Range-efficient computation of F_0 over massive data streams

A. Pavan
Dept. of Computer Science
Iowa State University
pavan@cs.iastate.edu

Srikanta Tirthapura
Dept. of Elec. and Computer Engg.
Iowa State University
snt@iastate.edu

Abstract

Efficient one-pass computation 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 the problem of efficiently estimating F_0 of a data stream where each element of the stream is an interval of integers.

We present a randomized algorithm which gives an (ϵ, δ) approximation of F_0, with the following time complexity (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 time to answer a query for F_0 is $O(\log 1/\delta)$. 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 [5], which requires $O(\frac{1}{\epsilon^5} \log \frac{1}{\delta} \log^5 n)$ processing time per item. Our algorithm can be used to compute the max-dominance norm of a stream of multiple signals, and significantly improves upon the current best bounds due to Cormode and Muthukrishnan [11]. This also provides efficient and novel solutions for data aggregation problems in sensor networks studied by Nath and Gibbons [22] and Considine et. al. [8].

1. Introduction

One of the most significant successes of research on data stream processing has been the efficient estimation of aggregates such as the frequency moments, quantiles, etc. in one-pass using limited space and time per item. An especially important aggregate is the zeroth frequency moment (F_0) of a data set, which is the number of distinct elements in the data. Most database query optimization algorithms need an estimate of F_0 [18]. 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 literature, for example in [2, 12, 16], the following model is studied. Each element in the stream is a single item (usually an integer), and the algorithm needs to process this item "efficiently", both with respect to time and space. Many algorithms have been designed in this model to estimate frequency moments [2, 16, 4, 9], and many other aggregates [13, 12, 10, 1, 21, 3] 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*. To motivate this requirement, we give two examples below.

Bar-Yossef, Kumar and Sivakumar [5] formalize the concept of *reductions* between algorithms for data streams, which leads to the need for *list-efficient* streaming algorithms. They give an algorithm for estimating the number of triangles in a graph G, where the edges of G arrive as a stream in an arbitrary order (this is referred to as an adjacency stream). Their algorithm 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 adjacency stream, the reduction produces a list of integers, and the size of this list could be as large as n, the number of vertices in the graph. If one used an algorithm which processed these integers one by one, the processing time per edge is $\Omega(n)$, which is prohibitive. Thus, this application needs an algorithm for computing F_0 and F_2 which can handle such 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 algorithms which can handle such intervals efficiently are called *range-efficient*.

Another application of such list and range-efficient algorithms is in aggregate computation over sensor networks [22, 8]. Considine et. al. [8] design sketches for the *distinct summation* problem described below. Given a multi-set of items $M = \{x_1, x_2, \ldots\}$, 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. An F_0 algorithm on this stream of l_is will give the distinct summation required. This can be solved using an algorithm for estimating F_0, but the algorithm should be *range-efficient* i.e., it should be able to process an interval of integers much faster than processing them one by one. Nath and Gibbons [22] study *duplicate insensitive sketches* for aggregate computation over sensor networks. Many duplicate insensitive sketches, such as the sketches for the approximate sum of values held by distinct sensor nodes, can be reduced to range-efficient computation of F_0.

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

Our Results: In this paper, we consider range-efficient computation of F_0. 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 $x_i, x_i + 1, \ldots y_i$. The length of an interval r_i, which is the number of integers in the interval, could be any number between 1 and n. We design an algorithm which returns an (ϵ, δ)-estimator of F_0, the total number of distinct integers contained in all the intervals in the stream R. Suppose the input stream was $[1, 10], [2, 5], [5, 12], [41, 50]$, then $F_0 = 22$.

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

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

Extensions and Applications of our Basic Algorithm: Our algorithm can be extended to the following more general scenarios:

- It can process a list of integers which is in an arithmetic progression as efficiently as it can handle an interval of integers.
- Our 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.

Dominance Norms: The problem of estimating max-dominance norms of multiple data streams is as follows. Given k streams of m 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 $\sum_{j=1}^{m} \max_{1 \le i \le k} a_{i,j}$. Assume for all $a_{i,j}$, $1 \le a_{i,j} \le n$.

The concept of max-dominance norm useful in diverse scenarios such as financial applications [23] and IP network monitoring [20]. Cormode and Muthukrishnan [11] gave a streaming algorithm for computing an $(\epsilon, \delta)-$estimator of the max-dominance norm. Their algorithm uses space $O(1/\epsilon^2(\log n + \epsilon^{-1} \log m \log\log m) \log 1/\delta)$ and processing time per item is $O(1/\epsilon^4 \log a \log m \log 1/\delta)$, where a is the value of the current element.

We show that the estimation of the max-dominance norm can be reduced to range-efficient F_0 computation, and derive an algorithm which uses $O(1/\epsilon^2(\log m + \log n) \log 1/\delta)$ space and whose amortized 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 time per item is $O(\frac{\log\log n \log a}{\epsilon^2} \log \frac{1}{\delta})$ Our algorithm performs better space-wise, and significantly better time-wise.

Our improved algorithm for range-efficient computation of F_0 also yields improved algorithms for all problems which can be reduced to F_0, including counting the number of triangles in the graph [5], and sensor network aggregation [22, 8].

Techniques: Our algorithm is based on random sampling. A key technical ingredient of our work 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. Our range-sampling algorithm, when combined with ideas from the random sampling algorithm due to Gibbons and Tirthapura [16], yields the algorithm for range-efficient F_0.

Range-sampling Problem: The range sampling problem that we consider is as follows. Given numbers a, b, p such that $0 < a, b < p$, we define function $f : [1, n] \to [0, p-1]$ 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 $\{x \in [x_1, x_2] | f(x) \in [0, q]\}$.

A naive solution to the range sampling problem would be to consider each element $x \in [x_1, x_2]$ separately, and see if $f(x) \in [0, q]$. The problem with this approach is that it would take $O(|x_2 - x_1|)$ time, which could be as large as $O(n)$, since the length of an interval could be as large as n.

We present a much more efficient solution to the range-sampling problem, whose time complexity is only $O(\log |x_2 - x_1|)$. Our solution is a recursive procedure which works by reducing the above problem to another range-sampling problem, but over a much 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 use-

ful in the design of other range-efficient algorithms for the data stream model.

Related Work: Estimating the F_0 of a data stream is a very well studied problem, because of its importance in various database 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 [14] 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 [2] 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 influential algorithms for estimating other frequency moments $F_2, F_3, ..$ of a data set.

Gibbons and Tirthapura [16], and Bar-Yossef et. al. [5] gave algorithms for estimating F_0 to within arbitrary relative error. The algorithm due to Gibbons and Tirthapura used random sampling, and its space complexity was $O(\log n \log \frac{1}{\delta} \frac{1}{\epsilon^2})$, and processing time per element $O(\log \frac{1}{\delta})$. The space complexity of this algorithm was later improved by Bar-Yossef et. al. [4] who gave an algorithm with better space bounds, which are the sum of $O(\log n)$ and $poly(1/\epsilon)$ terms rather than their product, but at the cost of increased processing time per element.

Prior to our work, the most efficient algorithm for range-efficient F_0 computation is due to Bar-Yossef, Kumar and Sivakumar [5], and takes $O(\frac{1}{\epsilon^5} \log \frac{1}{\delta} \log^5 n)$ processing time per interval and $O(\frac{1}{\epsilon^3} \log \frac{1}{\delta} \log n)$ space.

Considine et.al. [8] present algorithms to compute *distinct summations*, which is closely related, and can be reduced to range-efficient F_0 computation, as described earlier. Their algorithm is based on the algorithm of Flajolet and Martin [14], which assumes the presence of an ideal hash function, which can produce completely independent random numbers. Our algorithm uses hash functions which can be stored using little space, but which yield random numbers that are only pairwise independent. The algorithm in [5] does not make this assumption of ideal hash functions either, and hence works in the same model as ours. We note that in the algorithm of Considine et. al [8], it is possible to replace these ideal hash functions with pairwise-independent hash functions, just like in [2], but then we would again need a range-sampling subroutine similar to the one we present. In addition, our algorithm also provides a better running time for the *distinct summations* [8] problem.

The work of Feigenbaum et.al. [13] in 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 ([5] mention that their notion of reductions and list-efficiency were inspired by the above work of Feigenbaum et.al.). The algorithm for L^1-difference is not based on random sampling, and relies on 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.

There is much other interesting work on data stream algorithms, and we refer the reader to the surveys [21, 3].

Organization of the 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 gives the algorithm for computing F_0 using the range-sampling algorithm as the subroutine, and its correctness proof and analysis of complexity. Section 5 gives extensions of the basic algorithm to distributed streams and the computation of dominance-norms.

2. A High Level Overview

2.1. A Random Sampling Algorithm for F_0

At a high-level, our algorithm for computing F_0 follows a similar structure to the algorithm by Gibbons and Tirthapura [16]. We first recall the main idea of their 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 size might get too big if F_0 was 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. Initially, $\ell = 0$, and ℓ never decreases. The sample at level ℓ, $S(\ell)$ is determined by a hash function $S(\cdot, \ell)$, for $\ell = 0, 1, \ldots$.

When item i is presented to the algorithm, if $S(i, \ell) = 1$, then the item is stored in the sample S_ℓ, and it is not stored otherwise. Finally, the estimate of the number of distinct elements is $\frac{|S_\ell|}{p_\ell}$ where ℓ is the current sampling level. We call the algorithm described so far as the *single-item* algorithm.

Range-Sampling: 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 whether any point

in this interval belongs in the sample or not. More precisely, if ℓ is the current sampling level, and an interval $r_i = [x_i, y_i]$ arrives, we want 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 $O(y_i - x_i)$, which could be as much as $\Theta(n)$. Our main technical contribution is to reduce the time per interval significantly, to $O(\log(y_i - x_i))$ operations.

Our range-efficient algorithm simulates the single-item algorithm as follows. When an interval $r_i = [x_i, y_i]$ arrives, we compute the size of the set $\{x \in r_i | S(x, \ell) = 1\}$. If the size is greater than zero, then we add r_i to our sample, otherwise we discard r_i. If the number of intervals in our sample becomes too large, then we decrease the sampling probability by increasing the sampling level from ℓ to $\ell + 1$. We now discard all intervals r in the sample for which the set $\{x \in r | S(x, \ell + 1) = 1\}$ is empty.

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. We return $|\{x \in r_i, r_i \in S | S(x, \ell) = 1\}|/p_\ell$

Hash Functions: Since the choice of the hash function is crucial for the range-sampling algorithm, we first precisely define the hash functions we consider.

We use a standard 2-universal family of hash functions [7]. 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 $h : \{1 \ldots n\} \rightarrow \{0 \ldots p-1\}$ 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 $\ell = 0..\lfloor \log n \rfloor$, we define the following:

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

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

- The sampling function at level ℓ. 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. For every $x \in [1, n]$, $\Pr[S(x, \ell) = 1]$ is the same value, which we denote by p_ℓ. Since $h(x)$ is uniformly distributed in $\{0, \ldots p-1\}$, we have $p_\ell = |R_\ell|/p$.

The algorithms for computing F_0 due to [2, 16, 4] use hash functions defined over $GF(2^m)$, which are different from the ones that we use. In fact, for their algorithms, any pairwise independent hash function will suffice. Bar-Yossef et. al. [5], in their range-efficient F_0 algorithm, use the Toeplitz family of hash functions [17], and their algorithm uses specific properties of those hash functions.

In our case, we use the structure present in the above described hash function in our range-sampling algorithm. We do not know whether we could achieve efficient range-sampling using a different family of hash functions, say the functions defined over $GF(2^m)$.

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 $h(x) \in R_\ell$. This algorithm is later used as a subroutine by the range-efficient algorithm for F_0, which is presented in the next section.

Since $h(x) = (ax+b) \mod p$, $RangeSample([x_i, y_i], \ell)$ is the number of points $x \in [x_i, y_i]$, such that $(ax + b) \mod p \in [0, \lfloor \frac{p}{2^\ell} \rfloor - 1]$. Note that $h(x_i), h(x_i+1), \ldots, h(y_i)$ is an arithmetic progression over $\mathbb{Z}_p$ ($\mathbb{Z}_k$ is the ring of non-negative 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 $S = \langle x_1, x_2, \cdots x_l \rangle$ 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 the number of elements of S that lie in R.*

Before describing the algorithm for Problem 1, we informally describe the idea. For this informal description, we just consider the case $R = [0, L-1]$.

Divide S into subsequences $S_0, S_1, \cdots S_k, S_{k+1}$ as follows: $S_0 = x_1, x_2, \cdots, x_i$, where i is the smallest number such that $x_i > x_1$ and $x_i > x_{i+1}$. If $S_j = \langle x_i, x_{i+1}, \cdots x_m \rangle$, then $S_{j+1} = \langle x_{m+1}, x_{m+2} \cdots x_r \rangle$, where r is the smallest number such that $r > m+1$ and $x_r > x_{r+1}$, and if no such r exists then $x_r = x_l$. Note that if $S_j = \langle x_i, x_{i+1}, \cdots, x_m \rangle$, then $x_i < d$, $x_i < x_{i+1} < \cdots < x_m$. We denote the first element of S_j with f_j and the last element with e_j.

Let $L = d \times q + r$ where $r < d$. Note that the values of q and r are unique. We observe that for each $i > 0$, it is easy to compute the number of points of S_i that lie in R. More precisely,

Observation 1. *For every $i > 0$, 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 crucial.

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

We show in Claim 2 that the common difference of this sequence is $-r'$, where $r' = M \mod d$. We can always view any arithmetic progression over $\mathbb{Z}_d$ with common difference $-r'$ as an arithmetic progression with common difference $d - r'$. *The second crucial observation is that either $d - r'$ or r' is less than $d/2$, and we can choose to work with the smaller of the two.* Thus, we have reduced the original problem to a smaller problem, whose common difference is at most half of the common difference of the original problem.

We now give a formal description. We start with the following useful claims.

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

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

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 view f_is as elements over $\mathbb{Z}_d$, and below we show that the f_is form an arithmetic progression over $\mathbb{Z}_d$.

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

Proof. Recall that $S_i = \langle f_i, f_i + d, \cdots e_i \rangle$, and $M - d \leq e_i \leq M - 1$. 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 $f_i \geq r'$, 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 $f_{i+1} = (f_i - r') \mod d$. □

We would like to say that e_i's also form an arithmetic progression over $\mathbb{Z}_d$. However, since $M - 1 \geq e_i \geq M - d$, we cannot directly view e_i's as elements over $\mathbb{Z}_d$. So we define a funtion $map : \{M-d, M-d+1, \cdots, M-1\} \rightarrow \mathbb{Z}_d$ such that $map(e_i)$'s form an arithmetic progression over $\mathbb{Z}_d$. We define map as follows: $map(M - k) = k, 1 \leq k \leq d$.

Now we have the following claim for $map(e_i)$s, which is similar to Claim 2. We omit the proof since it is similar to the proof of Claim 2.

Claim 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 $T = \langle y_1, y_2, \cdots, y_k \rangle$ be an arithmetic progression over $\mathbb{Z}_t$ with a common difference $-s$, i.e.,

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

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 $R' = [0, l-1]$.

Claim 4. *Let $T' = \langle y'_1, y'_2, \cdots, y'_k \rangle$ be an arithmetic progression over $\mathbb{Z}_t$ defined as $y'_1 = -y_1$, and for $1 < i < k$, $y'_i = (y'_{i-1} + s) \; mod \; (t)$. 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 $-x \in R'$. □

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

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

1.
   - If $n = 0$, then **Return** 1 if $u \in R$, else **Return** 0.
   - Let $S = S_0 S_1 \cdots S_k S_{k+1}$. Let $Hits_0 = |S_0 \cap R|$ and $Hits_{k+1} = |S_{k+1} \cap R|$.
   - Compute k, $Hits_0$ and $Hits_{k+1}$.
   - If $d = 1$, then **Return** $Hits_0 + Hits_{k+1} + (L \times k)$.
2. Let $L = d \times q + r$, where $r < d$, and $M = d \times q' + r'$, where $r' < d$. Compute r and r'.
3. If $R = [0, L-1]$, then

   3.1. Compute f_1, where f_1 is the first element of S_1. Set $u_{new} = f_1$, $M_{new} = d$, $n_{new} = k$.

3.2. If $d - r' \leq d/2$, then $R_{new} = [0, r-1]$, and $d_{new} = d - r'$. In this case, we view f_is as arithmetic progression over $\mathbb{Z}_d$ with common difference $d - r'$.

By making a recursive call to `Hits`, we compute $High$, the cardinality of the following set

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

$High$ = `Hits`(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).

3.3. If $d - r' > d/2$ (so $r' \leq d/2$), then $u_{new} = -f_1$, $d_{new} = r'$, and $R_{new} = [-r+1, 0]$. In this case we consider the f_is as an arithmetic progression over $\mathbb{Z}_d$ with common difference $-r'$.

$High$ = `Hits`(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).

3.4. Let $Low = k - High$. **Return**

$$Hits_0 + Hits_{k+1} + (High \cdot (\lfloor \frac{L}{d} \rfloor + 1)) + (Low \cdot \lfloor \frac{L}{d} \rfloor)$$

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. Let $u_{new} = map(e_1), M_{new} = d, n_{new} = k$.

4.2. If $r' \leq d/2$, then let $R_{new} = [0, r-1]$, and $d_{new} = r'$. In this case we view $map(e_i)$s as an arithmetic progression over $\mathbb{Z}_d$ with common difference r'.

$High$ = `Hits`(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).

4.3. If $r' > d/2$ (so $d - r' \leq d/2$), then let $u_{new} = -map(e_1), d_{new} = d - r'$, and $R_{new} = [-r+1, 0]$. In this case we view $map(e_i)$s as an arithmetic progression over $\mathbb{Z}_d$ with common difference $-r'$.

$High$ = `Hits`(M_{new}, d_{new}, u_{new}, n_{new}, R_{new}).

4.4. Let $Low = k - High$. **Return**

$$(Hits_0 + Hits_{k+1} + (High \cdot (\lfloor \frac{L}{d} \rfloor + 1)) + (Low \cdot \lfloor \frac{L}{d} \rfloor)$$

Time Complexity:

First, a note on our model. We assume that arithmetic operations, including additions, multiplications and divisions in $\mathbb{Z}_k$ take unit time.

It is clear that Steps 1, and 2 can be computed using a constant number of operations. The algorithm makes a recursive call in Step 3 or Step 4. It is clear that $d_{new} \leq step/2$. 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: $M_{new} = d$, $d_{new} \leq d/2$, $n_{new} \leq \lceil n * d/M \rceil$.

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 procedure 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 $O(\log n \times (\log n + \log M))$. We can further reduce the space by a careful implementation of the recursive procedure as follows.

In general `hits`$(p_1, p_2, \ldots p_5)$ = β + γ `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 $O(\log n + \log M)$.

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

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

Step 1 correctly computes $|S_0 \cap R|$ and $|S_{k+1} \cap R|$. By Claim 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 $T = \langle f_1, \cdots, f_k \rangle$. At this point, the algorithm considers two cases.

If $d - r' \leq d/2$, then the goal is to compute the number of elements in T that lie in $[0, r-1]$. By Claim 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 Claim 2, T is an arithmetic progression over $\mathbb{Z}_d$ with common difference $-r'$. The goal is to compute

$$|T \cap [0, r-1]|.$$

Define a new sequence $T' = \langle f'_1, f'_2, \cdots, f'_k \rangle$ 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 Claim 4,

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

Thus the algorithm makes 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.

4. The range-efficient F_0 algorithm

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.

Recall that 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 $x_i, x_i+1, \ldots y_i$. Let p be a prime number between $10n$ and $20n$. From Section 2, recall the following notation.

For each level $\ell = 0..\lfloor \log n \rfloor$,
(1)$R_\ell = \{0 \ldots \lfloor \frac{p}{2^\ell} \rfloor - 1\}$
(2)The sampling function at level ℓ. 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.
(3)The sampling probability at level ℓ. $p_\ell = |R_\ell|/p$.

4.1. Algorithm Description

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 of $O(\log \frac{1}{\delta})$ such estimators, we get an (ϵ, δ)-estimator. A formal description of the algorithm appears in Figure 1.

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 $\alpha = \frac{c}{\epsilon^2}$ intervals. The algorithm 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.

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 $\ell > \lfloor \log p \rfloor$ 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 $\sum_{r \in S} RangeSample(r, \ell)/p_\ell$

Figure 1. The Range-Efficient F_0 Algorithm using the Range Sampling algorithm as a subroutine

---

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 $\ell = \lfloor \log p \rfloor$).

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. The Correctness of the Algorithm

The proof follows a parallel structure to the proof in [16]. We give the correctness proof because the hash functions used here are different from those in [16], and also for the sake of completeness.

Fact 1. For any $\ell \in [0 \ldots \lfloor \log n \rfloor]$, $1/2^{\ell+1} \leq p_\ell \leq 1/2^\ell$.

Fact 2. For any $\ell \in [0 \ldots \lfloor \log n \rfloor - 1]$, $19/40 \leq p_{\ell+1}/p_\ell \leq 1/2$.

Proof. $p_{\ell+1}/p_\ell = |R_{\ell+1}|/|R_\ell| = \lfloor \frac{p}{2^{\ell+1}} \rfloor / \lfloor \frac{p}{2^\ell} \rfloor$. Say $y = \frac{p}{2^\ell}$, and $z = \lfloor y \rfloor$. The above expression is $\lfloor y/2 \rfloor / \lfloor y \rfloor = \lfloor z/2 \rfloor / z \leq 1/2$. Since $p > 10n$, we have $2^{\ell+1} < p/10$. Hence, $z > 20$, leading to $\lfloor z/2 \rfloor / z \geq \frac{z-1}{2z} = 19/40$. □

Fact 3. For any $\ell \in [0 \ldots \lfloor \log n \rfloor - 1]$, the random variables $\{S(x, \ell) | x \in [1, n]\}$ are all pairwise independent.

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

Proof. Proof by induction. 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 verfied 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 only useful for us 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 $[x_i, y_i]$, the integer stream 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 ℓ such that $S(x, \ell) = 1$.

For level ℓ, define X_ℓ to be the number of distinct elements placed in level ℓ. Define ℓ^* to be the lowest numbered level ℓ such that $X_\ell \leq \alpha$.

Lemma 2. Let ℓ' denote the level at which the algorithm ends finally. The algorithm returns $Z = X_{\ell'}/p_{\ell'}$ and $\ell' \leq \ell^*$.

Proof. We first show that $\ell' < \ell^*$. If the algorithm never increased its sampling level, then $\ell' = 0$, and thus $\ell' \leq \ell^*$ trivially. If the algorithm did increase its sampling level, it must have been due to an overflow, and the number of intervals in the sample 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 $X_{\ell'-1} > \alpha$. Thus, $\ell' \leq \ell^*$.

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 (at least one) 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 $\ell = 0 \ldots \lfloor \log p \rfloor$, let B_ℓ denote the event that level ℓ s bad, and S_ℓ denote the event that the algorithm stops in level ℓ.

Theorem 1.

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

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 $d = \lfloor \log p \rfloor$ is reached.
- The level at which the algorithm stops is a bad level i.e. for some $\ell \in \{0 \ldots \lfloor \log p \rfloor - 1\}$ S_ℓ and B_ℓ are both true.

$$P = \sum_{\ell=0}^{d-1} \Pr[S_\ell \wedge B_\ell] + \Pr[S_d]$$

Let ℓ_m denote the lowest numbered level ℓ such that $E[X_\ell] < \alpha/C$ for some constant C to be determined by the analysis. We note than $\ell_m < d$ since $E[X_d] \leq 1 < \alpha/C$. We write P as:

$$P \leq \sum_{\ell=0}^{\ell_m} \Pr[B_\ell] + \sum_{\ell=\ell_m+1}^{d} \Pr[S_\ell]$$

In Lemma 4 we show that $\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] < 16/60$, and in Lemma 5 we show that $\sum_{\ell=\ell_m+1}^{d} \Pr[S_\ell] < 2/60$. Putting them together, the theorem is proved. □

Lemma 3. $E[X_\ell] = F_0 p_\ell$ 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 3), the variance of their sum is the sum of their variances, and the expression for the variance follows. □

Lemma 4. $\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] < 16/60$

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

$$\Pr[B_\ell] \leq \frac{\sigma_\ell^2}{\epsilon^2 \mu_\ell^2} \tag{1}$$

Substituting values from Lemma 3 into Equation 1, 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 $E[X_\ell] < \alpha/C$. Thus, $E[X_{\ell_m - 1}] = F_0 p_{\ell_m - 1} \geq \alpha/C$. Using Fact 1, we get $F_0 / 2^{\ell_m - 1} \geq \alpha/C$.

$$\sum_{\ell=0}^{\ell_m} \Pr[B_\ell] \leq \frac{4}{\epsilon^2} \frac{2^{\ell_m}}{F_0} \leq \frac{4}{\epsilon^2} \frac{2C}{\alpha} = \frac{8C}{c} < \frac{16}{60}$$

by using $c = 60, C = 2$.

□

Lemma 5. $\sum_{\ell=\ell_m+1}^{d} \Pr[S_\ell] < 1/6$

Proof. Let $P_s = \sum_{\ell=\ell_m+1}^{d} \Pr[S_\ell]$. We see that P_s is the probability that the algorithm stops in level ℓ_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 $X_{\ell_m} \geq \alpha$.

Thus,

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

From Lemma 3, we get $\sigma_{\ell_m}^2 < E[X_{\ell_m}] < \alpha/C$. Using this in the above expression, we get

$$P_s \leq \frac{\alpha}{C\alpha^2(1-1/C)^2} \leq \frac{\epsilon^2}{Cc(1-1/C)^2} < 2/60$$

The last inequality is got by substituting $C = 2, c = 8$ and $\epsilon < 1$. □

4.3. Time and Space Complexity

Space Complexity: The workspace required by the 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 $O(\frac{\log 1/\delta \log n}{\epsilon^2})$, which is the same as for the single-item algorithm.

Time Complexity: As noted in Section 3, we assume that arithmetic operations, including additions, multiplications and divisions in $\mathbb{Z}_k$ take unit time.

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

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

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

First Part: The time for checking 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:

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

2) 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 deleted at most once, the time taken 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 $O(\log(1/\epsilon))$.

3) Interval r intersects some intervals in S, but none of them contain x. This is similar to the previous case.

Third Part: 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 $\lfloor \log p \rfloor$ with high probability, since the algorithm does not reach level $\lfloor \log p \rfloor$ 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.

Time for Answering a Query: At first glance, this needs 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 $\sum_{r \in S} RangeSample(r, \ell)/p_\ell$, 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 $O(\log 1/\delta)$.

5. Applications and Extensions

5.1. Dominance Norms

We recall the problem here. Given input $\mathcal{I}$ consisting of k streams of m 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 max-dominance norm of $\mathcal{I}$ to range-efficient F_0 of a stream $\mathcal{O}$ derived as follows.

For each element $a_{i,j} \in \mathcal{I}$, we generate interval $[(j-1)m, (j-1)m + a_{i,j} - 1]$ in $\mathcal{O}$. It is easy to verify the following fact.

Fact 4. 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 our 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* [16], 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 1-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.

Our 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, different sampling levels). The referee constructs a sample of the union of the streams by sub-sampling each stream to the lowest sampling probability across all the streams. Since by our analysis, each individual stream is at a sampling probability which will likely give good estimates, the lowest sampling probability will also give us an (ϵ, δ)-estimator for F_0. The space complexity 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 [5]: *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. We later consider another possible definition, which seems harder to efficiently solve.

For vectors of dimension d, define a j-th 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 al 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 j-th coordinate range efficiently.

Our 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.

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

Fact 5. Function g has the following properties:

- g is an injective function
- A j-th 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 $g(y_1), g(y_2), \ldots g(y_n)$ 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. Due to Fact 5 and since our range efficient F_0 algorithm can handle an arithmetic sequence of integers, rather than just intervals, it follows that our algorithm can be made range-efficient in every coordinate.

Assuming that 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 $O(d\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log n)$.

Acknowledgments: We thank an anonymous referee for pointing out the connection to Dominance Norms and for suggestions which helped improve the running time of our algorithm.

References

[1] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of Computing*, pages 370 – 379, 2002.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM Sigmod-Sigact-Sigart Symp. on Principles of Database Systems (PODS)*, pages 1–16, June 2002.

[4] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. 6th International Workshop on Randomization and Approximation Techniques (RANDOM)*, pages 1–10, Sept. 2002. Lecture Notes in Computer Science, vol. 2483, Springer.

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

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

[7] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[8] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *20th International Conference on Data Engineering (ICDE)*, 2004.

[9] D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 151–156, 2004.

[10] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *Proc. 28th International Conf. on Very Large Data Bases (VLDB)*, pages 335–345, Aug. 2002.

[11] G. Cormode and S. Muthukrishnan. Estimating dominance norms of multiple data streams. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA)*, pages 148–160, 2003.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

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

[14] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[15] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proc. 27th International Conf. on Very Large Data Bases (VLDB)*, pages 541–550, Sept. 2001.

[16] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, June 2001.

[17] O. Goldreich. A sample of samplers - a computational perspective on sampling (survey). Technical Report TR97-020, ECCC, 1997.

[18] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. 21st International Conf. on Very Large Data Bases*, pages 311–322, Sept. 1995.

[19] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, page 283, 2003.

[20] Large-scale communication networks: Topology, routing, traffic and control. `http://www.ipam.ucla.edu/programs/cntop/cntop_schedule.html`.

[21] S. Muthukrishnan. Data streams: Algorithms and applications. Technical report, Rutgers University, Piscataway, NJ, 2003.

[22] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 250–262, 2004.

[23] `http://securities.stanford.edu/litigation_activity.html`.