For example: (1) Well coded and compressed online video, audio, and image streams would have roughly equal number of 1’s and 0’s (otherwise the stream should and can be better coded and compressed.), so they can be roughly viewed as $\gamma$-random streams, where $\gamma \approx 0.5$. (2) A network traffic stream can be roughly viewed as a $\gamma$-random stream, if $\gamma$ portion of the bits in the traffic are 1-bits. (3) A sensor in a sensor network generates a 1-bit to report an event and a 0-bit if nothing happened during a fixed amount of time. If the probability of an event is $\gamma$, the event bit stream can be roughly viewed as a $\gamma$-random stream.

1.3 Our contribution

We designed a novel sampling technique that enables us to sample the stream without having to check every stream element. By using this fast sampling technique, we are able to boost GT’s processing speed in maintaining an $(\epsilon, \delta)$-estimate of the distributed basic counting with negligible extra space usage. Table 1 summarizes the performance comparison of our method and GT’s.

1. The average total processing time for a $\gamma$-random bit stream is reduced from GT’s $\Theta(n \log 1/\delta)$ to $O((1/\epsilon^2) \log^2 n \log 1/\delta)$. Our method not only exponentially improves the overall processing speed, providing the first sublinear-time algorithm in the average case, but also provides a new tradeoff that a lower accuracy demand (a larger value for $\epsilon$) promises a faster processing speed, whereas GT’s method spends $\Theta(n \log 1/\delta)$ time regardless of the accuracy demand.

2. Our method’s worst-case total processing time matches GT’s $\Theta(n \log(1/\delta))$, which is necessary. However, this worst-case time cost rarely occurs with our method, whereas GT’s method always needs $\Theta(n \log(1/\delta))$ time in any case.

3. Each processor uses $O((1/\epsilon^2) \log(1/\delta) \log n)$ bits of workspace in our method. Compared with GT’s space usage of $O((1/\epsilon^2) \log(1/\delta) \log n)$ bits, our method’s extra space...
usage is a lower-order term for any real-world bit stream and a reasonably small $\epsilon$ (say $\epsilon \leq 0.1$) and is indeed undetectable in our experiments with both real-world and synthetic data. Further, this extra space cost occurs only $O(\log n)$ times in average during the stream processing.

4. The workspace at the referee and the communication cost between the processors and the referee in our method remains the same as GT's.

5. We conducted a comprehensive experimental study using both real-world and synthetic data. All experimental results show that our method is faster than GT's by a factor of several to several hundreds depending on the stream size and accuracy demand. Our method can potentially save the vast majority of the processing time and energy that is consumed by GT's method in the real-world deployment, where the stream size can be nearly unbounded. All experiments also show that the OS does not detect any extra space cost used by our method compared with GT's.

6. The fast sampling technique we proposed can also be of other independent interest in the design of sampling-based algorithms solving other problems.

1.4 Paper organization

After a survey of related work, we will present a high-level overview of our method in Section 3, where we will introduce the structure of the coordinated adaptive sampling and show the opportunity for boosting. We will then introduce the new sampling technique for boosting in Section 4. By plugging the faster sampling technique into the coordinated adaptive sampling, we are able to present the complete picture of our method in Section 5. The details of a comprehensive experimental study using both real-world and synthetic data are given in Section 6. This paper is concluded by Section 7.

2. RELATED WORK

In this section, we summarize the results on basic counting in the streaming model under various constraints and settings. For a broader overview of stream processing, we refer readers to the surveys [2, 20].

While the small-space basic counting of a whole single stream is trivial, it becomes much harder when counting the number of 1-bits on the union of multiple geographically distributed streams. It is shown in [19] that $\Omega(n)$ bits of workspace is necessary for an exact answer even for randomized algorithms, where $n$ is the size of each single stream. Datar et al. [11] considered the basic counting on a single stream, given the constraint of sliding windows. It is easy to show $\Omega(N)$ bits is required to maintain the exact knowledge of how many 1-bits in a sliding window of size $N$, representing the most recently received $N$ bits. They further showed a space lower bound of $\Omega((1/\epsilon)\log^2(\epsilon N))$ bits for both deterministic and randomized algorithms for maintaining an $\epsilon$-estimate of the basic counting over a sliding window of size $N$. They also proposed an exponential histogram based deterministic algorithm that guarantees an $\epsilon$-estimate using workspace matching the above space lower bound, but their algorithm cannot work for the union of multiple bit streams under the sliding windows model. Gibbons and Tirthapura [15] solved this new challenge by extending and applying their technique from [14] to the sliding window setting over multiple streams. Instead of maintaining one sample as was done in [14] for the case without sliding window, they maintain $\log N^{-1}$ samples where every sample has a different sample probability from $1, 1/2, \ldots, 1/\log N$, because the number of 1-bits in the sliding window can vary over time depending upon the input stream and thus one fixed good sample probability cannot be predetermined beforehand. By maintaining multiple samples, their technique is able to pick the best samples with the right sample probabilities from the multiple processors at the query time, so that the referee is guaranteed to have a good estimate for the basic counting on the union of the bit streams over the sliding window.

Xu et al. [26] considered the sliding window based basic counting on an asynchronous stream, where the arrival order of the stream elements is not necessarily the same as the order at which they were created. Such an asynchronous stream model is motivated by the real-world scenarios such as network data packets being received out of order at the destination due to the network delay and multi-path routing. The core idea of their solution is mostly identical to the one from [15] but is modified for asynchronous streams. It is also easy to extend their solution so that it can work for the union of multiple streams. Busch and Tirthapura [5] also solved the asynchronous stream basic counting problem over sliding windows for one stream. Their solution is deterministic and is based on a novel data structure called splittable histogram, but it is not clear how to extend their method to multiple streams.

The coordinated random sampling technique [14] can also be used for counting $F_0$ [1, 4, 18], the number of distinct elements, over one or the union of multiple data streams, with or without sliding windows [13, 14, 15]. Pavan and Tirthapura [21] generalized the data stream model in the calculation for $F_0$. In their stream model, every stream element is no longer a single integer but is a range of continuous integers. The $F_0$ is defined as the number of distinct integers in the union of the ranges that have been received in the stream. A trivial solution is to expand each range into a sequence of integers and use an existing $F_0$ algorithm to process each integer. The time cost for processing a range will then be at least linear of the range size, which is not acceptable when the range size is large. They proposed a divide and conquer based strategy, such that the time cost for processing each range is only a logarithm of the range size. Part of the idea behind our new sampling technique presented in this paper is inspired by their work.

All these related work need to observe each stream element at least once, leading to their time costs to be at least linear of the stream size.

3. A HIGH-LEVEL OVERVIEW

The high-level structure of our method is the coordinated adaptive sampling by Gibbon and Tirthapura [14], but uses a different hash function for the random sampling. We exploit the properties of the hash function, so that we can do random sampling over the data stream without having to check every stream element.

3.1 Coordinated adaptive random sampling

Alice and Bob use the same sampling procedure and the sample size, so we will only describe the behavior of Alice.  

\footnote{In this paper, $\log = \log_2$, unless specified explicitly.}
<table>
<thead>
<tr>
<th></th>
<th>GT’s [14]</th>
<th>Ours</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case total time cost</td>
<td>$\Theta(n \log \frac{1}{\delta})$</td>
<td>$\Theta(n \log \frac{1}{\delta})$</td>
<td>Always occurs with GT’s, but rarely occurs with ours.</td>
</tr>
<tr>
<td>Worst-case per-item time cost</td>
<td>$\Theta\left(\log n \log \frac{1}{\delta}\right)$</td>
<td>$\Theta\left(\log n \log \frac{1}{\delta}\right)$</td>
<td>significantly improved</td>
</tr>
<tr>
<td>Avg.-case total time cost if a $\gamma$-random bit stream</td>
<td>$\Theta\left(n \log \frac{1}{\delta}\right)$</td>
<td>$O\left(\frac{\gamma}{\delta^2} \log^2 n \log \frac{1}{\delta}\right)$</td>
<td>when $n$ is large.</td>
</tr>
<tr>
<td>Avg.-case per-item time cost if a $\gamma$-random bit stream</td>
<td>$\Theta\left(\log \frac{1}{\delta}\right)$</td>
<td>$O\left(\frac{1}{\delta^2} \log^2 n \log \frac{1}{\delta}\right) = o(1)$, negligible overhead</td>
<td></td>
</tr>
<tr>
<td>Space cost per-processor (bits)</td>
<td>$O\left(\frac{\gamma}{\delta} \log \frac{1}{\delta} \log n\right)$</td>
<td>$O\left(\frac{\gamma}{\delta} \log \frac{1}{\delta} \log n\right)$</td>
<td></td>
</tr>
<tr>
<td>Space cost by the referee (bits)</td>
<td>$O\left(\frac{\gamma}{\delta} \log \frac{1}{\delta} \log n\right)$</td>
<td>$O\left(\frac{\gamma}{\delta} \log \frac{1}{\delta} \log n\right)$</td>
<td></td>
</tr>
<tr>
<td>Comm. cost per query (bits)</td>
<td>$O\left(\frac{\gamma}{\delta} \log \frac{1}{\delta} \log n\right)$</td>
<td>$O\left(\frac{\gamma}{\delta} \log \frac{1}{\delta} \log n\right)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The performance comparison between GT’s method and ours.

Random sampling. Alice maintains a sample of some known size $\alpha$, which will be determined later. She randomly selects each 1-bit (by storing the bit’s stream location index) into the sample with some probability $p$. After processing the stream, the number of 1-bits selected into the sample multiplied by $1/p$ can be a good estimate of the number of 1-bits in the stream, if $\alpha$ is large enough.

Adaptive random sampling. However, Alice does not know in advance how many 1-bits will be present in her stream and thus cannot decide an appropriate sample probability $p$. If the sample probability is too high, the sample size may not be big enough to accommodate all the selected 1-bits; if the sample probability is too small, the sample may not select enough 1-bits to yield a good estimate. To overcome this difficulty, Alice adaptively changes the sample probability over the course of her stream processing. The sample probability is determined by the sample level $\ell$, which starts from 0 and increases towards 1, 2, … . At sample level $\ell$, every incoming 1-bit is selected into the sample with sample probability $P_\ell = 1/2^\ell$ (In our method, $P_\ell$ is not exactly but is nearly equal to $1/2^\ell$ and will be clear later). Alice is always aware of her current sample probability by remembering her current sample level $\ell$.

Coordinated adaptive random sampling. However, Gibbons and Tirthapura showed that if all the processors do their random sampling independently, $\Omega(\sqrt{n})$ bits of workspace at each processor is necessary even for an estimate of the distributed basic counting with a relative error bound [14]. The way in [14] to overcome this space lower bound is to coordinate the random sampling procedures by using a common hash function to simulate the sampling procedure. The hash function used in [14] is pair-wise independent and is defined over the field $GF(2^2)$, where $\lambda = \log n$. Their hash function maps the stream location indexes $\{1, 2, … , n\}$ to the sample levels $\{0, 1, … , \lambda\}$. The hash function provides that the probability that a stream location index is hashed to a particular sample level $i$ is exactly equal to $1/2^i$. An incoming 1-bit will be selected into the sample if and only if its stream location index is hashed to a sample level $i \geq \ell$, where $\ell$ is the current sample level. Thus, in average, only $1/2^{\ell+1} + 1/2^{\ell+2} + \ldots + 1/2^n \approx 1/2^\ell$ of the $n$ stream element locations will be selected, which is the goal of the adaptive sampling — each stream location should be selected with probability $P_\ell = 1/2^\ell$. Clearly, coordinated adaptive random sampling yields the same set of selected stream locations at all processors if they are on the same sample level.

Of course, those selected locations that do not have 1-bits will not be saved in the sample.

When the sample is full. The sample level will be incremented by one, when the sample becomes full. All the 1-bits that are currently in sample and whose hash values are less than the new sample level $\ell$ will be discarded from the sample. By doing so, all the 1-bits that Alice has received so far have been assigned the same sample probability $P_\ell$ for selection. Alice will then continue her processing of the new incoming 1-bits using the current sample probability $P_\ell$ until the sample becomes full again.

When the referee receives a query for $U$. All processors will send their samples and sample levels to the referee. For each sample, if its sample level is smaller than $\ell_{max}$, the largest sample level the referee has received, the referee will increase its sample level to $\ell_{max}$ by retaining only those 1-bits whose hash values are larger than or equal to $\ell_{max}$. After this resampling step, all the samples will share the same sample level. The referee will then union all the samples by conducting bit-wise OR of the selected 1-bits according to their stream locations. The number of 1-bits in union multiplied by $1/P_{\ell_{max}}$ will be returned by the referee as the answer to the query for $U$.

3.2 Use a different hash function for sampling

Our method follows GT’s structure but uses a different hash function to conduct the coordinated adaptive sampling procedure. Since the choice of the hash function is critical for the improvement of the processing speed, we provide its details in the following. We first pick a prime number $p$ uniformly at random from $[10n, 20n]$, then pick another two numbers $a$ and $b$ from $\{0, 1, \ldots , p - 1\}$ and $a \neq 0$. The hash function $h : \{1, 2, \ldots , n\} \rightarrow \{0, 1, \ldots , p - 1\}$ is defined as:

$h(x) = (a \cdot x + b) \mod p$. It is well known [6] that:

- $h(x)$ is uniformly distributed in $\{0, 1, \ldots , p - 1\}$.
- $h$ is pair-wise independent, i.e., for any $x_1 \neq x_2$ and $y_1, y_2$: $Pr[(h(x_1) = y_1) \wedge (h(x_2) = y_2)] = Pr[h(x_1) = y_1] \cdot Pr[h(x_2) = y_2]$.

For each $\ell \in \{0, 1, \ldots , \lfloor \log p \rfloor\}$, we define:

$R_\ell = \{0, 1, \ldots , \lfloor p/2^\ell \rfloor - 1\}$.

Our sampling policy is that a stream element $a_i$ will be selected into the sample if $a_i = 1$ and $h(i) \in R_\ell$. Since the hash values $\{0, 1, \ldots , p - 1\}$ are uniformly distributed, the
Algorithm 1: GT’s coordinated adaptive sampling for distributed basic counting using the pairwise independent hash function \( h(x) = (ax + b) \mod p \).

**Input:** Two geographically distributed bit streams 
\( A = \{a_1, a_2, \ldots, a_n\} \) and \( B = \{b_1, b_2, \ldots, b_n\} \), processed in parallel by Alice and Bob, respectively.

**Output:** \( U(A, B) = \sum_{i=1}^{n} (a_i \lor b_i) \), returned by the referee.

**1** Randomly pick a pairwise independent hash function: \( h(x) = (ax + b) \mod p \), where \( p \) is a prime number randomly picked from \([10n, 20n]\), and \( a, b \) are randomly picked from \([0,1, \ldots, p-1]\) and \( a \neq 0 \).

**2 Alice:**
3 Initialize an empty sample \( S \) of size \( \alpha \); \( \ell \leftarrow 0 \).
4 for \( i = 1, 2, \ldots, n \) do
5 \[ \text{if } a_i = 1 \text{ and } h(i) \in R_i \text{ then} \]
6 \[ S \leftarrow S \cup \{i\}; \]
7 \[ \text{while } |S| > \alpha \text{ do} /\sim \text{ Sample is full.} \]
8 \[ \ell = \ell + 1; \]
9 \[ \text{if } \ell > \lfloor \log p \rfloor \text{ then Exit} /\sim \text{ Algorithm fails.} \]
10 \[ \text{Discard every } x \in S \text{ such that } h(x) \notin R_i; \]
11 **Bob:** (Symmetric to Alice)
12 **Referee:** /\sim Upon receiving a query for \( U \).
13 Receive \( S_A, \ell_A \) from Alice and \( S_B, \ell_B \) from Bob;
14 \( \ell' \leftarrow \max(\ell_A, \ell_B) \);
15 if \( \ell_A < \ell' \) then Discard every \( x \in S_A \) such that \( h(x) \notin R_{\ell'} \);
16 else if \( \ell_B < \ell' \) then Discard every \( x \in S_B \) such that \( h(x) \notin R_{\ell'}; \)
17 return \( |S_A \cup S_B|/\ell' \).

The probability of selecting any 1-bit at sample level \( \ell \) is:
\[ P_{\ell} = |R_{\ell}|/|p|^2/\ell \approx 1/2^\ell \]

Algo. 1 shows the pseudocode of GT’s coordinated adaptive sampling, plugged in our new hash function, for estimating \( U \) of two streams.

3.3 The opportunity for speedup

GT’s method checks every stream element at least once (Line 4–5, Algo. 1), which yields their time cost for processing a stream of \( n \) bits is at least \( \Omega(n) \). We observe that the hash values \( h(1), h(2), \ldots \) of the hash function \( h \) we use (Section 3.2) follows some pattern, which will be made clear later. By taking advantage of this pattern, it is possible to skip over some stream elements without checking their hash values, because their hash values can be proved to be out of \( R_i \). In particular, suppose \( a_i \) is the element that we are currently processing and \( a_{i+d} \) is the next element whose hash value is within \( R_i \), we are able to design an algorithm that finds the value of \( d \) using \( O(\log d) \) time. That says we will not need to literally check the elements \( a_{i+1}, a_{i+2}, \ldots, a_{i+d-1} \) one by one. Our algorithm for finding the value of \( d \) is based on a recursive decomposition of the task, so that the new instance of the problem has a significantly smaller problem size. The recursive decomposition is based on some properties of the hash function \( h(x) = (ax + b) \mod p \) that we use for coordinated adaptive sampling.

It is worth noting that Vitter’s technique for fast sampling [25] cannot be used in our setting for speedup, because that technique is to boost independent sampling, which provably needs \( \Omega(\sqrt{n}) \) bits [14] for distributed basic counting in the streaming model. Also, traditional filter-based load shedding techniques [3, 24] will not have provable accuracy guarantee. So, we need a new technique for boosting the coordinated adaptive sampling.

4. NEW TECHNIQUE: DIRECT SAMPLING

The challenge for fast sampling is to find the next stream location that will be selected into the sample if that location has a 1-bit, without having to check the hash value of every stream location up to that sampled location. That is, suppose we are now at sample level \( \ell \) and at the stream location \( x \), we want to compute the following function quickly:

\[ \text{DirectSample}(x, \ell, p, a, b) = x + N^\ell_x \]

where
\[ N^\ell_x = \min \{ i \geq 0 \mid h(x + i) = (a(x + i) + b) \mod p \in R_i \} \]

Note that for an arbitrary setting of the parameters of \( h \), \( N^\ell_x \) may not be well defined because the set \( \{ i \geq 0 \mid h(x + i) \in R_i \} \) can be empty. For example, if \( p = 8, a = 4, b = 2, \ell = 2, x = 1, \) then for any \( i \geq 0 \), the value of \( h(x + i) \) is always either 2 or 6, neither of which belongs to \( R_i = \{0, 1\} \). However, it can be shown that when \( p \) is a prime number, \( N^\ell_x \) is always well defined. We first prove that \( \{ h(x), h(x + 1), \ldots, h(x + p - 1) \} \) and \( \{0, 1, \ldots, p - 1\} \) are actually the same set of \( p \) distinct numbers.

**Lemma 4.1.** For any \( p > 0 \), \( 0 < a < p \), \( 0 < b < p \), and \( x \geq 1 \), if \( p \) is a prime number, then:
\[ \{ h(x), h(x + 1), \ldots, h(x + p - 1) \} = \{0, 1, \ldots, p - 1\} \]

**Proof.** We prove the lemma by contradiction. Let \( A \) denote \( \{ h(x), h(x + 1), \ldots, h(x + p - 1) \} \) and \( B \) denote \( \{0, 1, \ldots, p - 1\} \). Suppose \( A \neq B \), then since every \( h(x + k) \) for \( k = 0, 1, \ldots, p - 1 \) is a member of \( B \) and \( |A| = |B| \), there must exist two integers \( i \) and \( j \), such that \( 0 \leq i < j \leq p - 1 \) and \( h(x + i) = h(x + j) \). That is,
\[ a(x + i) + b = a(x + j) + b \mod p \]
\[ \iff ai = aj \mod p \]
\[ \iff a(j - i) = 0 \mod p \] (2)

Combining the fact that \( 0 < a < p \) and \( 0 < j - i < p \), the above Equation 2 indicates that \( p \) is not a prime number, which is a contradiction, thus the lemma is proved.

**Lemma 4.2.** For any \( p > 0 \), \( 0 < a < p \), \( 0 < b < p \), \( x \geq 1 \), and \( L \geq 0 \), if \( p \) is a prime number, then \( \{ i \geq 0 \mid h(x + i) \leq L \} \) is always not empty.

**Proof.** By Lemma 4.1, we know that \( \{ h(x), h(x + 1), \ldots, h(x + p - 1) \} = \{0, 1, \ldots, p - 1\} \). Because \( L \geq 0 \), there must exist at least one member in the set \( \{ h(x), h(x + 1), \ldots, h(x + p - 1) \} \) whose value is not larger than \( L \), so the lemma is proved.

**Lemma 4.3.** \( N^\ell_x \) always exists.

**Proof.** Recall that by design the sample level \( \ell \) is no more than \( \lfloor \log p \rfloor \) (Line 9, Algo. 1), so \( [2^{-\ell}p] - 1 \geq 0 \) is always true. By setting \( L = [2^{-\ell}p] - 1 \geq 0 \), Lemma 4.2 has proved that the set \( \{ i \geq 0 \mid h(x + i) \leq L \} \) is always not empty, so \( N^\ell_x = \min \{ i \geq 0 \mid h(x + i) \in R_i \} = \min \{ i \geq 0 \mid h(x + i) \leq [2^{-\ell}p] - 1 \} \) is always well defined and exists.

We can now focus on the design of an efficient algorithm for finding \( N^\ell_x \).
**Problem 1.** Given integers \( p > 0, 0 \leq a < p, 0 \leq u < p, \) and \( L \geq 0 \), compute the following d:

\[
d = \begin{cases} 
\min\{i \geq 0 \mid (u + i \cdot a) \mod p \leq L\}, & \text{if } \{i \geq 0 \mid (u + i \cdot a) \mod p \leq L\} \neq \emptyset \\
-1, & \text{otherwise}
\end{cases}
\]

Let \( Z_p \) denote the ring of the nonnegative numbers modulo \( p \). Observe that the sequence of the values of \( h(x + i) \), for \( i = 0, 1, 2, \ldots \), is an arithmetic progression over \( Z_p \) with common difference \( a \). The task of finding \( N'_s \) is reduced to finding \( d (= N'_s) \) in Problem 1 by setting:

\[
u = h(x) = (ax + b) \mod p, \quad L = [2^{-i}p] - 1.
\]

Let \( S = \{u \mod p, (u + a) \mod p, (u + 2a) \mod p, \ldots\} \). For \( i = 0, 1, \ldots \), let \( S[i] = (u + i \cdot a) \mod p, \) the \( i \)th element in the sequence \( S \). Problem 1 can be restated as follows: find the smallest \( i \geq 0 \), such that \( S[i] \leq L \), or report that such \( i \) does not exist.

### 4.1 Possible solutions.

A naive method, as used in [14], is to iteratively check the value of \( (u + i \cdot a) \mod p \) for \( i = 0, 1, \ldots \). It takes \( O(d) \) time if a nonnegative \( d \) is eventually found and the procedure will even never stop if a nonnegative \( d \) does not exist at all.

A better solution is to use the \texttt{Hits} function from [21]. Given a fixed-sized prefix of the sequence \( S \), \texttt{Hits} can efficiently calculate the number of elements in the prefix whose values are less than or equal to \( L \) using \( O(\log y) \) time, where \( y \) is the length of the prefix. Assuming a nonnegative \( d \) exists for Problem 1, if we can make a good guess of the length of the prefix, such that \( y \geq d \), meaning the prefix includes at least one element whose value is less than or equal to \( L \). Then we can use a binary search over the prefix to locate the first element whose value is less than or equal to \( L \). Altogether, it will take \( O(\log^2 y) \) time, because there are \( O(\log y) \) binary search steps and each binary search step takes \( O(\log y) \) time for \texttt{Hits}. However, this method has several unsolved issues: (1) It is not clear how to make a good guess of the length of the prefix so that \( y \geq d \), since \( d \) is unknown: If \( y \geq d \), it wastes computational time; If \( y < d \), we will have to guess a longer prefix of \( S \), leading to an inefficient procedure. (2) Even if a good guess of \( y \) is made, it will take \( O(\log^2 y) \) time to locate the first element whose value is less than or equal to \( L \). We look for a solution that takes only \( O(\log y) \) time. (3) If \( d \geq 0 \) does not exist, the binary search method of using \texttt{Hits} fails.

Another possible solution is to use the \texttt{MinHit} function from [10]. Given a prefix of size \( y \) of the sequence \( S \), by using \( O(\log y) \) time, \texttt{MinHit} can find the first element in the prefix that is less than or equal to \( L \) or return \(-1 \) if such element does not exist. Similar unsolved issues exist: (1) It is not clear how to make a good guess of the size of a prefix so that it includes at least one element that is not larger than \( L \). Bad guesses lead to wasting of computational time as we have explained in the possible solution using \texttt{Hits}. (2) If \( d \geq 0 \) does not exist, the method of using \texttt{MinHit} fails.

### 4.2 Our solution

We now present our algorithm called \texttt{NextHit} for solving Problem 1. Our approach is to directly modify the internal mechanism of the \texttt{MinHit} function so it can work with an infinitely long sequence.

Note that the sequence \( S \) is an arithmetic progression over \( Z_p \) with common distance \( a \). For \( i = 0, 1, \ldots \), let \( S_i \) denote the subsequence which is the \( i \)th round of the progression and \( f_i \) denote the first element in \( S_i \). Let \( F = (f_0, f_1, f_2, \ldots) \) and \( |S_i| \) denote the number of elements in \( S_i \). Figure 2 shows an example \( S \) and its \( S_i \) subsequences and \( F \) sequence.

We start the design of \texttt{NextHit} with the following critical observation from [21].

**Observation 4.1** (Observation 2 of [21]). Sequence \( F = F \{f_0\} = (f_1, f_2, \ldots) \) is an arithmetic progression over \( Z_a \), with common difference \( a - r \) (or \( -r \), equivalently), where \( r \equiv p \mod a \). That is, for every \( i \geq 1 \):

\[
f_i = (f_1 + (i - 1) \cdot (a - r)) \mod a
\]

Figure 2 shows an example \( F \) sequence.

The next lemma says that if \( \{i \geq 0 \mid (u + i \cdot a) \mod p \leq L\} \neq \emptyset \), \( S[d] \) must be the first element whose value is not larger than \( L \) in the sequence \( F \).

**Lemma 4.4** (A generalization of Lemma 3.6 [10]). If \( d \neq -1 \), \( S[d] = f_m \in F \), where \( m = \min\{i \geq 0 \mid f_i \leq L\} \).

By observing \( S \) is an arithmetic progression over \( Z_p \), it is easy to get the following lemma.

**Lemma 4.5** (Lemma 3.7 of [10]). If \( m = \min\{i \geq 0 \mid f_i \leq L\} \) exists, then \( d = (mp - f_0 + f_m)/a \).

### 4.2.1 The overall strategy for solving Problem 1

We will first find the value of \( m \), which will give us the value of \( f_m \) due to Equation 3. Then, we will get the value of \( d \) (thus solving Problem 1) by using Lemma 4.5. That is, the task of solving Problem 1 can be reduced to the task of finding \( m \), which in fact, as we will explain soon, is another instance of Problem 1 with a different parameter setting.

The next lemma will be used to convert an instance of Problem 1 to another instance of Problem 1 which has a different set of parameters but returns the same answer.

**Lemma 4.6** (A generalization of Lemma 3.8 [10]). Let \( d \) denote the answer to an instance of Problem 1 with parameter setting: \( p, a, u, L \). Let \( d' \) denote the answer to another instance of Problem 1 with parameter setting: \( p', a', u' \equiv (p - u + L) \mod p, L' = L \). Then: \( d' = d \).

### 4.2.2 The mechanism of \texttt{NextHit}

Now we are ready to design the algorithmic mechanism of the \texttt{NextHit} algorithm using the discoveries we have presented. Given an instance of Problem 1 with parameter setting \( p, a, u, L \), the easy case is \( u \leq L \), for which we will obviously return \( d = 0 \). Otherwise, because \( S[d] = f_m \) (Lemma 4.4), we will reduce the task of solving Problem 1 with parameter setting \( p, a, u, L \) to the task of finding \( m \) such that \( f_m \) is the first element whose value is not larger than \( L \) in the \( F \) sequence. Observe that the \( F \) sequence is also an arithmetic progression over a smaller ring \( Z_a \) with common distance \( a - r \) (Observation 4.1), so the task of finding \( m \) is actually a new instance of Problem 1 with the following parameter setting:

\[
p_{\text{new}} = a, \quad a_{\text{new}} = a - r, \quad u_{\text{new}} = f_1, \quad L_{\text{new}} = L
\]
Figure 2: An example sequence $S = \langle u + 0 \cdot a \mod p, (u + 1 \cdot a) \mod p, (u + 2 \cdot a) \mod p, \ldots \rangle$, where $u = 7, a = 4, p = 13$. For $i = 0, 1, \ldots$ : (1) $S[i]$ denotes the $i$th element in $S$; (2) The subsequence $S[i]$ denotes the $i$th round of the progression over $\mathbb{Z}_p$; (3) $f_i$ is the smallest element in $S_i$. Suppose $L = 1$ in the setting of Problem 1, the answer should be $d = 5$, because $S[5]$ is the first element whose value is not larger than $L$.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | \cdots |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|-----|
| $S[i]$ | 7 | 11 | 2 | 6 | 10 | 1 | 5 | 9 | 0 | 4 | 8 | 12 | 3 | 7 | 11 | \cdots |
| $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $\ldots$ |
| $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | \ldots |

Note that the returned value of the above instance actually will be equal to $m - 1$, because the $F$ sequence is one indexing based. After $m$ is recursively calculated, we will use Observation 4.1 to directly obtain $f_m$:

$$f_m = (f_1 + (m - 1)(a - r)) \mod a$$

Then we can use Lemma 4.5 to directly calculate $d$, which is the answer to the original instance of Problem 1:

$$d = (mp - f_0 + f_m)/a$$

However, the recursion (Equations 4) may not always be effective because the new progression’s common distance $a - r$ may not be much smaller than the old progression’s common distance $a$ (We will explain later why the size of the common distance is relevant to both the time and space complexities of the NextHit algorithm). Fortunately, by using Lemma 4.6, we can overcome this difficulty by converting the recursive instance to another instance which returns the same answer but works on a progression with a smaller common distance. We summarize the two possible reductions in the following.

**Case 1:** $a - r \leq a/2$. We want to work with $a - r$. Problem 1 is recursively reduced to a new instance of Problem 1 of a smaller size that finds $m$ over sequence $F$ by setting:

$$p_{new} = a, \quad a_{new} = a - r, \quad u_{new} = f_1, \quad L_{new} = L$$

**Case 2:** $r < a/2$. We want to work with $r$. We first recursively reduce Problem 1 to the same setting as in Case 1, which will be further converted to the following parameter setting because of Lemma 4.6:

$$p_{new} = a, \quad a_{new} = a - r, \quad u_{new} = f_1, \quad L_{new} = L$$

After adding some trivial recursion exit conditions, we present the pseudocode of NextHit in Algo. 3, which directly reflects the algorithmic idea that we have presented. Once NextHit is clear, the calculation of DirectSample becomes trivial and is presented in Algo. 2.

**Theorem 4.1.** NextHit($p, a, u, L$) solves Problem 1 using $O(\log a)$ time and $O(\log p \cdot \log a)$ bits of space. When $p$ is a prime number, NextHit($p, a, u, L$) solves Problem 1 using $O(\min(\log a, \log d))$ time and $O(\log p \cdot \min(\log a, \log d))$ bits of space, where $d$ is the value returned by NextHit.

**Proof.** Correctness. Recall that NextHit($p, a, u, L$) should return $d = \min(i \geq 0 \mid (u + i \cdot a) \mod p \leq L)$, if such $d$ exists; otherwise, it will return $d = -1$. The three exit conditions (Line 1–3) capture all the possible cases where the algorithm can return and exit directly. At the end of Line 3, we know $S[d]$ does not occur in $S_0$ and $p > a / 2$, so we are ready to reduce the task of finding $d$ to the task of finding $m$ over the sequence $F$. Line 4–7 correctly calculate $f_1$ and $r$ as the preparation work for the subsequent recursive calls at Line 8–9. Line 8 calculates the 0-based index of the first element in the sequence $F$ that is not larger than $L$. By Lemma 4.6, we know the recursive call at Line 9 will return the same result as that from Line 8. In the case where $-1$ is returned by the recursive call (Line 8 or 9), it means no element in the sequence $F$ is less than or equal to $L$. In that case, we will return $d = -1$ (Line 10). Otherwise, Line 12–13 calculates and returns the value of $d$ using the results returned by the preceding recursive call (Lemma 4.5).

**Time Complexity.** We assume that the additions, multiplications, and divisions take unit time. At each recursive
call (Line 8 or 9), we have \(a_{\text{new}} \leq a/2\), plus the fact that in the worst case the recursion will return when \(a = 1\) (Line 2), so the depth of the recursions is no more than \(\log a\). Because the time cost for the local computation in each recursive call is constant (Line 1–7 and 10–13), the time complexity of \(\text{NextHit}\) is \(O(\log a)\).

In the particular case where \(p\) is a prime number, which is the case in the use of \(\text{NextHit}\) for distributed basic counting in this paper, we know the value of \(d\) returned by \(\text{NextHit}\) is always non-negative (Lemma 4.2). In this case, the length of the arithmetic progression that the caller of \(\text{NextHit}\) works with is \(d + 1\). Because at each recursive call, we have the common distance in the new progression reduced by at least half, so the length of the progression that the next recursive \(\text{NextHit}\) will work with is no more than a half of the caller’s progression. This observation implies that the depth of the recursion is no more than \(\log d\), so the overall time cost is bounded by \(O(\log d)\). Comparing with the time cost of \(\text{NextHit}\) in the general case, we see the time cost of \(\text{NextHit}\) in this particular case is \(O(\min(\log a, \log d))\).

**Space Complexity.** In each recursive call, \(\text{NextHit}\) needs to store a constant number of local variables such as \(a, p, u, L, \ldots\) etc. Since \(p\) dominates \(a, u\) and \(L\) (if \(L \geq p\), then \(\text{NextHit}(\ldots)\) returns without recursive calls (Line 1)), each recursive call needs \(O(\log p)\) stack space. Since the depth of the recursion is no more than \(O(\log a)\), which we have explained in the time complexity analysis, the space cost of the \(\text{NextHit}\) algorithm is upper bounded by \(O(\log^2 p)\) bits.

In the case where \(p\) is a prime number, the depth of the recursion is bounded by \(O(\log d)\), which we have explained the time complexity analysis, so the total space cost is no more than \(O(\log p \cdot \log d)\). Comparing with the space cost of \(\text{NextHit}\) in the general case, we get the time cost of \(\text{NextHit}\) in this particular case is \(O(\log p \cdot \min(\log a, \log d))\).

**Corollary 4.1.** \(\text{DirectSample}(x, \ell, p, a, b)\) finds the next \(\ell\)-level sample location on or after the \(x\)th stream element using \(O(\min(\log a, \log d))\) time and \(O(\log n \cdot \min(\log a, \log d))\) bits of space, where \(n\) is an upper bound of the stream size and \(x + d\) is the value returned by \(\text{DirectSample}\).

5. **BOOSTING THE DISTRIBUTED BASIC COUNTING VIA DIRECT SAMPLING**

Now we present the complete picture of our new method for basic counting on the union of multiple streams, followed by its correctness proof and the analysis of its time, space, and communication complexities.

5.1 Algorithm description

The formal description of the algorithm is presented in Algo. 4. Note that one instance of the algorithm does not automatically yield an \((\epsilon, \delta)\)-estimate of \(U\), but produces an estimate within a factor of \(e\) of \(U\) with a constant probability. The standard technique to reduce the failure probability from a constant to the user-input parameter \(\delta\) is to run in parallel \(O(\log(1/\delta))\) independent instances of the algorithm and return the median of the results from those instances.

The overall structure of the algorithm is still coordinated adaptive sampling, but uses the hash function \(h\) and thus can use the direct sampling technique as a subroutine for a faster sampling procedure. We first randomly pick a pairwise independent hash function \(h\) (Line 1) as defined in Section 3.2. The hash function \(h\) is shared by both the stream processors (Alice and Bob) and the referee for coordinated sampling. Each processor starts with an empty sample of size \(a = \lceil 60/\epsilon^2 \rceil\) and the sample level \(\ell = 0\) (Line 3).

**The use of direct sampling.** The algorithm does not check the hash value of every stream element. After processing a particular stream element \(a_i\) (Line 6–11), the algorithm calls \(\text{DirectSample}(i + 1, \ell, p, a, b)\) (Line 12) to find the next stream element that needs to be checked at the current sample level \(\ell\). The algorithm will go to sleep until that element arrives. That is, all the stream elements that do not have the possibility to be selected regardless of its value will be ignored (Line 5). When the element whose location index was returned by \(\text{DirectSample}\) arrives, it will be selected into the sample if it is a 1-bit (Line 6). Note that we do not need to check the element’s hash value as it has been guaranteed to be in \(R_{\ell}\) by \(\text{DirectSample}\). After the element is processed (Line 6–11), the function call at Line 12 gives the next stream location, at which the stream element will need to be checked. By using the direct sampling technique, our method intuitively is much faster than GT’s method, which we will show later through both theoretical analysis and empirical study.

In the case where the sample becomes overflow after the insertion of a new element (Line 8), the sample level will be increased by one (Line 9). All the stream elements that are currently in the sample but whose hash values do not belong to \(R_{\ell}\) will be discard from the sample (Line 11). The sample level will keep increasing until the sample is not overflowed (Line 8). There is a possibility that the sample level can exceed \(\log p\). If that happens, the algorithm fails, but we will later show the probability of this event is very low. The procedure at the referee to answer a query for \(U\) (Line 15–19) is the same as the one in Algo. 1.

5.2 Correctness

Note that our method still uses the coordinated adaptive sampling on the high level, but uses \(\text{DirectSample}\) for speedup, so the correctness proof of our method follows a parallel structure of the proof by Gibbons and Tirthapura [14]. We present the entire proof in the full version of this paper, because the hash function used in our method is different and also for completeness.

**Theorem 5.1** (Correctness of Algo. 4). The median of the answers returned by \(\beta = \lceil 24/\ln(1/\delta) \rceil\) independent instances (using different hash functions picked randomly and independently) of Algo. 4 is an \((\epsilon, \delta)\)-estimate of \(U\).

5.3 Time complexity

**Theorem 5.2** (Worst-case total time cost). Algo. 4 spends \(\Theta(n \log(1/\delta))\) time in the worst case for processing a bit stream of size \(n\) for maintaining an \((\epsilon, \delta)\)-estimate of \(U\).

**Comment:** (1) \(\Theta(n \log(1/\delta))\) is GT’s total time cost in any case, but it rarely occurs with our method. (2) It is necessary to spend \(\Theta(n \log(1/\delta))\) in the worst case, because in the case all stream elements are 0-bits, we need to check every stream element if we want to estimate \(U\) with a relative error guarantee.

**Theorem 5.3** (Worst-case per-element time cost). Algo. 4 spends \(\Theta(\log(1/\delta) \log n)\) time in the worst case for processing one stream element in order to maintain an \((\epsilon, \delta)\)-estimate of \(U\).
Algorithm 4: Distributed basic counting using direct sampling.

Input: Two geographically distributed bit streams $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$, processed in parallel by Alice and Bob, respectively.
Output: $U(A, B) = \sum_{i=1}^{n} (a_i \land b_i)$, returned by the referee.

1. Randomly pick a pairwise independent hash function: $h(x) = (ax + b) \mod p$, where $p$ is a prime number randomly picked from $[10^n, 20^n]$, and $a, b$ are randomly picked from $\{0, 1, \ldots, p - 1\}$ and $a \neq 0$.

2. Alice:
   1. Initialize an empty sample $S$ of size $\alpha = [60/e^2]$; $\ell \leftarrow 0$;
   2. $i \leftarrow 1$; // DirectSample(1, 0, p, a, b) $\equiv 1$, the first location to check at sample level 0.
   3. Sleep until $a_i$ arrives
   4. if $a_i = 1$ then
   5. $S \leftarrow S \cup \{i\}$
   6. while $|S| > \alpha$ do // Sample is full.
   7. $\ell = \ell + 1$;
   8. if $\ell > \lceil \log p \rceil$ then Exit; // Algorithm fails.
   9. Discard every $x \in S$ such that $h(x) \notin R_\ell$;
   10. $i \leftarrow$ DirectSample$(i + 1, \ell, p, a, b)$; // The next location that needs to be checked.

12. Bob: (Symmetric to Alice)

14. Receive $S_A, \ell_A$ from Alice and $S_B, \ell_B$ from Bob;
15. $\ell^* \leftarrow \max(\ell_A, \ell_B)$
16. if $\ell_A < \ell^*$ then Discard every $x \in S_A$ such that $h(x) \notin R_{\ell^*}$;
17. else if $\ell_B < \ell^*$ then Discard every $x \in S_B$ such that $h(x) \notin R_{\ell^*}$;
18. return $|S_A \cup S_B|/P_{\ell^*}$;

Comment: In practice, when the stream size is large, only a very small portion of the stream will be checked in our method. This indicates the probability of any particular element being processed by more than a constant number of instances of Algo. 4 is very low. So, our method’s worst-case time cost per element will be $O(\log n)$ with high probability.

Intuitively, the new ingredient $\text{DirectSample}$ can significantly speed up the stream processing in practice. Next, we show that the average total time cost of Algo. 4 for processing a $\gamma$-random bit stream is indeed sublinear of the stream size. Recall that the notion of $\gamma$-random bit stream (Definition 1.2) can fit into the distribution of many real-world streams.

Theorem 5.4. The average time cost of Algo. 4 for processing a $\gamma$-random bit stream of size $n$ is $O\left(\frac{1}{\gamma^2} \log^2 n \log \frac{1}{\delta}\right)$ in the maintenance of an $(\epsilon, \delta)$-estimate for $U$.

Comment: (1) Our method is significantly faster than GT’s in practice, providing the first sublinear-time algorithm in the average case for the distributed basic counting problem over most real-world streams. (2) Our method provides the users with the new tradeoff that a lower accuracy demand ($\delta \leq 0.1$) promises a faster processing speed, whereas GT’s is $O(\log \log(1/\delta))$ in any case.

5.4 Space and communication complexity

Theorem 5.5. (Space and communication cost). To maintain an $(\epsilon, \delta)$-estimate of $U$: (1) the workspace at each processor is $O((1/\epsilon^2 + \log n) \log(1/\delta) \log n)$ bits. (2) the memory usage by the referee is $O((1/\epsilon^2) \log(1/\delta) \log n)$ bits. (3) The communication cost per query between each processor and the referee is $O((1/\epsilon^2) \log(1/\delta) \log n)$ bits.

Comment: Compared with the space usage of GT’s, each processor’s extra space cost in our method is $O(\log^2 n \log(1/\delta))$ bits due to the $\text{DirectSample}$ function calls. This extra space usage is a lower order term compared with GT’s space usage when the value of $\epsilon$ is reasonably small (for ex., $\epsilon \leq 0.1$). Further, this extra space cost occurs only $O(\log n)$ times for each algorithm instance in average during the stream processing, because there are only $O(\log n)$ $\text{DirectSample}$ function calls within one algorithm instance in average over the course of stream processing. In practice the stream size $n$ is often very large, which is usually the case in stream processing (otherwise there is no need to design space-efficient streaming algorithms), the stream locations that are returned by $\text{DirectSample}$ within one algorithm instance will be very sparse, meaning the probability that those $\beta$ parallel instances will call the $\text{DirectSample}$ function at the same time is very low. So, in practice, the extra space used for stack maintenance in the $\beta$ instances will be only $O(\log^2 n)$ bits and occurs only $O(\log n)$ times with high probability. We will later show through experiments with both real-world and synthetic data that this extra space usage is too negligible to be detected by the OS.

5.5 Extension: more than two streams

It is trivial to extend our method for maintaining an $(\epsilon, \delta)$-estimate for the distributed basic counting over $k > 2$ streams that are processed by $k$ processors, for some constant $k$. The procedure at each of these $k$ processors will be exactly the same as the procedure in the 2-stream scenario. The difference is when a query arrives, the referee needs to collect the samples and their sample levels from all the processors, and then run the following procedure to generate the estimate for $U$ (Equation 1). The correctness proof and time and space complexity analysis for the 2-stream case can be directly applied to this $k$-stream scenario.

Referee: // Upon receiving a query for $U$.
for $i = 1, 2, \ldots, k$
    Receive $(S_i, \ell_i)$ from Processor $i$
    $\ell^* \leftarrow \max(\ell_i \mid 1 \leq i \leq k)$
for $i = 1, 2, \ldots, k$
    if $\ell_i < \ell^*$ then Discard every $x \in S_i$ such that $h(x) \notin R_{\ell^*}$;
return $|S_1 \cup \ldots \cup S_k|/P_{\ell^*}$;

6. EXPERIMENTS

Goal. Recall that our method speeds up GT’s coordinated adaptive sampling (a.k.a. distinct sampling) without losing its accuracy guarantee. Since the accuracy of adaptive sampling has been well tested by prior work [13, 10], it is sufficient to demonstrate the time and space efficiency of our method compared with GT’s. Further, our method speeds up the processing of every individual stream and thus improve the overall parallel processing speed on multiple distributed streams, it is sufficient to demonstrate the performance of our method and GT’s on one single stream.

Main messages. (1) Our method is exponentially faster than GT’s method, whose processing time is linear of the stream size. (2) Our method’s processing speed increases
Table 2: The time cost (in seconds) of both GT’s and our methods in the processing of multiple bit streams with different accuracy demands $\epsilon$. The processing speed of GT’s does not change with $\epsilon$, while ours becomes faster when $\epsilon$ increases. In all cases, our method is significantly faster than GT’s method.

<table>
<thead>
<tr>
<th>Data</th>
<th>Methods</th>
<th>$\epsilon = 0.01$</th>
<th>$\epsilon = 0.02$</th>
<th>$\epsilon = 0.05$</th>
<th>$\epsilon = 0.1$</th>
<th>$\epsilon = 0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio Bible 460, 805, 446 1-bits</td>
<td>GT’s</td>
<td>18.7823</td>
<td>18.3439</td>
<td>18.2506</td>
<td>18.2531</td>
<td>18.2386</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>3.6273</td>
<td>1.0784</td>
<td>0.2841</td>
<td>0.0528</td>
<td>0.0285</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>$&gt; 5x$</td>
<td>$&gt; 17x$</td>
<td>$&gt; 64x$</td>
<td>$&gt; 345x$</td>
<td>$&gt; 639x$</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>3.2259</td>
<td>1.0117</td>
<td>0.3180</td>
<td>0.1055</td>
<td>0.0278</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>$&gt; 8x$</td>
<td>$&gt; 28x$</td>
<td>$&gt; 89x$</td>
<td>$&gt; 268x$</td>
<td>$&gt; 1020x$</td>
</tr>
<tr>
<td>Earth Image 789, 808, 848 1-bits</td>
<td>GT’s</td>
<td>31.9561</td>
<td>31.6179</td>
<td>31.5310</td>
<td>31.4197</td>
<td>31.5040</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>3.1705</td>
<td>1.3294</td>
<td>0.2932</td>
<td>0.0843</td>
<td>0.0245</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>$&gt; 10x$</td>
<td>$&gt; 23x$</td>
<td>$&gt; 107x$</td>
<td>$&gt; 372x$</td>
<td>$&gt; 1285x$</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>5.4275</td>
<td>1.6266</td>
<td>0.2791</td>
<td>0.0678</td>
<td>0.0372</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>$&gt; 2x$</td>
<td>$&gt; 7x$</td>
<td>$&gt; 44x$</td>
<td>$&gt; 184x$</td>
<td>$&gt; 336x$</td>
</tr>
<tr>
<td>Synthetic-0.3 300, 023, 303 1-bits</td>
<td>GT’s</td>
<td>15.6868</td>
<td>15.2895</td>
<td>15.2082</td>
<td>15.1802</td>
<td>15.1939</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>2.9945</td>
<td>1.4045</td>
<td>0.4531</td>
<td>0.0998</td>
<td>0.0531</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>$&gt; 5x$</td>
<td>$&gt; 10x$</td>
<td>$&gt; 33x$</td>
<td>$&gt; 152x$</td>
<td>$&gt; 286x$</td>
</tr>
<tr>
<td>Synthetic-0.4 400, 003, 206 1-bits</td>
<td>GT’s</td>
<td>18.5577</td>
<td>17.5786</td>
<td>18.0129</td>
<td>17.9844</td>
<td>17.9933</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>2.3465</td>
<td>0.8511</td>
<td>0.2100</td>
<td>0.0782</td>
<td>0.0306</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>$&gt; 7x$</td>
<td>$&gt; 20x$</td>
<td>$&gt; 85x$</td>
<td>$&gt; 229x$</td>
<td>$&gt; 586x$</td>
</tr>
</tbody>
</table>

when the accuracy demand decreases (the value of $\epsilon$ increases), while GT’s method’s processing speed does not change regardless of the value of $\epsilon$. (3) Our method does not introduce any detectable space overhead, regardless of the data sets and the value of $\epsilon$, compared with the space cost by GT’s. All the above observations are perfectly consistent with the theoretical results summarized in Table 1.

**Setup and measurement methods** All of our experiments were conducted on a Dell Precision T5500s machine that has a 2.13GHz Quad-core E5506 CPU with 4M cache. The machine runs 64-bit Ubuntu 10.04 and has 8GB DDR3 ECC 1066MHz SDRAM. We faithfully implemented the coordinated adaptive sampling both with and without using the DirectSample technique (Algo. 1 and 4) using the C++ programming language. All executables were built by GCC 4.4.3. We use the the system time to measure the time cost. We measure the space usage of our programs by VmPeak minus the memory cost for storing the data sets. VmPeak is an entry in the /proc/<pid>/status file, provided by the Linux system. It captures the peak usage in KB of the total amount of virtual memory used by the process, including the memory cost for the code, data, and shared libraries plus the pages that have been swapped out. VmPeak represents the peak of the actual total memory cost of the process.

**Data Source.** We used the following real-world and synthetic data in our experiments. We scan and process each data set and the value of $\epsilon$. (3) Our method does not introduce any detectable space overhead, regardless of the value of $\epsilon$. (4) Our method is significantly faster than GT’s method.

Audio Bible. It has a total of 919,658,056 bits, of which 460,805,446 are 1-bits.

—Video of President G. W. Bush’s speech. It has a total of 1,434,146,160 bits, of which 710,447,850 are 1-bits.

—NASA Earth image. It has a total of 1,575,903,872 bits, of which 789,808,848 are 1-bits.

—Day 37 of Worldcup 98’s network traffic. It has a total of 893,788,160 bits, of which 257,380,419 are 1-bits.

—Synthetic data. Two synthetic bit sequences, where the probabilities of having a 1-bit are 0.3 and 0.4, respectively. Each sequence has a total of 1,000,000,000 bits, of which 300,023,303 and 400,002,206 are 1-bits, respectively.

### 6.1 Time efficiency

Table 2 shows the total time cost of both methods in processing different data sets with different accuracy demands.

Overall boosting. Our method overall is significantly faster than GT’s by a factor of several to several hundred times. This speedup becomes more significant when the value of $\epsilon$ increases. In all cases, our method is significantly faster than GT’s method.

**A new tradeoff.** The time cost of GT’s method is indepen-

---

1https://ia600306.us.archive.org/23/items/Political_videos-GeorgeWBush20030904_5_224/Political_videos-GeorgeWBush20030904_5_224.ogg
2http://eoimages.gsfc.nasa.gov/images/imagercords/73000/73909/world.topo.bathy.200412.3x21600x10800.png
3http://ita.ee.lbl.gov/html/contrib/WorldCup.html
Figure 3: Stream size vs. time, $\epsilon = 0.01$. The processing time of GT's method is linear of the stream size, while ours is sublinear. Our method overall is much faster than GT's, especially when the stream size (the number of 1-bits in the stream, indeed) becomes larger.

dependent from the value of $\epsilon$, because their technique processes every stream element. However, our method becomes much faster when the value of $\epsilon$ increases. This is because a larger $\epsilon$ yields a smaller sample size, which makes the sample level be increased more often. Recall that a higher sample level selects stream elements with smaller probability, so it helps our DirectSample technique be able to skip more stream elements. This new feature in our method is important and useful, because it provides the user with a new trade-off that a lower accuracy demand will not only save memory space but also will speed up the data processing.

Even faster for longer streams. The more 1-bits are present in the stream, the more significant improvement our method makes against GT's method. This is because more 1-bits leads to a higher sample level during the stream processing, which yields a lower sampling probability. That helps DirectSample skip over more stream elements, leading to an overall faster processing speed.

Sublinear-linear time cost. Figure 3 shows that the time cost of GT's method is linear of the stream size, simply because their method processes every stream element, whereas our method's processing time is sublinear of the stream size. The figure again shows our method overall is much faster than GT’s, especially when the stream size (the number of 1-bits in the stream, indeed) becomes larger for the reason that we have given above. The plot in Figure 3 shows the case where $\epsilon = 0.01$. Figures regarding other $\epsilon$ values are given in the full version of this paper, from which the same observations can be made.

6.2 Space efficiency

We show the space usage of both methods in Figure 4, which reveals the following observations.

Space usage only depends on $\epsilon$. The space usages of both methods are independent of stream size, but heavily depend upon the value of $\epsilon$. When the value of $\epsilon$ decreases, the space usage increases. This is consistent with the theoretical results — the space usage of one instance of GT’s and our method are $O((1/\epsilon^2) log n)$ bits and $O((1/\epsilon^2 + log n) log n)$ bits, respectively. Note that we use the 64-bit unsigned long int to represent the stream size, so the impact of the log $n$ term in the big-oh bounds becomes fixed.

No detectable extra space usage. For all data sets and all $\epsilon$ values, there is no detectable extra space usage by our method. That is, the extra space cost of $O(log^2 n)$ bits by our method from the theoretical analysis is too negligible to be detected by the OS in practice.

7. CONCLUSION

In this paper, we designed DirectSample, a new technique for fast sampling, and used it in the coordinated adaptive sampling for distributed basic counting. Both the theoretical analysis and the experimental results show that our method is exponentially faster than the state-of-the-art GT’s method. Further, the extra space usage by our method from the theoretical analysis is not only negligible in theory but also undetectable in practice. Our new method can potentially save a vast majority of processing time and energy needed by GT’s method in the processing of real-world streams, whose size is nearly unbounded. We also believe the new DirectSample technique can be of other independent interest.

8. REFERENCES

Figure 4: The space usage (in KB) of both methods in the processing of multiple bit streams with different $\epsilon$. The space usage of both methods increases when $\epsilon$ decreases and is independent from the stream size. There is no detectable extra space cost by our method compared with GT's.