SIAM J. COMPUT.
Vol. 37, No. 2, pp. 359–379

RANGE-EFFICIENT COUNTING OF DISTINCT ELEMENTS IN A MASSIVE DATA STREAM*

A. PAVAN† AND SRIKANTA TIRTHAPURA‡

Abstract. Efficient one-pass estimation of F_0, the number of distinct elements in a data stream, is a fundamental problem arising in various contexts in databases and networking. We consider *range-efficient estimation of* F_0: estimation of the number of distinct elements in a data stream where each element of the stream is not just a single integer but an interval of integers. We present a randomized algorithm which yields an (ϵ, δ)-approximation of F_0, with the following time and space complexities (n is the size of the universe of the items): (1) The amortized processing time per interval is $O(\log \frac{1}{\delta} \log \frac{n}{\epsilon})$. (2) The workspace used is $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log n)$ bits. Our algorithm improves upon a previous algorithm by Bar-Yossef, Kumar and Sivakumar [*Proceedings of the* 13*th ACM–SIAM Symposium on Discrete Algorithms (SODA)*, 2002, pp. 623–632], which requires $O(\frac{1}{\epsilon^5} \log \frac{1}{\delta} \log^5 n)$ processing time per item. This algorithm can also be used to compute the max-dominance norm of a stream of multiple signals and significantly improves upon the previous best time and space bounds by Cormode and Muthukrishnan [*Proceedings of the* 11*th European Symposium on Algorithms (ESA)*, Lecture Notes in Comput. Sci. 2938, Springer, Berlin, 2003, pp. 148–160]. This algorithm also provides an efficient solution to the *distinct summation problem*, which arises during data aggregation in sensor networks [*Proceedings of the* 2*nd International Conference on Embedded Networked Sensor Systems*, ACM Press, New York, 2004, pp. 250–262, *Proceedings of the* 20*th International Conference on Data Engineering (ICDE)*, 2004, pp. 449–460].

Key words. data streams, range efficiency, distinct elements, reductions, sensor networks

AMS subject classifications. 68W20, 68W25, 68W40, 68P15

DOI. 10.1137/050643672

1. Introduction.

One of the most significant successes of research on data stream processing has been the efficient estimation of the frequency moments of a stream in one-pass using limited space and time per item. An especially important aggregate is the number of distinct elements in a stream, which is often referred to as the zeroth frequency moment and denoted by F_0. The counting of distinct elements arises in numerous applications involving a stream. For example, most database query optimization algorithms need an estimate of F_0 of the data set [HNSS95]. In network traffic monitoring, this can be used to compute the number of distinct web pages requested from a web site or the number of distinct source addresses among all Internet protocol (IP) packets passing through a router. Further, the computation of many other aggregates of a data stream can be reduced to the computation of F_0.

In most data stream algorithms in the literature the following model is studied: Each element in the stream is a single item (which can be represented by an integer), and the algorithm needs to process this item "efficiently," both with respect to time and space; i.e., the time taken to process each element should be small, and the total

*Received by the editors October 27, 2005; accepted for publication (in revised form) November 13, 2006; published electronically May 16, 2007. A preliminary version of this article appeared in *Proceedings of the* 21*st International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 2005.

http://www.siam.org/journals/sicomp/37-2/64367.html

†Department of Computer Science, Iowa State University, Ames, IA 50010 (pavan@cs.iastate.edu). This author's research was supported in part by NSF grant CCF-0430807.

‡Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010 (snt@iastate.edu). This author's research was supported in part by NSF grant CNS-0520102.

workspace used should be small. Many algorithms have been designed in this model to estimate frequency moments [AMS99, GT01, BYJK$^+$02, CK04] and other aggregates [FKSV02, DGIM02, CDIM02, AJKS02, Mut05, BBD$^+$02] of massive data sets.

However, in many cases it is advantageous to design algorithms which work on a more general data stream, where each element of the stream is not a single item but a *list of items*. In a stream of integers, this list often takes the form of an interval of integers. To motivate this requirement for processing intervals of integers, we give some examples below.

Bar-Yossef, Kumar, and Sivakumar [BYKS02] formalize the concept of *reductions* between data stream problems and demonstrate that, in many cases, reductions naturally lead to the need for processing a list of items quickly, much faster than processing them one by one. They consider the problem of estimating the number of triangles in a graph G, where the edges of G arrive as a stream in an arbitrary order. They present an algorithm that uses a reduction from the problem of computing the number of triangles in a graph to the problem of computing the zeroth and second frequency moments (denoted F_0 and F_2, respectively) of a stream of integers. However, for each edge e in the stream, the reduction produces a list of integers, and the size of each such list could be as large as n, the number of vertices in the graph. If one used an algorithm for F_0 or F_2 which processed these integers one by one, the processing time per edge would be $\Omega(n)$, which is prohibitive. Thus, this reduction needs an algorithm for computing F_0 and F_2 which can handle a list of integers efficiently, and such an algorithm is called *list-efficient*. In many cases, including the above, these lists are simply intervals of integers, and an algorithm which can handle such intervals efficiently is called *range-efficient*.

Another application of such list-efficient and range-efficient algorithms is in aggregate computation over sensor networks [NGSA04, CLKB04]. The goal here is to compute the sum (or average) of a stream of sensor observations. However, due to multipath routing of data, the same observation could be repeated multiple times at a node, and the sum should be computed over only *distinct elements* of the stream. This leads to the *distinct summation* problem defined below, which was studied by Considine et al. [CLKB04] and Nath et al. [NGSA04]. Given a multiset of items $M = \{x_1, x_2, \dots\}$, where $x_i = (k_i, c_i)$, compute $\sum_{distinct(k_i,c_i)\in M} c_i$. The distinct summation problem can be reduced to F_0 computation as follows: For each $x_i = (k_i, c_i)$, generate a list l_i of c_i distinct but consecutive integers such that for distinct x_i's the lists l_i do not overlap, but if element x_i reappeared, the same list l_i would be generated. An F_0 algorithm on this stream of l_i's will give the distinct summation required, but the algorithm should be able to efficiently process a list of elements. Each list l_i is a range of integers, so this needs a range-efficient algorithm for F_0.

Yet another application of range-efficient algorithms for F_0 is in computing the *max-dominance norm* of multiple data streams. The concept of the max-dominance norm is useful in financial applications [CM03] and IP network monitoring [CM03]. The max-dominance norm problem is as follows: Given k streams of m integers each, let
$a_{i,j}, i = 1, 2, \dots, k, j = 1, 2, \dots, m$, represent the jth element of the ith stream. The max-dominance norm is defined as $\sum_{j=1}^{m} \max_{1\le i\le k} a_{i,j}$. Assume for all $a_{i,j}$, $1 \le a_{i,j} \le n$. In section 5 we show that the computation of the max-dominance norm can be reduced to range-efficient F_0 and derive efficient algorithms using this reduction.

The above examples illustrate that range-efficient computation of F_0 is a fundamental problem, useful in diverse scenarios involving data streams. In this paper, we

present a novel algorithm for range-efficient computation of F_0 of a data stream that provides the current best time and space bounds. It is well known [AMS99] that exact computation of the F_0 of a data stream requires space linear in the size of the input in the worst case. In fact, even deterministically approximating F_0 using sublinear space is impossible. For processing massive data streams, it is clearly infeasible to use space linear in the input size. Thus, we focus on designing randomized approximation schemes for range-efficient computation of F_0.

DEFINITION 1. *For parameters* $0 < \epsilon < 1$ *and* $0 < \delta < 1$, *an* (ϵ, δ)*-estimator for a number* Y *is a random variable* X *such that* $\Pr[|X - Y| > \epsilon Y] < \delta$.

1.1. Our results. We consider the problem of range-efficient computation of F_0, defined as follows. The input stream is $R = r_1, r_2, \ldots, r_m$, where each stream element $r_i = [x_i, y_i] \subset [1, n]$ is an interval of integers $x_i, x_i + 1, \ldots, y_i$. The length of an interval r_i could be any number between 1 and n. Two parameters, $0 < \epsilon < 1$ and $0 < \delta < 1$, are supplied by the user.

The algorithm is allowed to view each element of the stream only once and has a limited workspace. It is required to process each item quickly. Whenever the user desires, it is required to output the F_0 of the input stream, i.e., the total number of distinct integers contained in all the intervals in the input stream R. For example, if the input stream was $[1, 10], [2, 5], [5, 12], [41, 50]$, then $F_0 = |[1, 12] \cup [41, 50]| = 22$.

We present an algorithm with the following time and space complexities:

- the amortized processing time per interval is $O(\log \frac{1}{\delta} \log \frac{n}{\epsilon})$;
- the time to answer a query for F_0 at anytime is $O(1)$;
- the workspace used is $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log n)$ bits.

Prior to this work, the most efficient algorithm for range-efficient F_0 computation was by Bar-Yossef, Kumar, and Sivakumar [BYKS02] and took $O(\frac{1}{\epsilon^5} \log \frac{1}{\delta} \log^5 n)$ processing time per interval and $O(\frac{1}{\epsilon^3} \log \frac{1}{\delta} \log n)$ space. Our algorithm performs significantly better with respect to time and better with respect to space.

Extensions to the basic algorithm. The basic algorithm for range-efficient F_0 can be extended to the following more general scenarios:

- It can process a list of integers that is in an arithmetic progression as efficiently as it can handle an interval of integers.
- The algorithm can also be used in the *distributed streams* model, where the input stream is split across multiple parties and F_0 has to be computed on the union of the streams observed by all the parties.
- If the input consists of multidimensional ranges, then the algorithm can be made range-efficient in every coordinate [BYKS02].

Our algorithm is based on random sampling. The set of distinct elements in the stream is sampled at an appropriate probability, and this sample is used to determine the number of distinct elements. The random sampling algorithm is based on the algorithm of Gibbons and Tirthapura [GT01] for counting the number of distinct elements in a stream of integers; however, their algorithm [GT01] was not range-efficient.

A key technical ingredient in our algorithm is a novel *range sampling* algorithm, which can quickly determine the size of a sample resulting from an interval of integers. Using this range sampling algorithm, an interval of integers can be processed by the algorithm much faster than processing them one by one, and this ultimately leads to a faster range-efficient F_0 algorithm.

1.2. Applications. As a result of the improved algorithm for range-efficient F_0, we obtain improved space and time bounds for dominance norms and distinct

summation.%newpage

Dominance norms. Using a reduction to range-efficient F_0, which is elaborated on in section 5, we derive an (ϵ, δ)-approximation algorithm for the max-dominance norm with the following performance guarantees:

- the workspace in bits is $O((\log m + \log n)\frac{1}{\epsilon^2}\log\frac{1}{\delta})$;
- the amortized processing time per item is $O(\log\frac{a}{\epsilon}\log\frac{1}{\delta})$, where a is the value of the item being processed;
- the worst-case processing time per item is $O((\log\log n \log a)\frac{1}{\epsilon^2}\log\frac{1}{\delta})$.

In prior work, Cormode and Muthukrishnan [CM03] gave an (ϵ, δ)-approximation algorithm for the max-dominance norm. Their algorithm uses space $O((\log n + l\frac{1}{\epsilon}\log m \log\log m)\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ and spends $O((\log a \log m)\frac{1}{\epsilon^4}\log\frac{1}{\delta})$ time to process each item, where a is the value of the current element. Our algorithm performs better spacewise and significantly better timewise.

Distinct summation and data aggregation in sensor networks. Our algorithm also provides improved space and time bounds for the distinct summation problem, through a reduction to the range-efficient F_0 problem. Given a multiset of items $M = \{x_1, x_2, \dots\}$, where each x_i is a tuple (k_i, c_i), where $k_i \in [1, m]$ is the "type" and $c_i \in [1, n]$ is the "value." The goal is to compute $S = \sum_{distinct(k_i,c_i)\in M} c_i$. In distinct summation, it is assumed that a particular type is always associated with the same value. However, the same (type, value) pair can appear multiple times in the stream. This reflects the setup used in sensor data aggregation, where each sensor observation has an identifier and a value, and the same observation may be transmitted to the sensor data "sink" through multiple paths. The sink should compute aggregates such as the sum and average over only distinct observations.

We provide an (ϵ, δ)-approximation for the distinct summation problem with the following guarantees:

- the amortized processing time per item is $O(\log\frac{1}{\delta}\log\frac{n}{\epsilon})$;
- the time to answer a query is $O(1)$;
- the workspace used is $O((\log m + \log n)\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ bits.

Considine et al. [CLKB04] and Nath et al. [NGSA04] present alternative algorithms for the distinct summation problem. Their algorithms are based on the algorithm of Flajolet and Martin [FM85] and assume the presence of an ideal hash function, which produces completely independent random outputs on different inputs. However, their analysis does not consider the space required to store such a hash function. Moreover, it is known that hash functions that generate completely independent random values on different inputs cannot be stored in limited space.

Our algorithm does not make an assumption about the availability of ideal hash functions. Instead, we use a hash function which can be stored using limited space but generates only pairwise-independent random numbers. We note that Alon, Matias, and Szegedy [AMS99] provide a way to replace the hash functions used in the Flajolet–Martin algorithm [FM85] by pairwise-independent hash functions. However, the F_0 algorithm by Alon, Matias, and Szegedy [AMS99] is not range-efficient. If the hash functions introduced by Alon, Matias, and Szegedy [AMS99] were used in the algorithms in [CLKB04, NGSA04], there still is a need for a range sampling function similar to the one we present in this paper.

Counting triangles in a data stream. As described above, the problem of counting triangles in a graph that arrives as a stream of edges can be reduced to range-efficient computation of F_0 and F_2 on a stream of integers. Our range-efficient

F_0 algorithm provides a faster solution to one of the components of the problem. However, due to the high complexity of the range-efficient algorithm for F_2, this is not sufficient to obtain an overall reduction in the runtime of the algorithm for counting triangles in graphs. There is recent progress on the problem of counting triangles by Buriol et al. [BFL$^+$06] (see also Jowhari and Ghodsi [JG05]), who give an algorithm through direct random sampling, without using the reduction to F_0 and F_2. The algorithm by Buriol et al. is currently the most efficient solution to this problem and has a space complexity proportional to the ratio between the number of length 2 paths in the graph and the number of triangles in the graph and an expected update time $O(\log |V| \cdot (1 + s \cdot |V|/|E|))$, where s is the space requirement and V and E are the vertices and the edges of the graph, respectively.

1.3. Related work. Estimating F_0 of a data stream is a very well studied problem, because of its importance in various database and networking applications. It is well known that computing F_0 exactly requires space linear in the number of distinct values, in the worst case. Flajolet and Martin [FM85] gave a randomized algorithm for estimating F_0 using $O(\log n)$ bits. This assumed the existence of certain ideal hash functions, which we do not know how to store in small space. Alon, Matias, and Szegedy [AMS99] describe a simple algorithm for estimating F_0 to within a constant relative error which worked with pairwise-independent hash functions and also gave many important algorithms for estimating other frequency moments $F_k, k > 1$ of a data set.

Gibbons and Tirthapura [GT01] and Bar-Yossef et al. [BYJK$^+$02] present algorithms for estimating F_0 to within arbitrary relative error. Neither of these algorithms is range-efficient. The algorithm by Gibbons and Tirthapura used random sampling; its space complexity was $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log n)$, and processing time per element was $O(\log \frac{1}{\delta})$. The space complexity of this algorithm was improved by Bar-Yossef et al. [BYJK$^+$02], who gave an algorithm with better space bounds, which are the order of the sum of $O(\log n)$ and $poly(\frac{1}{\epsilon})$ terms rather than their product, but at the cost of increased processing time per element.

Indyk and Woodruff [IW03] present a lower bound of $\Omega(\log n + \frac{1}{\epsilon^2})$ for the space complexity of approximating F_0, thus narrowing the gap between known upper and lower bounds for the space complexity of F_0. Since the F_0 problem is a special case of the range-efficient F_0 problem, these lower bounds apply to range-efficient F_0, too. Thus our space bounds of $O((\log m + \log n)\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ compare favorably with this lower bound, showing that our algorithms are near optimal spacewise. However an important metric for range-efficient F_0 is the processing time per item, for which no useful lower bounds are known.

The work of Feigenbaum et al. [FKSV02] on estimating the L^1-difference between streams also has the idea of reductions between data stream algorithms and uses a range-efficient algorithm in their reduction. Bar-Yossef, Kumar, and Sivakumar [BYKS02] mention that their notion of reductions and list-efficiency were inspired by the above-mentioned work of Feigenbaum et al. The algorithm for the L^1-difference [FKSV02] is not based on random sampling but relies on quickly summing many random variables. They develop limited independence random variables which are *range summable*, while our sampling-based algorithm makes use of a *range sampling* technique. Further work on range summable random variables includes Gilbert et al. [GKMS03], Reingold and Naor (for a description see [GGI$^+$02]), and Calderbank et al. [CGL$^+$05].

There is much other interesting work on data stream algorithms. For an overview,

the reader is referred to the excellent surveys in [Mut05, BBD+02].

Organization of this paper. The rest of the paper is organized as follows: Section 2 gives a high level overview of the algorithm for range-efficient F_0. Section 3 gives the range sampling algorithm, its proof, and analysis of complexity. Section 4 presents the algorithm for computing F_0 using the range sampling algorithm as the subroutine. Section 5 gives extensions of the basic algorithm to distributed streams and the computation of dominance norms.

2. A high level overview. Our algorithm is based on random sampling. A random sample of the distinct elements of the data stream is maintained at an appropriate probability, and this sample is used in finally estimating the number of distinct elements in the stream. A key technical ingredient is a novel *range sampling* algorithm, which quickly computes the number of integers in a range $[x, y]$ which belong to the current random sample. The other technique needed to make random sampling work here is *adaptive sampling* (see Gibbons and Tirthapura [GT01]), where the sampling probability is decreased every time the sample overflows. The range sampling algorithm, when combined with adaptive random sampling, yields the algorithm for range-efficient estimation of F_0.

2.1. A random sampling algorithm for F_0. At a high level, our random sampling algorithm for computing F_0 follows a similar structure to the algorithm by Gibbons and Tirthapura [GT01]. We first recall the main idea of their random sampling algorithm. The algorithm keeps a random sample of all the distinct elements seen so far. It samples each element of the stream with a probability P, while making sure that the sample has no duplicates. Finally, when an estimate is asked for F_0, it returns the size of the sample, multiplied by $1/P$. However, the value of P cannot be decided in advance. If P is large, then the sample might become too big if F_0 is large. On the other hand, if P is too small, then the approximation error might be very high if the value of F_0 was small. Thus, the algorithm starts off with a high value of $P = 1$ and decreases the value of P every time the sample size exceeds a predetermined maximum sample size, α.

The algorithm maintains a current sampling level ℓ which determines a sampling probability P_ℓ. The sampling probability P_ℓ decreases as level ℓ increases. Initially, $\ell = 0$, and ℓ never decreases. The sample at level ℓ, denoted $S(\ell)$, is determined by a hash function $S(\cdot, \ell)$ for $\ell = 0, 1, \ldots, \ell_{max}$, where ℓ_{max} is some maximum level to be specified later.

When item x is presented to the algorithm, if $S(x, \ell) = 1$, then x is stored in the sample $S(\ell)$, and it is not stored otherwise. Each time the size of $S(\ell)$ exceeds α (i.e., an overflow occurs), the current sampling level ℓ is incremented, and an element $x \in S(\ell)$ is placed in $S(\ell + 1)$ only if $S(x, \ell + 1) = 1$. We will always ensure that if $S(x, \ell + 1) = 1$, then $S(x, \ell) = 1$. Thus $S(\ell + 1)$ is a subsample of $S(\ell)$. Finally, anytime an estimate of the number of distinct elements is asked for, the algorithm returns $\frac{|S(\ell)|}{P_\ell}$, where ℓ is the current sampling level. We call the algorithm described so far as the *single-item* algorithm.

2.2. Range efficiency. When each element of the stream is an interval of items rather than a single item, the main technical problem is to quickly figure out how many points in this interval belong in the sample. More precisely, if ℓ is the current sampling level and an interval $r_i = [x_i, y_i]$ arrives, it is necessary to know the size of the set $\{x \in r_i | S(x, \ell) = 1\}$. We call this the *range sampling* problem.

A naive solution to range sampling is to consider each element $x \in r_i$ individually

and check if $S(x,\ell)=1$, but the processing time per item would be $\Theta(y_i - x_i)$, which could be as much as $\Theta(n)$. We show how to reduce the time per interval significantly, to $O(\log(y_i - x_i))$ operations.

The range-efficient algorithm simulates the single-item algorithm as follows: When an interval $r_i = [x_i, y_i]$ arrives, the size of the set $\{x \in r_i | S(x,\ell)=1\}$ is computed. If the size is greater than zero, then r_i is added to the sample; otherwise, r_i is discarded. If the number of intervals in the sample becomes too large, then the sampling probability is decreased by increasing the sampling level from ℓ to $\ell+1$. When such an increase in sampling level occurs, every interval r in the sample such that $\{x \in r | S(x,\ell+1)=1\}$ is empty is discarded.

Finally, when an estimate for F_0 is asked for, suppose the current sample is the set of intervals $S = \{r_1, \ldots, r_k\}$ and ℓ is the current sampling level. The algorithm returns $|T|/P_\ell$, where $T = \{x \in r_i \mid r_i \in S, S(x,\ell)=1\}$.

2.2.1. Hash functions. Since the choice of the hash function is crucial for the range sampling algorithm, we first precisely define the hash functions used for sampling. We use a standard 2-universal family of hash functions [CW79]. First choose a prime number p between $10n$ and $20n$, and then choose two numbers a, b at random from $\{0, \ldots, p-1\}$. Define hash function $h : \{1, \ldots, n\} \to \{0, \ldots, p-1\}$ as $h(x) = (a \cdot x + b) \mod p$.

The two properties of h that are important to the F_0 algorithm are as follows:

1. For any $x \in \{1, \ldots, n\}$, $h(x)$ is uniformly distributed in $\{0, \ldots, p-1\}$.
2. The mapping is pairwise independent.
   For $x_1 \neq x_2$ and $y_1, y_2 \in \{0, \ldots, p-1\}$,
   $\Pr[(h(x_1) = y_1) \wedge (h(x_2) = y_2)] = \Pr[h(x_1) = y_1] \cdot \Pr[h(x_2) = y_2]$.

For each level $\ell = 0, \ldots, \lfloor \log n \rfloor$, we define the following:

- Region $R_\ell \subset \{0, \ldots, p-1\}$:

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

- For every $x \in [1, n]$, the sampling function $S(x,\ell)$ is defined as $S(x,\ell) = 1$ if $h(x) \in R_\ell$ and $S(x,\ell) = 0$ otherwise.
- The sampling probability at level ℓ is denoted by P_ℓ. For all $x_1, x_2 \in [1, n]$, $\Pr[S(x_1,\ell) = 1] = \Pr[S(x_2,\ell) = 1]$, and we denote this probability by P_ℓ. Since $h(x)$ is uniformly distributed in $\{0, \ldots, p-1\}$, we have $P_\ell = |R_\ell|/p$.

2.2.2. Range sampling. During range sampling, the computation of the number $\{x \in r_i | S(x,\ell)=1\}$ reduces to the following problem: Given numbers a, b, and p such that $0 \leq a, b < p$, function $f : [1, n] \to [0, p-1]$ is defined as $f(x) = (a \cdot x + b) \mod p$. *Given intervals* $[x_1, x_2] \subset [1, n]$ *and* $[0, q] \subset [0, p-1]$*, quickly compute the size of the set* $\{x \in [x_1, x_2] | f(x) \in [0, q]\}$.

Note that n could be a large number, since it is the size of the universe of integers in the stream. We present an efficient solution to the above problem. Our solution is a recursive procedure which works by reducing the above problem to another range sampling problem but over a significantly smaller range, whose length is less than half the length of the original range $[x_1, x_2]$. Proceeding thus, we get an algorithm whose time complexity is $O(\log(x_2 - x_1))$. Our range sampling algorithm might be of independent interest and may be useful in the design of other range-efficient algorithms for the data stream model.

The algorithms for computing F_0 found in [AMS99, GT01, BYJK$^+$02] use hash functions defined over $GF(2^m)$, which are different from the ones that we use. Bar-

Yossef et al. [BYKS02], in their range-efficient F_0 algorithm, use the Toeplitz family of hash functions [Gol97], and their algorithm uses specific properties of those hash functions.

3. Range sampling algorithm. In this section, we present a solution to the range sampling problem, its correctness, and time and space complexities. The algorithm $RangeSample(r_i = [x_i, y_i], \ell)$ computes the number of points $x \in [x_i, y_i]$ for which $h(x) \in R_\ell$, where $h(x) = (ax + b) \mod p$. This algorithm is later used as a subroutine by the range-efficient algorithm for F_0, which is presented in the next section:

$$RangeSample([x_i, y_i], \ell) = \left|\left\{x \in [x_i, y_i] | h(x) \in \left[0, \left\lfloor \frac{p}{2^\ell} \right\rfloor - 1\right]\right\}\right|.$$

Note that the sequence $h(x_i), h(x_i + 1), \ldots, h(y_i)$ is an arithmetic progression over $\mathbb{Z}_p$ ($\mathbb{Z}_k$ is the ring of integers modulo k) with a common difference a. Thus, we consider the following general problem.

PROBLEM 1. *Let $M > 0$, $0 \leq d < M$, and $0 < L \leq M$. Let $S = \langle u = x_1, x_2, \ldots, x_n\rangle$ be an arithmetic progression over $\mathbb{Z}_M$ with common difference d, i.e, $x_i = (x_{i-1} + d) \mod M$. Let R be a region of the form $[0, L-1]$ or $[-L+1, 0]$. Compute $|S \cap R|$.*

We first informally describe the idea behind our algorithm for Problem 1 and then go on to the formal description. For this informal description, we consider only the case $R = [0, L-1]$.

We first define a total order among the elements of $\mathbb{Z}_m$. Observe that an element of $\mathbb{Z}_m$ has infinitely many representations. For example, if $x \in \mathbb{Z}_m$, then $x, x+m, x+2m, \ldots$ are possible representations of x. For $x \in Z_m$, the standard representation of x is the smallest nonnegative integer $std(x)$ such that $x \equiv std(x)$ in $\mathbb{Z}_m$. If x and y are two elements of $\mathbb{Z}_m$, then we say $x < y$ if $std(x)$ is less than $std(y)$ in the normal ordering of integers. We say $x > y$ if $std(x) \neq std(y)$ and $x \not< y$. In the rest of the paper, we use the standard representation for elements over $\mathbb{Z}_m$. Given an element $x \in \mathbb{Z}_m$, we define $-x$ as $m - x$.

3.1. Intuition. Divide S into subsequences $S_0, S_1, \ldots, S_k, S_{k+1}$ as follows: $S_0 = \langle x_1, x_2, \ldots, x_i\rangle$, where i is the smallest natural number such that $x_i > x_{i+1}$. The subsequences $S_j, j > 0$ are defined inductively. If $S_{j-1} = \langle x_t, x_{t+1}, \ldots, x_m\rangle$, then $S_j = \langle x_{m+1}, x_{m+2}, \ldots, x_r\rangle$, where r is the smallest number such that $r > m+1$ and $x_r > x_{r+1}$; if no such r exists, then $x_r = x_n$. Note that if $S_j = \langle x_t, x_{t+1}, \ldots, x_m\rangle$ then $x_t < d$ and $x_t, x_{t+1}, \ldots, x_m$ are in ascending order. Let f_j denote the first element of S_j, and let e_j denote the last element of S_j.

We treat the computation of $|S_0 \cap R|$ and $|S_{k+1} \cap R|$ as special cases and now focus on the the computation of $|S_i \cap R|$ for $1 \leq i \leq k$. Let $L = d \times q + r$, where $r < d$. Note that the values of q and r are unique. We observe that, for each $i \in [1, k]$, it is easy to compute the number of points of S_i that lie in R. More precisely we make the following observation.

Observation 1. For every $i \in [1, k]$, if $f_i < r$, then $|S_i \cap R| = \lfloor \frac{L}{d} \rfloor + 1$, else $|S_i \cap R| = \lfloor \frac{L}{d} \rfloor$.

Thus Problem 1 is reduced to computing the size of the following set:

$$\{i \mid 1 \leq i \leq k, f_i \in [0, r-1]\}.$$

The following observation is critical.

Observation 2. The sequence $\langle f_1, f_2, \ldots, f_k \rangle$ forms an arithmetic progression over $\mathbb{Z}_d$.

We show in Lemma 2 that the common difference of this sequence is $d - r'$, where $r' = M \bmod d$. Thus, we have reduced the original problem (Problem 1) with a common difference of d to a smaller problem, whose common difference is $d - r'$. However, this reduction may not always be useful since $d - r'$ may be not be much smaller than d. However, we now show that it is always possible to get a reduction to a subproblem with a significantly smaller common difference.

We can always view any arithmetic progression over $\mathbb{Z}_d$ with common difference $d - r'$ as an arithmetic progression with common difference $-r'$. The next crucial observation is as follows.

Observation 3. At least one of $d - r'$ or r' is less than or equal to $d/2$, and we can choose to work with the smaller of $d - r'$ or r'.

Thus, we have reduced Problem 1 to a smaller problem, whose common difference is at most half the common difference of Problem 1. Proceeding thus, we get a recursive algorithm whose time complexity is logarithmic in d.

3.2. Formal description. We start with the following useful lemmas.

LEMMA 1. *If* $R = [0, L-1]$, *then for* $1 \le i \le k$,

$$|S_i \cap R| = \begin{cases} \lfloor \frac{L}{d} \rfloor + 1 & \text{if } f_i \in [0, r-1], \\ \lfloor \frac{L}{d} \rfloor & \text{if } f_i \notin [0, r-1]. \end{cases}$$

If $R = [-L+1, 0]$, *then*

$$|S_i \cap R| = \begin{cases} \lfloor \frac{L}{d} \rfloor + 1 & \text{if } e_i \in [-r+1, 0], \\ \lfloor \frac{L}{d} \rfloor & \text{if } e_i \notin [-r+1, 0]. \end{cases}$$

Note that each f_i is less than d, and so we can view the f_i's as elements over $\mathbb{Z}_d$. Below we show that the f_i's form an arithmetic progression over $\mathbb{Z}_d$.

LEMMA 2. *Let* $M = d \times q' + r'$, *where* $r' < d$. *Then, for* $1 \le i < k$, $f_{i+1} = (f_i - r') \bmod d$.

Proof. Recall that $S_i = \langle f_i, f_i + d, \ldots, e_i \rangle$ and $M - d \le e_i \le M - 1$. We have two cases.

If $f_i < r'$, then $e_i = f_i + q' \times d$. Thus

$$\begin{aligned} f_{i+1} &= (e_i + d) \mod M \\ &= (f_i + q' \times d + r' + d - r') \mod M \\ &= (f_i + M + (d - r')) \mod M \\ &= (d - (r' - f_i)) \mod M. \end{aligned}$$

Since f_{i+1} is less than d, the final expression for f_{i+1} can be written in $\mathbb{Z}_d$ as follows: $f_{i+1} = (f_i + (d - r')) \mod d = (f_i - r') \mod d$.

In the second case, if $f_i \ge r'$, then $e_i = (f_i + (q' - 1) \times d) \mod M$. Thus, $f_{i+1} = (f_i + q' \times d) \mod M = (f_i - r') \mod M$.

Since f_{i+1} is less than d, the above can be written in $\mathbb{Z}_d$ as $f_{i+1} = (f_i - r') \bmod d$. ∎

Note that similar to the f_i's, the e_i's are also restricted to having a value in a range of length d, since $M - 1 \ge e_i \ge M - d$. However, the e_i's do not directly form an arithmetic progression over Z_d. Thus, we define a function $map : \{M - d, M - d + 1, \ldots, M - 1\} \to \mathbb{Z}_d$ such that $map(e_i)$'s form an arithmetic progression over $\mathbb{Z}_d$.

DEFINITION 2. *For $M - d \leq x < M$, $map(x) = M - x - 1$.*

We now present the following lemma for $map(e_i)$'s, which is similar to Lemma 2. We omit the proof since it is similar to the proof of Lemma 2.

LEMMA 3. *Let $M = d \times q' + r'$, where $r' < d$. Then, for $1 < i \leq k$, $map(e_i) = (map(e_{i-1}) + r') \mod d$.*

Next we argue that, for our purposes, any arithmetic progression with common difference $-r'$ can be viewed as a different arithmetic progression with common difference r'. Let $T = \langle y_1, y_2, \ldots, y_k \rangle$ be an arithmetic progression over $\mathbb{Z}_t$ with a common difference $-s$, i.e.,

$$y_i = (y_{i-1} - s) \bmod t.$$

Let R be of the form $[0, L-1]$ or $[-L+1, 0]$. Define R' as follows: if $R = [0, L-1]$, then $R' = [-L+1, 0]$, else $R' = [0, L-1]$.

LEMMA 4. *Let $T' = \langle y'_1, y'_2, \ldots, y'_k \rangle$ be an arithmetic progression over $\mathbb{Z}_t$ defined as $y'_1 = -y_1$, and for $1 < i < k$, $y'_i = (y'_{i-1} + s) \bmod t$. Then,*

$$|T \cap R| = |T' \cap R'|.$$

Proof. If $y_1 - ks = x \mod t$, then $-y_1 + ks = -x \mod t$. Thus $x \in R$ if and only if $-x \in R'$. ∎

3.2.1. Description of the algorithm. Now we describe our algorithm for Problem 1.

PROCEDURE `Hits`(M, d, u, n, R).

Precondition: R is of the form $[0, L-1]$ or $[-L+1, 0]$, where $L \leq M$, and $u < M, d < M$.

Goal: Compute $|S \cap R|$, where

$$S = \langle u, (u+d) \mod M, \ldots, (u + n \times d) \mod M \rangle.$$

1. (a) `If` $n = 0$, `then` **Return** 1 `if` $u \in R$, `else` **Return** 0.
   (b) `Let` $S = S_0, S_1, \ldots, S_k, S_{k+1}$. `Compute` k.
   (c) $\text{Hits}_0 \leftarrow |S_0 \cap R|$. $\text{Hits}_{k+1} \leftarrow |S_{k+1} \cap R|$.
   (d) `If` $d = 1$, `then`
   Return $\text{Hits}_0 + \text{Hits}_{k+1} + (L \times k)$.
2. `Compute` r `and` r' `such that` $L = d \times q + r$, $r < d$ and $M = d \times q' + r'$ and $r' < d$.
3. `If` $R = [0, L-1]$, `then`
   3.1. `Compute` f_1, where f_1 is the first element of S_1.
   3.2. $u_{new} \leftarrow f_1$, $M_{new} \leftarrow d, n_{new} \leftarrow k$.
   3.3. `If` $d - r' \leq d/2$, `then` $R_{new} \leftarrow [0, r-1]$ and $d_{new} \leftarrow d - r'$.
   Comment: In this case, we view the f_i's as arithmetic progression over $\mathbb{Z}_d$ with common difference $d - r'$.
   3.4. High = Hits(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).
   Comment: By making a recursive call to Hits, we are computing High, the cardinality of the set $\{i \mid f_i \in [0, r-1], 1 \leq i \leq k\}$.
   3.5. `If` $d - r' > d/2$ (so $r' \leq d/2$), `then` $u_{new} \leftarrow -f_1$, $d_{new} \leftarrow r'$, and $R_{new} \leftarrow [-r+1, 0]$.
   Comment: In this case we consider the f_i's as an arithmetic progression over $\mathbb{Z}_d$ with common difference $-r'$.
   3.6. High = Hits(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).

3.7. Low $\leftarrow (k -$ High). **Return**

$$\text{Hits}_0 + \text{Hits}_{k+1} + \left(\text{High} \cdot \left(\left\lfloor \frac{L}{d} \right\rfloor + 1 \right) \right) + \left(\text{Low} \cdot \left\lfloor \frac{L}{d} \right\rfloor \right).$$

4. `If` $R = [-L+1, 0]$ `then`
   4.1. `Compute` e_1 and $map(e_1)$, where e_1 is the last element of S_1.
   4.2. $u_{new} \leftarrow map(e_1), M_{new} \leftarrow d, n_{new} \leftarrow k$.
   4.3. `If` $r' \leq d/2$, `then` $R_{new} \leftarrow [0, r-1]$, and $d_{new} \leftarrow r'$.
   Comment: In this case we view $map(e_i)$'s as an arithmetic progression over $\mathbb{Z}_d$ with common difference r'.
   4.4. High = Hits(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).
   4.5. `If` $r' > d/2$ (so $d - r' \leq d/2$), `then` $u_{new} \leftarrow -map(e_1), d_{new} \leftarrow d - r'$, and $R_{new} \leftarrow [-r+1, 0]$.
   Comment: In this case we view $map(e_i)$'s as an arithmetic progression over $\mathbb{Z}_d$ with common difference $-r'$.
   4.7. High = Hits(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).
   4.8. Low $\leftarrow (k -$ High). **Return**

$$\text{Hits}_0 + \text{Hits}_{k+1} + \left(\text{High} \cdot \left(\left\lfloor \frac{L}{d} \right\rfloor + 1 \right) \right) + \left(\text{Low} \cdot \left\lfloor \frac{L}{d} \right\rfloor \right).$$

3.3. Correctness of Hits.

THEOREM 1. *Given M, d, u, n, R, Algorithm Hits correctly computes the solution to Problem* 1.

Proof. We first consider the case $R = [0, L-1]$. It is easy to verify that when $d = 1$ or when $n = 0$ the algorithm correctly computes the answer. Note that

$$|S \cap R| = |S_0 \cap R| + |S_{k+1} \cap R| + \sum_{i=1}^{i=k} |S_i \cap R|.$$

Step 1 correctly computes $|S_0 \cap R|$ and $|S_{k+1} \cap R|$. By Lemma 1, for $1 \leq i \leq k$, $|S_i \cap R|$ is $\lfloor \frac{L}{d} \rfloor + 1$ if $f_i \in [0, r-1]$ and is $\lfloor \frac{L}{d} \rfloor$ if $f_i \notin [0, r-1]$. Let High denote the number of f_i's for which $f_i \in [0, r-1]$. Given the correct value of High, the algorithm correctly computes the final answer in step 3.4.

Thus the goal is to show that the algorithm correctly computes the value of High. Let $T = \langle f_1, \ldots, f_k \rangle$. At this point, the algorithm considers two cases.

If $d - r' \leq d/2$, then the algorithm is required to compute the number of elements in T that lie in $[0, r-1]$. By Lemma 2, T is an arithmetic progression over $\mathbb{Z}_d$, with common difference $d - r'$. The starting point of this progression is f_1; the number of elements in the progression is k. Thus the algorithm makes a correct recursive call to Hits.

If $d - r' > d/2$, then by Lemma 2 T is an arithmetic progression over $\mathbb{Z}_d$ with the common difference $-r'$. The algorithm is required to compute $|T \cap [0, r-1]|$.

Define a new sequence $T' = \langle f'_1, f'_2, \ldots, f'_k \rangle$ over $\mathbb{Z}_d$ as follows: $f'_1 = -f_1$ and $f'_{i+1} = (f'_i + r') \mod d, 1 \leq i \leq k-1$. By Lemma 4,

$$|T \cap [0, r-1]| = |T' \cap [-r+1, 0]|.$$

Thus the algorithm makes the correct recursive call in step 3.4 to compute $|T' \cap [-r+1, 0]|$ which equals $|T \cap [0, r-1]|$.

Thus the algorithm correctly computes $S \cap R$ when R is of the form $[0, L-1]$. Correctness for the case when R is of the form $[-L+1, 0]$ follows similarly. ∎

3.4. Complexity of Hits. For the time complexity, we assume that all arithmetic operations in Z_k, including addition, multiplication, and division, take unit time.

THEOREM 2. *The time complexity of* Hits(M,d,u,n,R) *is* $O(\min\{\log d, \log n\})$, *and the space complexity is* $O(\log M + \log n)$.

Proof. Time complexity: It is clear that steps 1 and 2 can be completed using a constant number of operations. The algorithm makes a recursive call in step 3 or 4. Also, it is clear that $d_{new} \leq d/2$. Thus if we parametrize the running time of the algorithm with d, then

$$\begin{aligned} T(d) &= T(d_{new}) + O(1) \\ &\leq T(d/2) + O(1) = O(\log d). \end{aligned}$$

We can get a better bound on the running time as follows: The input parameters to the recursive procedure are M, d, u, n, R. When the recursive procedure is called, the parameters are $M_{new} = d$, $d_{new} \leq d/2$, $n_{new} \leq \lceil n \cdot d/M \rceil$.

In every recursive call except (perhaps) the first, it must be true that $d \leq M/2$. Thus, in every recursive call except for the first, $n_{new} \leq n/2$. If we parametrize the running time on n, then the total time of the algorithm is $O(\log(n))$.

Space complexity: Whenever the Hits algorithm makes a recursive call, it needs to store values of a constant number of local variables such as Hits_0, Hits_{k+1}, n, etc. Since M dominates u, d, and L, each time the algorithm makes a recursive call, it needs $O(\log n + \log M)$ of stack space. Since the depth of the recursion is no more than $\log n$, the total space needed is $O(\log n \cdot (\log n + \log M))$. We can further reduce the space by a careful implementation of the recursive procedure as follows.

In general $\text{Hits}(p_1, p_2, \ldots, p_5) = \beta + \gamma\ \text{Hits}(p'_1, p'_2, \ldots, p'_5)$, where β and γ are functions of $p_1, \ldots, p_5$. This is a tail-recursive procedure, which can be implemented without having to allocate space for a new stack for every recursive call and without having to tear down the stack upon a return. Thus, the total space can be reduced to $O(\log n + \log M)$. ∎

4. Algorithm for range-efficient F_0. We now describe the complete algorithm for estimating F_0, using the range sampling algorithm as a subroutine. We then present its correctness and time and space complexities.

From section 2, recall the following notation: The input stream is $r_1, r_2, \ldots, r_m$, where each stream element $r_i = [x_i, y_i] \subset [1, n]$ is an interval of integers $x_i, x_i + 1, \ldots, y_i$. Integer p is a prime number between $10n$ and $20n$. For each level $\ell = 0, \ldots, \lceil \log p \rceil$, the following hold:

1. $R_\ell = \{0, \ldots, \lfloor \frac{p}{2^\ell} \rfloor - 1\}$.
2. For every $x \in [1, n]$, the sampling function at level ℓ, $S(x, \ell)$, is defined as $S(x, \ell) = 1$ if $h(x) = 1$ and $S(x, \ell) = 0$ otherwise.
3. The sampling probability at level ℓ is $P_\ell = |R_\ell|/p$.

4.1. Algorithm description. A formal description of the algorithm appears in Figure 1. The algorithm does not directly yield an (ϵ, δ)-estimator for F_0 but instead gives an estimator which is within a factor of ϵ of F_0 with a constant probability. Finally, by taking the median $O(\log \frac{1}{\delta})$ of such estimators, we get an (ϵ, δ)-estimator.

The random sample. The algorithm maintains a sample S of the intervals seen so far. S is initially empty. The maximum size of S is $\alpha = \frac{60}{\epsilon^2}$ intervals. The algorithm

Initialization.

1. Choose a prime number p such that $10n \leq p \leq 20n$, and choose two numbers a, b at random from $\{0, \ldots, p-1\}$.
2. $S \leftarrow \phi$.
3. $\ell \leftarrow 0$.

When a new interval $r_i = [x_i, y_i]$ arrives:

1. If r_i intersects with any interval in S, then
   (a) While there is an interval $r \in S$ such that $r \cap r_i \neq \phi$,
      i. $S \leftarrow S - r$;
      ii. $r_i \leftarrow r_i \cup r$.
   (b) $S \leftarrow S \cup r_i$.
2. Else If $RangeSample(r_i, \ell) > 0$, then
   (a) $S \leftarrow S \cup \{r_i\}$ // insert into sample;
   (b) While ($|S| > \alpha$) // overflow,
      i. $\ell \leftarrow \ell + 1$.
         If $\ell > \lceil \log p \rceil$, then // maximum level
                                // reached, algorithm fails
            return;
      ii. $S \leftarrow \{r \in S | RangeSample(r, \ell) > 0\}$.

When an estimate for F_0 is asked for: Return $\sum_{r \in S} RangeSample(r, \ell)/P_\ell$

FIG. 1. *The range-efficient F_0 algorithm using the range sampling algorithm as a subroutine.*

also has a current sampling level ℓ, which is initialized to 0. We maintain the following invariants for the random sample S.

Invariant 1. All the intervals in S are disjoint.

Invariant 2. Every interval in S has at least one element selected into the sample at the current sampling level.

When a new interval $r_i = [x_i, y_i]$ arrives. The algorithm first checks if r_i intersects with any currently existing interval in S. If so, it deletes all the intersecting intervals from S, merges all of them with r_i to form a single interval, and inserts the resulting interval into S. If r_i does not intersect with any currently existing interval, it calls the range sampling subroutine to determine if *any* point in $[x_i, y_i]$ belongs in S at level ℓ. If yes, then the whole interval is stored in the sample.

Overflow. It is possible that after adding a new element to the sample, the size of S exceeded α. In such a case, the algorithm increases its sampling level and subsamples the elements of the current sample into the new level. This subsampling is repeated until either the size of the sample becomes less than α or the maximum sampling level is reached (i.e., $\ell = \lceil \log p \rceil$).

Estimating F_0. When an estimate is asked for, the algorithm uses the range sampling routine to determine the size of the current sample and returns this value boosted by $1/P_\ell$, where ℓ is the current sampling level.

4.2. Proof of correctness. The proof follows a parallel structure to the proof by Gibbons and Tirthapura [GT01]. We give the complete correctness proof here because the hash functions used here are different from those in [GT01] and also for the sake of completeness.

FACT 1. *For any $\ell \in [0, \ldots, \lceil \log p \rceil]$, $1/2^{\ell+1} \leq P_\ell \leq 1/2^\ell$.*

FACT 2. *For any $\ell \in [0, \ldots, \lceil \log p \rceil]$, the random variables $\{S(x,\ell) | x \in [1,n]\}$ are all pairwise independent.*

LEMMA 5. *Invariants* 1 *and* 2 *are true before and after the processing of each interval in the stream.*

Proof. We prove by induction on the number of intervals that have been processed. The base case is clear, since S is initialized to ϕ. Suppose a new interval r arrives. If r intersects with any interval already in S, then our algorithm clearly maintains Invariant 1, and it can be verified that Invariant 2 also holds. If r does not intersect with any element already in S, then it is included in the sample if and only if r has at least one element which would be sampled at the current sampling level. This ensures that Invariant 2 is maintained. It can be easily verified that the steps taken to handle an overflow also maintain the invariants. ∎

Let random variable Z denote the result of the algorithm. We analyze the algorithm by looking at the following hypothetical process. This process is useful for us only to visualize the proof and is not executed by the algorithm. The stream of intervals R is "expanded" to form the stream I of the constituent integers. For each interval $[x_i, y_i] \in I$, the integer stream I consists of $x_i, x_i + 1, \ldots, y_i$.

Let $D(I)$ denote the set of all distinct elements in I. We want to estimate $F_0 = |D(I)|$. Each element in $D(I)$ is placed in different levels as follows. All the elements of $D(I)$ are placed in level 0. An element $x \in D(I)$ is placed in every level $\ell > 0$ such that $S(x,\ell) = 1$. For level $\ell = 0 \ldots \lceil \log p \rceil$, let X_ℓ denote the number of distinct elements placed in level ℓ.

LEMMA 6. *$E[X_\ell] = F_0 P_\ell$, and $Var[X_\ell] = F_0 P_\ell (1 - P_\ell)$.*

Proof. By definition, $X_\ell = \sum_{x \in D(I)} S(x,\ell)$. Thus, $E[X_\ell] = \sum_{x \in D(I)} E[S(x,\ell)] = |D(I)| P_\ell = F_0 P_\ell$. Because the random variables $\{S(x,\ell) | x \in D(I)\}$ are all pairwise independent (Fact 2), the variance of their sum is the sum of their variances, and the expression for the variance follows. ∎

Let ℓ^* denote the smallest integer $\ell \geq 0$ such that $X_\ell \leq \alpha$. Let ℓ' denote the level at which the algorithm finally ends.

LEMMA 7. *The algorithm returns $Z = X_{\ell'}/P_{\ell'}$ and $\ell' \leq \ell^*$.*

Proof. We first show that $\ell' \leq \ell^*$. If the algorithm never increased its sampling level, then $\ell' = 0$ and thus $\ell' \leq \ell^*$ trivially. Suppose $\ell' \geq 1$. Thus the algorithm must have increased its sampling level from $\ell' - 1$ to ℓ'. The increase in level must have been due to an overflow, and the number of (disjoint) intervals at level $\ell' - 1$ must have been more than α. Since, due to Invariant 2, each interval in the sample at level $\ell' - 1$ has at least one point x such that $S(x, \ell' - 1) = 1$, it must be true that $X_{\ell'-1} > \alpha$, and similarly it follows that $X_\ell > \alpha$ for any $\ell < \ell'$. However, by definition of ℓ^*, it must be true that $X_{\ell^*} \leq \alpha$. Thus it must be true that $\ell^* \geq \ell'$.

Next, we show that the algorithm returns $X_{\ell'}/P_{\ell'}$. Consider the set $S' = \{x \in D(I) | S(x,\ell') = 1\}$. Consider some element $x \in S'$. Integer x must have arrived as a part of some interval $r \in R$. Either r must be in S (if r did not intersect with any range already in the sample and was not later merged with any other interval), or there must be an interval $r' \in S$ such that $r \subset r'$. In both cases, x is included in S. Further, because of Invariant 1, x will be included in exactly one interval in S and will be counted (exactly once) as a part of the sum $\sum_{r \in S} RangeSample(r, \ell')$. Conversely, an element y such that $S(y,\ell') = 0$ will not be counted in the above sum. Thus, the return value of the algorithm is exactly $|S'|/P_\ell$, which is $X_{\ell'}/P_{\ell'}$. ∎

Define a level ℓ to be *good* if X_ℓ/P_ℓ is within ϵ relative error of F_0. Otherwise, level ℓ is *bad*. For each $\ell = 0, \ldots, \lceil \log p \rceil$, let B_ℓ denote the event that level ℓ is bad, and let T_ℓ denote the event that the algorithm stops at level ℓ.

THEOREM 3.

$$\Pr\{Z \in [(1-\epsilon)F_0, (1+\epsilon)F_0]\} \geq 2/3.$$

Proof. Let P denote the probability that the algorithm fails to produce a good estimate. This happens if any of the following is true:

- The maximum level $max = \lceil \log p \rceil$ is reached.
- The level at which the algorithm stops is a bad level, i.e., only for some $\ell \in \{0, \ldots, \lceil \log p \rceil - 1\}$, T_ℓ and B_ℓ are both true.

Thus

$$P = \Pr[T_{max}] + \sum_{\ell=0}^{max-1} \Pr[T_\ell \wedge B_\ell].$$

Let ℓ_m denote the lowest numbered level ℓ such that $E[X_\ell] < \alpha/2$. We prove that ℓ_m exists and $0 \leq \ell_m < max$.

Using Lemma 6, we first note that $E[X_{max}] = P_{max}F_0$. Using Fact 1 and $max = \lceil \log p \rceil$, we get $P_{max} \leq 1/p$. Since $F_0 \leq n$, we get $E[X_{max}] \leq n/p \leq 1/10$; since p is a prime number between $10n$ and $20n$:

$$E[X_{max}] \leq \frac{1}{10}. \tag{1}$$

Since $\alpha = 60/\epsilon^2$, we have $E[X_{max}] < \alpha/2$. Thus ℓ_m exists.

Next, we argue that $\ell_m < max$. By definition of ℓ_m, $E[X_{\ell_m - 1}] = P_{\ell_m - 1}F_0 \geq \alpha/2$. From Fact 1, we know that $P_{\ell_m - 1} \leq 4P_{\ell_m}$. Thus, $E[X_{\ell_m}] = P_{\ell_m}F_0 \geq (1/4)\alpha/2 = 60\epsilon^2/8 > 7$:

$$E[X_{\ell_m}] > 7. \tag{2}$$

From (1) and (2), we have $\ell_m < max$.

We now bound P as

$$\begin{aligned} P &= \Pr[T_{max}] + \sum_{\ell=\ell_m+1}^{max-1} \Pr[T_\ell \wedge B_\ell] + \sum_{\ell=0}^{\ell_m} \Pr[T_\ell \wedge B_\ell] \\ &\leq \Pr[T_{max}] + \sum_{\ell=\ell_m+1}^{max-1} \Pr[T_\ell] + \sum_{\ell=0}^{\ell_m} \Pr[B_\ell] \\ &\leq \sum_{\ell=\ell_m+1}^{max} \Pr[T_\ell] + \sum_{\ell=0}^{\ell_m} \Pr[B_\ell]. \end{aligned}$$

In Lemma 8 we show that $\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] < 16/60$, and in Lemma 9 we show that $\sum_{\ell=\ell_m+1}^{max} \Pr[T_\ell] < 1/30$. Putting them together, the theorem is proved. ∎

LEMMA 8. $\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] < 16/60$.

Proof. Let μ_ℓ and σ_ℓ denote the mean and standard deviation of X_ℓ, respectively. Then,

$$\Pr[B_\ell] = \Pr|X_\ell - \mu_\ell| \geq \epsilon\mu_\ell.$$

Using Chebyshev's inequality, $\Pr|X_\ell - \mu_\ell| \geq t\sigma_\ell \leq \frac{1}{t^2}$ and substituting $t = \frac{\epsilon\mu_\ell}{\sigma_\ell}$, we get

$$\Pr[B_\ell] \leq \frac{\sigma_\ell^2}{\epsilon^2\mu_\ell^2}. \tag{3}$$

Substituting values from Lemma 6 into (3), we get

$$\Pr[B_\ell] \leq \frac{1 - P_\ell}{\epsilon^2 P_\ell F_0} < \frac{1}{F_0 \epsilon^2 P_\ell}.$$

Thus,

$$\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] = \frac{1}{F_0 \epsilon^2} \sum_{\ell=0}^{\ell_m} \frac{1}{P_\ell}.$$

From Fact 1, we have $1/P_\ell \leq 2^{\ell+1}$. Thus, we have

$$\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] \leq \frac{1}{F_0 \epsilon^2} \sum_{\ell=0}^{\ell_m} 2^{\ell+1} < \frac{1}{F_0 \epsilon^2} 2^{\ell_m+2}.$$

By definition, ℓ_m is the lowest numbered level ℓ such that $E[X_\ell] < \alpha/2$. Thus, $F_0 P_{\ell_m - 1} = E[X_{\ell_m - 1}] \geq \alpha/2$. Using Fact 1, we get $F_0/2^{\ell_m - 1} \geq \alpha/2$. It follows that

$$\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] \leq \frac{4}{\epsilon^2} \frac{2^{\ell_m}}{F_0} \leq \frac{4}{\epsilon^2} \frac{4}{\alpha} < \frac{16}{60}$$

by using $\alpha = \frac{60}{\epsilon^2}$. ∎

LEMMA 9. $\sum_{\ell=\ell_m+1}^{max} \Pr[T_\ell] < 1/30$.

Proof. Let $P_s = \sum_{\ell=\ell_m+1}^{max} \Pr[T_\ell]$. We see that P_s is the probability that the algorithm stops in level $\ell_m + 1$ or greater. This implies that at level ℓ_m there were at least α intervals in the sample S. Invariant 2 implies that there were at least α elements sampled at that level, so that $X_{\ell_m} \geq \alpha$.

$$\begin{aligned} P_s &\leq \Pr[X_{\ell_m} \geq \alpha] \\ &= \Pr\left[X_{\ell_m} - \mu_{\ell_m} \geq \alpha - \frac{\alpha}{2}\right] \\ &\leq \frac{\sigma_{\ell_m}^2}{\alpha^2 (1/2)^2}. \end{aligned}$$

From Lemma 6, we get $\sigma_{\ell_m}^2 < E[X_{\ell_m}] < \alpha/2$. Using this in the above expression, we get

$$P_s \leq \frac{2}{\alpha} = \frac{2\epsilon^2}{60} < \frac{1}{30}.$$

The last inequality is obtained by using $\epsilon < 1$. ∎

4.3. Time and space complexity. In this section we prove bounds on the time and space complexity of the algorithm.

LEMMA 10. *The space complexity of the range-efficient* (ϵ, δ)*-estimator for* F_0 *is* $O(\frac{\log 1/\delta \log n}{\epsilon^2})$.

Proof. The workspace required is the space for the sample S plus the workspace for the range sampling procedure *RangeSample*. Sample S contains α intervals, where each interval can be stored using two integers, thus taking $2 \log n$ bits of space. The workspace required by the range sampling procedure is $O(\log n)$ bits, so that the

total workspace is $O(\alpha \log n + \log n) = O(\frac{\log n}{\epsilon^2})$. Since the we need to run $O(\log 1/\delta)$ instances of this algorithm, the total space complexity is $O(\frac{\log 1/\delta \log n}{\epsilon^2})$. We note that the space complexity is identical to that of the single-item case, where each data item is a single number instead of a range. □

As noted in section 3, we assume that arithmetic operations, including addition, multiplication, and division, in $\mathbb{Z}_k$ take unit time.

LEMMA 11. *The amortized time taken to process an interval $r = [x, y]$ by the range-efficient (ϵ, δ)-estimator for F_0 is $O(\log(n/\epsilon) \log 1/\delta)$.*

Proof. The time to handle a new interval $r = [x, y]$ consists of three parts:

1. time for checking if r intersects any interval in the sample;
2. time required for range sampling (from Theorem 2 this is $O(\log(y-x))$, which is always $O(\log n)$, and perhaps is much smaller than $\Theta(\log n)$).
3. time for handling an overflow in the sample.

We now analyze the time for the first and third parts.

First part. This is the time to check if the interval intersects any of the $O(1/\epsilon^2)$ intervals already in the sample and to merge them, if necessary. This takes $O(1/\epsilon^2)$ time if done naively. This can be improved to $O(\log(1/\epsilon))$ amortized time as follows: Since all the intervals in S are disjoint, we can define a linear order among them in the natural way. We store S in a balanced binary search tree $T(S)$ augmented with in-order links. Each node of $T(S)$ is an interval, and by following the in-order links, we can get the sorted order of S. When a new interval $r = [x, y]$ arrives, we first search for the node in $T(S)$ which contains x. There are three cases possible:

(a) Interval r does not intersect any interval in S. In this case, r is inserted into the tree, which takes $O(\log(1/\epsilon))$ time.

(b) Interval r intersects some interval in S, and there is an interval $t \in S$ which contains x. In such a case, by following the in-order pointers starting from t, we can find all the intervals that intersect r. All these intervals are merged together with r to form a single new interval, say r'. We delete all the intersecting intervals from $T(S)$, and insert r' into $T(S)$. Since each interval is inserted only once and is deleted at most once, the time spent in finding and deleting each intersecting interval can be charged to the insertion of the interval. Thus, the amortized time for handling r is the time for searching for x plus the time to insert r'. Since $|S| = O(1/\epsilon^2)$, this time is $O(\log(1/\epsilon))$.

(c) Interval r intersects some intervals in S, but none of them contain x. This is similar to case (b).

Third part. This is the time for handling an overflow, subsampling to a lower level. For each change of level, we will have to apply the range sampling subroutine $O(1/\epsilon^2)$ times. Each change of level selects roughly half the the number of points belonging in the previous level into the new level. However, since each interval in the sample may contain many points selected in the current level, it is possible that more than one level change may be required to bring the sample size to less than α intervals.

However, we observe that the total number of level changes (over the whole data stream) is less than $\lceil \log p \rceil$ with high probability, since the algorithm does not reach level $\lceil \log p \rceil$ with high probability. Since $\log p = \Theta(\log n)$, the total time taken by level changes over the whole data stream is $O(\frac{\log^2 n}{\epsilon^2} \log \frac{1}{\delta})$. If the length of the data stream dominates the above expression, then the amortized cost of handling the overflow is $O(1)$.

Thus, the amortized time to handle an interval $r = [x, y]$ per instance of the algorithm is $O(\log(y-x) + \log(1/\epsilon))$. Since there are $O(\log 1/\delta)$ instances, the amortized time per interval is $O(\log((y-x)/\epsilon) \log 1/\delta)$ which is also $O(\log(n/\epsilon) \log 1/\delta)$.

It follows that the worst-case processing time per item is $O(\frac{\log^2 n}{\epsilon^2} \log \frac{1}{\delta})$. If our focus was on optimizing the worst-case processing time per item, then we could reduce the above to $O(\frac{\log \log n \log(y-x)}{\epsilon^2} \log \frac{1}{\delta})$ by changing levels using a binary search rather than sequentially when an overflow occurs. ∎

LEMMA 12. *The algorithm can process a query for F_0 in $O(1)$ time.*

Proof. We first describe how to process a query for F_0 in $O(\log 1/\delta)$ time. At first glance, it seems necessary to apply the range sampling procedure on α intervals and compute the sum. However, we can do better as follows: With each interval in S, store the number of points in the interval which are sampled at the current sampling level. This is first computed either when the interval was inserted into the sample or when the algorithm changes levels. Further, the algorithm also maintains the current value of the sum $\sum_{r \in S} \mathit{RangeSample}(r, \ell)/P_\ell$, where ℓ is the current level and S is the current sample. This sum is updated every time a new interval is sampled or when the algorithm changes level. The above changes do not affect the asymptotic cost of processing a new item. Given this, an F_0 query can be answered for each instance of the algorithm in constant time. Since it is necessary to compute the median of many instances of the algorithm, the time to answer an F_0 query is $O(\log 1/\delta)$.

Given that $O(\log 1/\delta)$ is asymptotically less than the time required to process a new interval, the algorithm can always update the estimate of F_0 while processing a new interval, without affecting the asymptotic cost of processing an interval. If such an estimate of F_0 is maintained continuously, then a query for F_0 can be answered in $O(1)$ time. ∎

5. Applications and extensions.

5.1. Dominance norms. We recall the problem here. Given input $\mathcal{I}$ consisting of k streams of m positive integers each, let $a_{i,j}, i = 1, \ldots k, j = 1 \ldots m$, represent the jth element of the ith stream. The max-dominance norm is defined as $\sum_{j=1}^{m} \max_{1 \le i \le k} a_{i,j}$. We can reduce the max-dominance norm of $\mathcal{I}$ to range-efficient F_0 of a stream $\mathcal{O}$ derived as follows: Let n denote an upper bound on $a_{i,j}$. For each element $a_{i,j} \in \mathcal{I}$, the interval $[(j-1)n, (j-1)n + a_{i,j} - 1]$ is generated in $\mathcal{O}$. It is easy to verify the following fact.

FACT 3. *The max-dominance norm of $\mathcal{I}$ equals the number of distinct elements in $\mathcal{O}$.*

Note that the elements of stream $\mathcal{O}$ can take values in the range $[0, nm-1]$. Using the range-efficient algorithm on $\mathcal{O}$, the space complexity is now $O(1/\epsilon^2(\log m + \log n) \log 1/\delta)$. Since the length of the interval in $\mathcal{O}$ corresponding to $a_{i,j} \in \mathcal{I}$ is $a_{i,j}$, from section 4.3 it follows that the amortized time complexity of handling item $a_{i,j}$ is $O(\log \frac{a_{i,j}}{\epsilon} \log \frac{1}{\delta})$ and the worst-case complexity is $O(\frac{\log \log n \log a_{i,j}}{\epsilon^2} \log \frac{1}{\delta})$.

As shown below, our algorithm for dominance norms can easily be generalized to the distributed context.

5.2. Distributed streams. In the *distributed streams model* [GT01], the data arrives as k independent streams, where for $i = 1 \ldots k$ stream i goes to party i. Each party processes its complete stream and then sends the contents of their workspace to a common referee. Similar to one-round simultaneous communication complexity, there is no communication allowed between the parties themselves. The referee is required to estimate the aggregate F_0 over the *union* of all the data streams

1 to k. The space complexity is the sum of the sizes of all messages sent to the referee.

The range-efficient F_0 algorithm can be readily adapted to the distributed streams model. All the parties share a common hash function h, and each party runs the above-described (single party) algorithm on its own stream, targeting a sample size of $c\alpha$ intervals. Finally, each party sends its sample to the referee.

It is possible that samples sent by different parties are at different sampling probabilities (or, equivalently, are at different sampling levels). The referee constructs a sample of the union of the streams by subsampling each stream to the lowest sampling probability across all the streams. By our earlier analysis, each individual stream is at a sampling probability which will likely give good estimates. Thus the final sampling probability will also give us an (ϵ, δ)-estimator for F_0. The space complexity (total space used across all nodes) of this scheme is $O(k\frac{\log 1/\delta \log n}{\epsilon^2})$, where k is the number of parties, and the time per item is the same as in the single stream algorithm.

5.3. Range-efficiency in every coordinate. If each data point is a vector of dimension d rather than a single integer, then a modified definition of range-efficiency is required. One possible definition was given by [BYKS02], *range-efficiency in every coordinate*, and this proved to be useful in their reduction from the problem of computing the number of triangles in graphs to that of computing F_0 and F_2 of an integer stream.

For vectors of dimension d, define a jth coordinate range $(a_1, \ldots, a_{j-1}, [a_{j,s}, a_{j,e}], a_{j+1}, \ldots, a_d)$ to be the set of all vectors $\hat{x}$ with the ith coordinate $x_i = a_i$ for $i \neq j$ and $x_j \in [a_{j,s}, a_{j,e}]$. An algorithm is said to be range-efficient in the jth coordinate if it can handle a jth coordinate in a range-efficient manner.

The F_0 algorithm can be made range-efficient in every coordinate in the following way: We first find a mapping g between a d-dimensional vector $a = (a_1, a_2 \ldots a_d)$ (where for all $i = 1 \ldots d$, $a_i \in [0, m-1]$) and the one-dimensional line as follows:

$g(a_1, a_2, \ldots a_d) = m^{d-1}a_1 + m^{d-2}a_2 + \cdots + m^0 a_d$.

FACT 4. *Function g has the following properties:*

- *g is an injective function.*
- *A jth coordinate range $(a_1, \ldots, a_{j-1}, [a_{j,s}, a_{j,e}], a_{j+1}, \ldots, a_d)$ maps to an arithmetic progression $g(y_1), \ldots, g(y_n)$, where $y_i \in (a_1, \ldots, a_{j-1}, [a_{j,s}, a_{j,e}], a_{j+1}, \ldots, a_d)$.*

Because of the above, the number of distinct elements does not change when we look at the stream $g(x)$ rather than stream x. Because of Fact 4 and since the range-efficient F_0 algorithm can handle an arithmetic sequence of integers rather than just intervals, it follows that the algorithm can be made range-efficient in every coordinate.

Assuming the arithmetic operations on the integers that were mapped to take $O(d)$ time, the amortized processing time per item is $O(d \log \frac{1}{\delta} \left(\log n + \frac{1}{\epsilon^2}\right))$, and the time for answering a query is $O(d \log 1/\delta)$. The workspace used is $O(d\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log n)$.

Finally, we note that the algorithm presented in this paper can handle streams in which each data item is an arithmetic progression rather than a range of consecutive integers. Let $[x_i, y_i]$ be a data item that represents an arithmetic progression $x_i, x_i + c, x_1 + 2c, \ldots y_i$. Now the range sample algorithm must compute the following:

$$RangleSample([x_i, y_i], \ell) = \left| \left\{ x \in [x_i, y_i] \mid h(x) \in \left[0, \left\lfloor \frac{p}{2^\ell} \right\rfloor - 1\right] \right\} \right|.$$

Now the sequence $h(x_i), h(x_i + c), \ldots, h(y_i)$ forms an arithmetic progression over $\mathbb{Z}_p$ with a common difference ac, instead of common difference a. The range sampling algorithm described in section 3 can handle this case.

Acknowledgments. We thank Graham Cormode for pointing out the connection to dominance norms and for suggestions which helped improve the running time of the algorithm. We thank the anonymous referees for their comments that greatly improved the presentation.

REFERENCES

[AJKS02] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar, *Approximate counting of inversions in a data stream*, in Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), 2002, pp. 370–379.

[AMS99] N. Alon, Y. Matias, and M. Szegedy, *The space complexity of approximating the frequency moments*, J. Comput. System Sci., 58 (1999), pp. 137–147.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and issues in data stream systems*, in Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS), 2002, pp. 1–16.

[BFL+06] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, *Counting triangles in data streams*, in Proceedings of the ACM Symposium on Principles of Database Systems (PODS), 2006, pp. 253–262.

[BYJK+02] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, *Counting distinct elements in a data stream*, in Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM), Lecture Notes in Comput. Sci. 2483, Springer, Berlin, 2002, pp. 1–10.

[BYKS02] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, *Reductions in streaming algorithms, with an application to counting triangles in graphs*, in Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 623–632.

[CDIM02] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan, *Comparing data streams using hamming norms (how to zero in)*, IEEE Transactions on Knowledge and Data Engineering, 15 (2003), pp. 529–540.

[CGL+05] A. R. Calderbank, A. Gilbert, K. Levchenko, S. Muthukrishnan, and M. Strauss, *Improved range-summable random variable construction algorithms*, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2005.

[CK04] D. Coppersmith and R. Kumar, *An improved data stream algorithm for frequency moments*, in Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2004, pp. 151–156.

[CLKB04] J. Considine, F. Li, G. Kollios, and J. Byers, *Approximate aggregation techniques for sensor databases*, in Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE), 2004, pp. 449–460.

[CM03] G. Cormode and S. Muthukrishnan, *Estimating dominance norms of multiple data streams*, in Proceedings of the 11th European Symposium on Algorithms (ESA), Lecture Notes in Comput. Sci. 2832, Springer, Berlin, 2003, pp. 148–160.

[CW79] J. L. Carter and M. L. Wegman, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[DGIM02] M. Datar, A. Gionis, P. Indyk, and R. Motwani, *Maintaining stream statistics over sliding windows*, SIAM J. Comput., 31 (2002), pp. 1794–1813.

[FKSV02] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, *An approximate L^1-difference algorithm for massive data streams*, SIAM J. Comput., 32 (2002), pp. 131–151.

[FM85] P. Flajolet and G. N. Martin, *Probabilistic counting algorithms for database applications*, J. Comput. System Sci., 31 (1095), pp. 182–209.

[GGI+02] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss, *Fast, small-space algorithms for approximate histogram maintenance*, in Proceedings of the ACM Symposium on Theory of Computing (STOC), 2002, pp. 389–398.

[GKMS03] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, *One-pass wavelet decompositions of data streams*, IEEE Trans. Knowl. Data Eng., 15 (2003), pp. 541–554.

[Gol97] O. Goldreich, *A Sample of Samplers: A Computational Perspective on Sampling (Survey)*, Technical report TR97-020, Electronic Colloquium on Computational Complexity, 1997.

[GT01] P. B. Gibbons and S. Tirthapura, *Estimating simple functions on the union of data streams*, in Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2001, pp. 281–291.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes, *Sampling-based estimation of the number of distinct values of an attribute*, in Proceedings of the 21st International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, CA, 1995, pp. 311–322.

[IW03] P. Indyk and D. Woodruff, *Tight lower bounds for the distinct elements problem*, in Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS), 2003, p. 283.

[JG05] H. Jowhari and M. Ghodsi, *New streaming algorithms for counting triangles in graphs*, in Proceedings of the International Conference on Computing and Combinatorics (COCOON), Lecture Notes in Comput. Sci. 3595, Springer, Berlin, 2005, pp. 710–716.

[Mut05] S. Muthukrishnan, *Data Streams: Algorithms and Applications*, Foundations and Trends in Theoretical Computer Science, Now Publishers, 2005.

[NGSA04] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson, *Synposis diffusion for robust aggregation in sensor networks*, in Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, ACM Press, New York, 2004, pp. 250–262.