Distinct-Values Estimation over Data Streams

Phillip B. Gibbons

Intel Research Pittsburgh, Pittsburgh PA 15213, USA
`phillip.b.gibbons@intel.com`
`http://www.pittsburgh.intel-research.net/people/gibbons/`

Abstract. In this chapter, we consider the problem of estimating the number of *distinct values* in a data stream with repeated values. Distinct-values estimation was one of the first data stream problems studied: In the mid-1980's, Flajolet and Martin gave an effective algorithm that uses only logarithmic space. Recent work has built upon their technique, improving the accuracy guarantees on the estimation, proving lower bounds, and considering other settings such as sliding windows, distributed streams, and sensor networks.

1 Introduction

Estimating the number of distinct values in a data set is a well-studied problem with many applications [1–34]. The statistics literature refers to this as the problem of estimating the number of *species* or *classes* in a population (see [4] for a survey). The problem has been extensively studied in the database literature, for a variety of uses. For example, estimates of the number of distinct values for an attribute in a database table are used in query optimizers to select good query plans. In addition, histograms within the query optimizer often store the number of distinct values in each bucket, to improve their estimation accuracy [30, 29]. Distinct-values estimates are also useful for network resource monitoring, in order to estimate the number of distinct destination IP addresses, source-destination pairs, requested urls, etc. In network security monitoring, determining sources that send to many distinct destinations can help detect fast-spreading worms [12, 32].

Distinct-values estimation can also be used as a general tool for duplicate-insensitive counting: Each item to be counted views its unique id as its “value”, so that the number of distinct values equals the number of items to be counted. Duplicate-insensitive counting is useful in mobile computing to avoid double-counting nodes that are in motion [31]. It can also be used to compute the number of distinct neighborhoods at a given hop-count from a node [27] and the size of the transitive closure of a graph [7]. In a sensor network, duplicate-insensitive counting together with multi-path in-network aggregation enables robust and energy-efficient answers to count queries [8, 24]. Moreover, duplicate-insensitive counting is a building block for duplicate-insensitive computation of other aggregates, such as sum and average.

stream S_1: C, D, B, B, Z, B, B, R, T, S, X, R, D, U, E, B, R, T, Y, L, M, A, T, W
stream S_2: T, B, B, R, W, B, B, T, T, E, T, R, R, T, E, M, W, T, R, M, M, W, B, W

Fig. 1. Two example data streams of $N = 24$ items from a universe $\{A, B, \ldots, Z\}$ of size $n = 26$. S_1 has 15 distinct values while S_2 has 6.

More formally, consider a data set, S, of N items, where each item is from a universe of n possible values. Because multiple items may have the same value, S is a *multi-set*. The *number of distinct values* in S, called the *zeroth frequency moment* F_0, is the number of values from the universe that occur at least once in S. In the context of this chapter, we will focus on the standard data stream scenario where the items in S arrive as an ordered sequence, i.e., as a data stream, and the goal is to estimate F_0 using only one pass through the sequence and limited working space memory. Fig. 1 depicts two example streams, S_1 and S_2, with 15 and 6 distinct values, respectively.

The data structure maintained in the working space by the estimation algorithm is called a *synopsis*. We seek an estimation algorithm that outputs an estimate $\hat{F_0}$, a function of the synopsis, that is guaranteed to be close to the true F_0 for the stream. We focus on the following well-studied error metrics and approximation scheme:

– **relative error metric:** the *relative error* of an estimate $\hat{F_0}$ is $|\hat{F_0} - F_0|/F_0$.
– **ratio error metric:** the *ratio error* of an estimate $\hat{F_0}$ is $\max(F_0/\hat{F_0}, \hat{F_0}/F_0)$.
– **standard error metric:** the *standard error* of an estimator Y with standard deviation $\sigma_Y(F_0)$ is $\sigma_Y(F_0)/F_0$.
– **(ϵ, δ)-approximation scheme:** an (ϵ, δ)-*approximation scheme* for F_0 is a randomized procedure that, given any positive $\epsilon < 1$ and $\delta < 1$, outputs an estimate $\hat{F_0}$ that is within a relative error of ϵ with probability at least $1 - \delta$.

Note that the standard error σ provides a means for an (ϵ, δ) trade-off: Under distributional assumptions (e.g., Gaussian approximation), the estimate is within $\epsilon = \sigma, 2\sigma, 3\sigma$ relative error with probability $1 - \delta = 65\%, 95\%, 99\%$, respectively. In contrast, the (ϵ, δ)-approximation schemes in this chapter do not rely on any distributional assumptions.

In this chapter, we survey the literature on distinct-values estimation. Section 2 discusses previous approaches based on sampling or using large space synopses. Section 3 presents the pioneering logarithmic space algorithm developed by Flajolet and Martin [13], as well as related extensions [1, 11]. We also discuss practical issues in using these algorithms in practice. Section 4 presents an algorithm that provides arbitrary precision ϵ [17], and a variant that improves on the space bound [2]. Section 5 gives lower bounds on the space needed to estimate F_0 for a data stream. Finally, Section 6 considers distinct-values estimation in a variety of important scenarios beyond the basic data stream set-up, including scenarios with selection predicates, deletions, sliding windows,

Algorithm	**Comment/Features**
Linear Counting [33]	linear space, very low standard error
FM [13], PCSA [13]	log space, good in practice, standard error
AMS [1]	realistic hash functions, constant ratio error
LogLog [11], Super-LogLog [11]	reduces PCSA synopsis space, standard error
Coordinated Sampling [17]	(ϵ, δ)-approximation scheme
BJKST [2]	improved space bound, (ϵ, δ)-approximation scheme

Table 1. Summary of the main algorithms presented in this chapter for distinct-values estimation over a data stream of values

distributed streams, and sensor networks. Table 1 summarizes the main algorithms presented in this chapter.

2 Preliminary Approaches and Difficulties

In this section, we consider several previously studied approaches to distinct-values estimation and the difficulties with these approaches. We begin with previous algorithms based on random sampling.

2.1 Sampling-Based Algorithms

A common approach for distinct-values estimation from the statistics literature (as well as much of the early work in the database literature until the mid-1990s) is to collect a sample of the data and then apply sophisticated estimators on the distribution of the values in the sample [4, 21, 22, 25, 26, 19, 5, 20]. This extended research focus on sampling-based estimators is due in part to three factors. First, in the applied statistics literature, the option of collecting data on more than a small sample of the population is generally not considered because of its prohibitive expense. For example, collecting data on every person in the world or on every animal of a certain species is not feasible. Second, in the database literature, where scanning an entire large data set is feasible, using samples to collect (approximate) statistics has proven to be a fast and effective approach for a variety of statistics [6]. Third, and most fundamental to the data streams context, the failings of existing sampling-based algorithms to accurately estimate F_0 (all known sampling-based estimators provide unsatisfactory results on some data sets of interest [5]) have spurred an ongoing focus on devising more accurate algorithms.

F_0 is a particularly difficult statistic to estimate from a sample. To gain intuition as to why this is the case, consider the following 33% sample of a data stream of 24 items:

$$B, B, T, R, E, T, M, W$$

Given this 33% sample (with its 6 distinct values), does the entire stream have 6 distinct values, 18 distinct values (i.e., the 33% sample has 33% of the distinct values), or something in between? Note that this particular sample can be

```
for $i := 0, \ldots, s-1$ do $M[i] := 0$
foreach (stream item with value $v$) do
    $M[h(v)] := 1$
let $z := |\{i : M[i] = 0\}|$
return $s \ln \frac{s}{z}$
```

Fig. 2. The Linear Counting algorithm [33]

obtained by taking every third item of either S_1 (where $F_0 = 15$) or S_2 (where $F_0 = 6$) from Fig. 1. Thus, despite sampling a large (33%) percentage of the data, estimating F_0 remains challenging, because the sample can be viewed as fairly representative of either S_1 or S_2—two streams with very different F_0's. In fact, in the worse case, all sampling-based F_0 estimators are provably inaccurate: Charikar et al. [5] proved that estimating F_0 to within a small constant factor (with probability $> \frac{1}{2}$) requires (in the worst case) that *nearly the entire data set be sampled* (see Section 5).

Thus, any approach based on a uniform sample of say 1% of the data (or otherwise reading just 1% of the data) is unable to provide good guaranteed error estimates in either the worst case or in practice. Highly-accurate answers are possible only if (nearly) the entire data set is read. This motivates the need for an effective streaming algorithm.

2.2 Streaming Approaches

Clearly, F_0 can be computed *exactly* in one pass through the entire data set, by keeping track of *all* the unique values observed in the stream. If the universe is $[1..n]$, a bit vector of size n can be used as the synopsis, initialized to 0, where bit i is set to 1 upon observing an item with value i. However, in many cases the universe is quite large (e.g., the universe of IP addresses is $n = 2^{32}$), making the synopsis much larger than the stream size N! Alternatively, we can maintain a set of all the unique values seen, which takes $F_0 \log_2 n$ bits because each value is $\log_2 n$ bits. However, F_0 can be as large as N, so again the synopsis is quite large. A common approach for reducing the synopsis by (roughly) a factor of $\log_2 n$ is to use a hash function $h()$ mapping values into a $\Theta(n)$ size range, and then adapting the aforementioned bit vector approach. Namely, bit $h(i)$ is set upon observing an item with value i. Whang et al. [33], for example, proposed an algorithm, called *linear counting*, depicted in Fig. 2. The hash function $h()$ in the figure maps each value v uniformly at random to a number in $[0, s-1]$. They show that using a load factor $F_0/s = 12$ provides estimates to within a 1% standard error. There are two limitations of this algorithm. First, we need to have a good *a priori* knowledge of F_0 in order to set the hash table size s. Second, the space is proportionate to F_0, which can be as large as n. Note that the approximation error arises from collisions in the hash table, i.e., distinct values in the stream mapping to the *same* bit position. To help alleviate this

```
for $i := 0, \dots, L-1$ do $M[i] := 0$
foreach (stream item with value $v$) do {
    $b :=$ the largest $i \geq 0$ such that the $i$ rightmost bits in $h(v)$ are all 0
    $M[b] := 1$
}
let $Z := \min\{i \ : \ M[i] = 0\}$   // i.e., the least significant 0-bit in $M$
return $\left\lfloor \frac{2^Z}{.77351} \right\rfloor$
```

Fig. 3. The FM algorithm [13], using only a single hash function and bit vector

source of error, Bloom filters (with their multiple hash functions) have been used effectively [23], although the space remains $\Theta(n)$ in the worst case.

The challenge in estimating F_0 using $o(n)$ space is that because there is insufficient space to keep track of *all* the unique values seen, it is impossible to determine whether or not an arriving stream value increases the number of distinct values seen thus far.

3 Flajolet and Martin's Algorithm

In this section, we present Flajolet and Martin's (FM) pioneering algorithm for distinct-values estimation. We also describe a related algorithm by Alon, Matias, and Szegedy. Finally, we discuss practical issues and optimizations for these algorithms.

3.1 The Basic FM Algorithm

Over two decades ago, Flajolet and Martin [13] presented the first small space distinct-values estimation algorithm. Their algorithm, which we will refer to as the *FM algorithm*, is depicted in Fig. 3. In this algorithm, the $L = \Theta(\log(\min(N, n)))$ space synopsis consists of a bit vector M initialized to all 0, where N is the number of items and n is the size of the universe. The main idea of the algorithm is to let each item in the data set select at random a bit in M and set it to 1, with (quasi-)geometric distribution; i.e., $M[i]$ is selected with probability ($\approx$) $2^{-(i+1)}$. This selection is done using a hash function h that maps each value v uniformly at random to an integer in $[0, 2^L - 1]$, and then determining the largest b such that the b rightmost bits in $h(v)$ are all 0. In this way, each distinct value gets mapped to $b = i$ with probability $2^{-(i+1)}$. For the vector length L, it suffices to take any number $> \log_2(F_0) + 4$ [13]. As F_0 is unknown, we can conservatively use $L = \log_2(\min(N, n)) + 5$.

The intuition behind the FM algorithm is as follows. Using a hash function ensures that all items with the same value will make the same selection; thus the final bit vector M is independent of any duplications among the item values. For each distinct value, the bit b is selected with probability $2^{-(b+1)}$. Accordingly, we

Item	Hash Function 1		Hash Function 2		Hash Function 3	
value v	b	$M[\cdot]$	b	$M[\cdot]$	b	$M[\cdot]$
15	1	00000010	1	00000010	0	00000001
36	0	00000011	1	00000010	0	00000001
4	0	00000011	0	00000011	0	00000001
29	0	00000011	2	00000111	1	00000011
9	3	00001011	0	00000111	0	00000011
36	0	00001011	1	00000111	0	00000011
14	1	00001011	0	00000111	1	00000011
4	0	00001011	0	00000111	0	00000011
	$Z = 2$		$Z = 3$		$Z = 2$	

$$\text{Estimate } \hat{F}_0 = \left\lfloor \frac{2^{(2+3+2)/3}}{.77351} \right\rfloor = 6$$

Fig. 4. Example run of the FM algorithm on a stream of 8 items, using three hash functions. Each $M[\cdot]$ is depicted as an $L = 8$ bit binary number, with $M[0]$ being the rightmost bit shown. The estimate, 6, matches the number of distinct values in the stream.

expect $M[b]$ to be set if there are at least 2^{b+1} distinct values. Because bit $Z - 1$ is set but not bit Z, there are likely greater than 2^Z but fewer than 2^{Z+1} distinct values. Flajolet and Martin's analysis shows that $E[Z] \approx \log_2(.77351 \cdot F_0)$, so that $2^Z/.77351$ is a good choice in that range.

To reduce the variance in the estimator, Flajolet and Martin take the average over tens of applications of this procedure (with different hash functions). Specifically, they take the average, $\bar{Z}$, of the Z's for different hash functions and then compute $\left\lfloor 2^{\bar{Z}}/.77351 \right\rfloor$. An example is given in Fig. 4.

The error guarantee, space bound, and time bound are summarized in the following theorem.

Theorem 1. **[13]** *The FM algorithm with k (idealized) hash functions produces an estimator with standard error $O(1/\sqrt{k})$, using $k \cdot L$ memory bits for the bit vectors, for any $L > \log_2(\min(N, n)) + 4$. For each item, the algorithm performs $O(k)$ operations on L-bit memory words.*

The space bound does not include the space for representing the hash functions. The time bound assumes that computing h and b are constant time operations. Sections 3.3 and 3.4 will present optimizations that significantly reduce both the space for the bit vectors and the time per item, without increasing the standard error.

3.2 The AMS Algorithm

Flajolet and Martin [13] analyze the error guarantees of their algorithm assuming the use of an explicit family of hash functions with ideal random properties (namely, that h maps each value v uniformly at random to an integer in the

```
Consider a universe $U = \{1, 2, \ldots, n\}$. Let $d$ be the smallest integer so that $2^d > n$
Consider the members of $U$ as elements of the finite field $F = GF(2^d)$,
    which are represented by binary vectors of length $d$
Let $a$ and $b$ be two random members of $F$, chosen uniformly and independently
Define $h(v) := a \cdot v + b$, where the product and addition are computed in the field $F$
$R := 0$
foreach (stream item with value $v$) do {
    $b :=$ the largest $i \geq 0$ such that the $i$ rightmost bits in $h(v)$ are all 0
    $R := \max(R, b)$
}
return $2^R$
```

Fig. 5. The AMS algorithm [1], using only a single hash function

given range). Alon, Matias, and Szegedy [1] adapted the FM algorithm to use (more realistic) linear hash functions. Their algorithm, which we will call the *AMS algorithm*, produces an estimate with provable guarantees on the ratio error. We discuss the AMS algorithm in this section.

First, note that in general, the final bit vector M returned by the FM algorithm in Section 3.1 consists of three parts, where the first part is all 0's, the third part is all 1's, and the second part (called the *fringe* in [13]) is a mix of 0's and 1's that starts with the most significant 1 and ends with the least significant 0. Let R (Z) be the position of the most (least) significant bit that is set to 1 (0, respectively). For example, for $M[\cdot] = 000101111$, $R = 5$ and $Z = 4$. Whereas the FM algorithm uses Z in its estimate, the AMS algorithm uses R. Namely, the estimate is 2^R. Fig. 5 presents the AMS algorithm. Note that unlike the FM algorithm, the AMS algorithm does not maintain a bit vector but instead directly keeps track of the most significant bit position set to 1.

The space bound and error guarantees for the AMS algorithm are summarized in the following theorem.

Theorem 2. **[1]** *For every $r > 2$, the ratio error of the estimate returned by the AMS algorithm is at most r with probability at least $1 - 2/r$. The algorithm uses $\Theta(\log n)$ memory bits.*

Proof. Let $b(v)$ be the value of b computed for $h(v)$. By the construction of $h()$, we have that for every fixed v, $h(v)$ is uniformly distributed over F. Thus the probability that $b(v) \geq i$ is precisely $1/2^i$. Moreover, for every fixed distinct v_1 and v_2, the probability that $b(v_1) \geq i$ and $b(v_2) \geq i$ is precisely $1/2^{2i}$.

Fix an i. For each element $v \in U$ that appears at least once in the stream, let $W_{v,i}$ be the indicator random variable whose value is 1 if and only if $b(v) \geq i$. Let $Z_i = \sum W_{v,i}$, where v ranges over all the F_0 elements v that appear in the stream. By linearity of expectation and because the expectation of each $W_{v,i}$ is $1/2^i$, the expectation $\mathrm{E}[Z_i]$ of Z_i is $F_0/2^i$. By pairwise independence, the variance of Z_i is $F_0 \frac{1}{2^i}(1 - \frac{1}{2^i}) < F_0/2^i$. Therefore, by Markov's Inequality, if $2^i > rF_0$ then $\Pr[Z_i > 0] < 1/r$, since $\mathrm{E}[Z_i] = F_0/2^i < 1/r$. Similarly, by Chebyshev's Inequality, if $r2^i < F_0$ then $\Pr[Z_i = 0] < 1/r$, since $\mathrm{Var}[Z_i] < F_0/2^i = \mathrm{E}[Z_i]$ and hence $\Pr[Z_i = 0] \leq \mathrm{Var}[Z_i]/(\mathrm{E}[Z_i]^2) < 1/\mathrm{E}[Z_i] = 2^i/F_0$. Because the algorithm outputs $\hat{F_0} = 2^R$, where R is the maximum i for which $Z_i > 0$, the two inequalities above show that the probability that the ratio between $\hat{F_0}$ and F_0 is not between $1/r$ and r is smaller than $2/r$, as needed.

As for the space bound, note that the algorithm uses $d = O(\log n)$ bits representing an irreducible polynomial needed in order to perform operations in F, $O(\log n)$ bits representing a and b, and $O(\log \log n)$ bits representing the current maximum R value. ■

The probability of a given ratio error can be reduced by using multiple hash functions, at the cost of a linear increase in space and time.

Corollary 1. *With k hash functions, the AMS algorithm uses $O(k \log n)$ memory bits and, for each item, performs $O(k)$ operations on $O(\log n)$-bit memory words.*

The space bound includes the space for representing the hash functions. The time bound assumes that computing $h(v)$ and $b(v)$ are constant time operations.

3.3 Practical Issues

Although the AMS algorithm has stronger error guarantees than the FM algorithm, the FM algorithm is somewhat more accurate in practice, for a given synopsis size [13]. In this section we argue why this is the case, and discuss other important practical issues.

First, the most significant 1-bit (as is used in the AMS algorithm) can be set by a single outlier hashed value. For example, suppose a hashed value sets the i'th bit whereas all other hashed values set bits $\leq i - 3$, so that $M[\cdot]$ is of the form 0001001111. In such cases, it is clear from M that 2^i is a significant overestimate of F_0: it is not supported by the other bits (in particular, both bits $i - 1$ and $i - 2$ are 0's not 1's). On the other hand, the least significant 0-bit (as is used in the FM algorithm) is supported by all the bits to its right, which must be all 1's. Thus FM is more robust against single outliers.

Second, although the AMS algorithm requires only $\approx \log \log n$ bits to represent the current R, while the FM algorithm needs $\approx \log n$ bits to represent the current $M[\cdot]$, this space savings is insignificant compared to the $O(\log n)$ bits the AMS algorithm uses for the hash function computations. In practice, the same class of hash functions (i.e., $h(v) = a \cdot v + b$) can be used in the FM algorithm, so that it too uses $O(\log n)$ bits per hash function.

Note that there are common scenarios where the space for the hash functions is not as important as the space for the accumulated synopsis (i.e., R or $M[\cdot]$). For example, in the distributed streams scenario (discussed in Section 6), an accumulated synopsis is computed for each stream locally. In order to estimate the total number of distinct values across all streams, the current accumulated synopses are collected. Thus the message size depends only on the accumulated synopsis size. Similarly, when distinct-values estimation techniques are used for

$M_j[\cdot]$'s:	00001111, 00001011, 00000111, 00010111, 00011111, 00001111
Interleaved:	000000000000000000000110110011101111111111111111
End-encoded:	11011001110, 16

Fig. 6. Example FM synopsis compression, for $k = 6$ and $L = 8$

duplicate-insensitive aggregation in sensor networks (see Section 6), the energy used in sending messages depends on the accumulated synopsis size and not the hash function size. In such scenarios, the $(\log \log n)$-bit AMS synopses are preferable to the $(\log n)$-bit FM synopses.

Note, however, that the size of the FM accumulated synopsis can be significantly reduced, using the following simple compression technique [13, 27, 8]. Recall that at any point during the processing of a stream, $M[\cdot]$ consists of three parts, where the first part is all 0's, the second part (the *fringe*) is a mix of 0's and 1's, and the third part is all 1's. Recall as well that multiple $M[\cdot]$ are constructed for a given stream, each using a distinct hash function. These $M[\cdot]$ are likely to share many bits in common, because each $M[\cdot]$ is constructed using the same algorithm on the same data. Suppose we are using k hash functions, and for $j = 1, \ldots, k$, let $M_j[\cdot]$ be the bit vector created using the jth hash function. The naive approach of concatenating $M_1[\cdot]$, $M_2[\cdot]$, ..., $M_k[\cdot]$ uses $k \cdot L$ bits (recall that each $M_j[\cdot]$ is L bits), which is $\approx k \log_2 n$ bits. Instead, we will interleave the bits, as follows:

$$M_1[L-1], M_2[L-1], \ldots, M_k[L-1], M_1[L-2], \ldots, M_k[L-2], \ldots, M_1[0], \ldots, M_k[0],$$

and then run-length encode the resulting bit vector's prefix and suffix. From the above arguments, the prefix of the interleaved bit vector is all 0's and the suffix is all 1's. We use a simple encoding that ignores the all 0's prefix, consisting of (1) the interleaved bits between the all 0's prefix and the all 1's suffix and (2) a count of the length of the all 1's suffix. An example is given in Fig. 6. Note that the interleaved fringe starts with the maximum bit set among the k fringes and ends with their minimum unset bit. Each fringe is likely to be tightly centered around the current $\log_2 F_0$, e.g., for a given hash function and $c > 0$, the probability that no bit larger than $\log_2 F_0 + c$ is set is $(1 - 2^{-(\log_2 F_0 + c)})^{F_0} \approx e^{-1/2^c}$. At any point in the processing of the stream, a similar argument shows that the interleaved fringe is expected to be fewer than $O(k \log k)$ bits. Thus our simple encoding is expected to use fewer than $\log_2 \log_2 n + O(k \log k)$ bits. In comparison, the AMS algorithm uses $k \log_2 \log_2 n$ bits for its accumulated synopsis, although this too can be reduced through compression techniques.

The number of bit vectors, k, in the FM algorithm is a tunable parameter trading off space for accuracy. The relative error as a function of k has been studied empirically in [13, 27, 8, 24] and elsewhere, with $< 15\%$ relative error reported for $k = 20$ and $< 10\%$ relative error reported for $k = 64$, on a variety

```
for $j := 1, \ldots, k$ and $i := 0, \ldots, L-1$ do $M_j[i] := 0$
foreach (stream item with value $v$) do {
    $x := h(v) \bmod k$   // Note: $k$ is a power of 2
    $b :=$ the largest $i \geq 0$ such that the $i$ rightmost bits in $\lfloor h(v)/k \rfloor$ are all 0
    $M_x[b] := 1$
}
$\bar{Z} := \frac{1}{k} \sum_{j=1}^{k} \min\{i \ : \ M_j[i] = 0\}$
return $\left\lfloor \frac{k}{.77351} 2^{\bar{Z}} \right\rfloor$
```

Fig. 7. The PCSA algorithm [13]

of data sets. These studies show a strong diminishing return for increases in k. Theorem 1 shows that the standard error is $O(1/\sqrt{k})$. Thus reducing the standard error from 10% to 1% requires increasing k by a factor of 100! In general, to obtain a standard error at most ϵ we need $k = \Theta(1/\epsilon^2)$. Estan, Varghese and Fisk [12] present a number of techniques for further improving the constants in the space vs. error trade-off, including using multi-resolution and adaptive bit vectors.

Both the FM and AMS algorithms use the largest i such that the i rightmost bits in $h(v)$ are all 0, in order to create an exponential distribution onto an integer range. A related, alternative approach by Cohen [7] is to (1) use a hash function that maps uniformly to the interval $[0, 1]$, (2) maintain the minimum hashed value x seen thus far in the stream S (i.e., $x = \min_{v \in S}\{h(v)\}$), and then (3) return $\frac{1}{x} - 1$ as the estimate for F_0. The intuition is that if there are F_0 distinct values mapped uniformly at random to $[0, 1]$, then we may expect them to divide the interval into $F_0 + 1$ relatively evenly-spaced subintervals, i.e., subintervals of size $\frac{1}{F_0+1}$. As $[0, x]$ is the first such subinterval, $x = \frac{1}{F_0+1}$, and hence $\frac{1}{x} - 1$ is used as the estimate for F_0. As with the FM and AMS algorithms, the error guarantee can be improved by taking multiple hash functions and averaging. Empirically, this approach is not as accurate as the FM algorithm for a given synopsis size [27].

3.4 Improving the Per-Item Processing Time

The FM algorithm as presented in Section 3.1 performs $O(k)$ operations on memory words of $\approx \log_2 n$ bits (recall Theorem 1) for each stream item. This is because a different hash function is used for each of the k bit vectors. To reduce the processing time per item from $O(k)$ to $O(1)$, Flajolet and Martin present the following variant on their algorithm, called *Probabilistic Counting with Stochastic Averaging (PCSA).*

In the PCSA algorithm (see Fig. 7), k bit vectors are used (for k a power of 2) but only a single hash function $h()$. For each stream item with value v, the $\log_2 k$ least significant bits of $h(v)$ are used to select a bit vector. Then the remaining $L - \log_2 k$ bits of $h(v)$ are used to select a position within that bit

```
for $j := 1, \ldots, k$ do $R_j := 0$
foreach (stream item with value $v$) do {
    $x := h(v) \bmod k$   // Note: $k$ is a power of 2
    $b :=$ the largest $i \geq 0$ such that the $i$ rightmost bits in $\lfloor h(v)/k \rfloor$ are all 0
    $R_x := \max(R_x, b)$
}
$\bar{Z} := \frac{1}{k} \sum_{j=1}^{k} R_j$
return $\lfloor (0.79402k - 0.84249) 2^{\bar{Z}} \rfloor$
```

Fig. 8. The LogLog algorithm [11]

vector, according to an exponential distribution (as in the basic FM algorithm). To compute an estimate, PCSA averages over the positions of the least significant 0-bits, computes 2 to the power of that average, and divides by the bias factor .77351. To compensate for the fact that each bit vector has seen only $1/k$'th of the distinct items on average, the estimate is multiplied by k.

The error guarantee, space bound, and time bound are summarized in the following theorem.

Theorem 3. [13] *The PCSA algorithm with k bit vectors and an (idealized) hash function produces an estimator with standard error $0.78/\sqrt{k}$, using $k \cdot L$ memory bits for the bit vectors, for any $L > \log_2(\min(N, n)/k) + 4$. For each item, the algorithm performs $O(1)$ operations on $(\log_2 n)$-bit memory words.*

The space bound does not include the space for representing the hash function. The time bound assumes that computing h and b are constant time operations. Although some of the operations use only L-bit words, the word size is dominated by the $\log_2 n$ bits for the item value v.

Durand and Flajolet [11] recently presented a variant of PCSA, called the *LogLog* algorithm, that reduces the size of the accumulating synopsis from $\log n$ to $\log \log n$. (As in FM and PCSA, the size of the hash function is not accounted for in the space bound.) The algorithm, depicted in Fig. 8, differs from PCSA by maintaining the maximum bit set (as in AMS) and using a different function for computing the estimate. The analysis in [11] gives the bias correction factor $(0.79402k - 0.84249)$, where k is the number of maximums (i.e., R_j's) maintained. With k maximums, the standard error is shown to be $1.30/\sqrt{k}$ (assuming idealized hash functions). This is higher than with the PCSA algorithm (supporting our earlier argument that the least-significant 0-bit is more accurate than the most-significant 1-bit), but the synopsis size is smaller (when ignoring the hash function space and not using the compression tricks in Section 3.3).

Durand and Flajolet [11] also present a further improvement, called *Super-LogLog*, that differs from LogLog in two aspects. First, it discards the largest 30% of the estimates, in order to decrease the variance. As discussed in Section 3.3, using the maximum bit set is subject to overestimation caused by outlier bits being set. By discarding the largest estimates, these outliers are discarded. Note that a different correction factor is needed in order to compensate for this additional source of bias [11]. Second, it represents each maximum R_j using only $L = \log_2(\log_2(n/k) + 3)$ bits, and again corrects for the additional bias. The error guarantee, space bound, and time bound are summarized in the following theorem.

Theorem 4. **[11]** *The Super-LogLog algorithm with k maximums and an (idealized) hash function produces an estimator with standard error $1.05/\sqrt{k}$, using $k \cdot L$ memory bits for the maximums, where $L = \lceil \log_2 \lceil \log_2(\min(N, n)/k) + 3 \rceil \rceil$. For each item, the algorithm performs $O(1)$ operations on $(\log_2 n)$-bit memory words.*

The space bound does not include the space for representing the hash function. The time bound assumes that computing h and b are constant time operations. Although some of the operations use only L-bit words, the word size is dominated by the $\log_2 n$ bits for the stream value v. The standard error $1.05/\sqrt{k}$ for Super-LogLog is higher than the $0.78/\sqrt{k}$ error for PCSA. Comparing the synopsis sizes (ignoring the hash functions), super-LogLog uses a fixed $\approx k \log_2 \log_2(n/k)$ bits, whereas PCSA using the compression tricks of Section 3.3 uses an expected $\approx \log_2 \log_2 n + O(k \log k)$ bits.

4 (ϵ, δ)-Approximation Schemes

None of the algorithms presented thus far provides the strong guarantees of an (ϵ, δ)-approximation scheme. In this section, we present two such algorithms: the Coordinated Sampling algorithm of Gibbons and Tirthapura [17], and an improvement by Bar-Yossef et al. [2] that achieves near optimal space.

4.1 Coordinated Sampling

Gibbons and Tirthapura [17] gave the first (ϵ, δ)-approximation scheme for F_0. Their algorithm, called *Coordinated Sampling*, is depicted in Fig. 9.

In the algorithm, there are $k = \Theta(\log(1/\delta))$ instances of the same procedure, differing only in their use of different hash functions $h_j()$. For each instance, the hash function is used to assign each potential stream value to a "level", such that half the values are assigned to level 0, a quarter to level 1, etc. The algorithm maintains a set of the $\approx \tau$ distinct stream values that have the highest levels among those observed thus far. More specifically, it keeps track of the minimum level ℓ_j such that there are at most τ distinct stream values with level at least ℓ_j, as well as the set, S_j, of these stream values.

As in the AMS algorithm (Section 3.2), any uniform pairwise independent hash function can be used for $h_j()$; for example, linear hash functions can be used. Let $b_j(v)$ be the value of b computed in the algorithm for $h_j(v)$. Following the argument in Section 3.2, we have that $\Pr\{b_j(v) = \ell\} = \frac{1}{2^{\ell+1}}$ and $\Pr\{b_j(v) \geq \ell\} = \frac{1}{2^\ell}$ for $\ell = 0, \ldots, \log n - 1$. Hence, S_j is always a uniform random sample of

```
for $j := 1, \ldots, k$ do { $\ell_j := 0$, $S_j := \emptyset$ }
foreach (stream item with value $v$) do {
    for $j := 1, \ldots, k$ do {
        $b :=$ the largest $i \geq 0$ such that the $i$ rightmost bits in $h_j(v)$ are all 0
        if $b \geq \ell_j$ and $(v, b) \notin S_j$ do {
            $S_j := S_j \cup \{(v, b)\}$
            // if $S_j$ is too large, discard the level $\ell_j$ sample points from $S_j$
            while $|S_j| > \tau$ do {
                $S_j := S_j - \{(v', b') \ : \ b' = \ell_j\}$
                $\ell_j := \ell_j + 1$
            }
        }
    }
}
return $\text{median}_{j=1,\ldots,k}(|S_j| \cdot 2^{\ell_j})$
```

Fig. 9. The Coordinated Sampling algorithm [17]. The values of k and τ depend on the desired ϵ and δ: $k = 36 \log_2(1/\delta)$ and $\tau = 36/\epsilon^2$, where the constant 36 is determined by the worst case analysis and can be much smaller in practice.

the distinct stream values observed thus far, where each value is in the sample with probability $2^{-\ell_j}$. Thus, Coordinated Sampling uses $|S_j| \cdot 2^{\ell_j}$ as the estimate for the number of distinct values in the stream. To ensure that the estimate is within ϵ with probability $1 - \delta$, it computes the median over $\Theta(\log(1/\delta))$ such estimates.

Each step of the algorithm within the “for” loop can be done in constant (expected) time by maintaining the appropriate data structures, assuming (as we have for the previous algorithms) that computing hash functions and determining b are constant time operations. For example, each S_j can be stored in a hash table T_j of 2τ entries, where the pair (v, b) is the hash key. This enables both tests for whether a given (v, b) is in S_j and insertions of a new (v, b) into S_j to be done in constant expected time. We can enable constant time tracking of the size of S_j by maintaining an array of $\log n + 1$ “level” counters, one per possible level, which keep track of the number of pairs in S_j for each level. We also maintain a running count of the size of S_j. This counter is incremented by 1 upon insertion into S_j and decremented by the corresponding level counter upon deleting all pairs in a level. In the latter case, in order to quickly delete from S_j all such pairs, we leave these deleted pairs in place, removing them lazily as they are encountered in subsequent visits to T_j. (We need not explicitly mark them as deleted because subsequent visits see that their level numbers are too small and treat them as deleted.)

The error guarantee, space bound, and time bound are summarized in the following theorem. The space bound includes the space for representing the hash functions. The time bound assumes that computing hash functions and b are constant expected time operations.

Theorem 5. **[17]** *The Coordinated Sampling algorithm provides an (ϵ, δ)-approximation scheme, using $O(\frac{\log n \log(1/\delta)}{\epsilon^2})$ memory bits. For each item, the algorithm performs an expected $O(\log(1/\delta))$ operations on $(\log_2 n)$-bit memory words.*

Proof. We have argued above about the time bound. The space bound is $O(k \cdot \tau)$ memory words, i.e., $O(\frac{\log n \log(1/\delta)}{\epsilon^2})$ memory bits.

In what follows, we sketch the proof that Coordinated Sampling is indeed an (ϵ, δ)-approximation scheme. A difficulty in the proof is that the algorithm decides when to stop changing levels based on the outcome of random trials, and hence may stop at an incorrect level, and make correspondingly bad estimates. We will argue that the probability of stopping at a "bad" level is small, and can be accounted for in the desired error bound.

Accordingly, consider the jth instance of the algorithm. For $\ell \in \{0..\log n\}$ and $v \in \{1..n\}$, we define the random variables $X_{\ell,v}$ such that $X_{\ell,v} = 1$ if v's level is at least ℓ and 0 otherwise. For the stream S, we define $X_\ell = \sum_{v \in S} X_{\ell,v}$ for every level ℓ. Note that after processing S, the value of ℓ_j is the lowest numbered level f such that $X_f \leq \tau$. The algorithm uses the estimate $2^f \cdot X_f$.

For every level $\ell \in \{0 \ldots \log n\}$, we define B_ℓ such that $B_\ell = 1$ if $2^\ell X_\ell \notin [(1-\epsilon)F_0, (1+\epsilon)F_0]$ and 0 otherwise. Level ℓ is "bad" if $B_l = 1$, and "good" otherwise. Let E_ℓ denote the event that the final value of ℓ_j is ℓ i.e., that f equals ℓ. The heart of the proof is to show the following:

$$\Pr\{\text{Given instance produces an estimate not in } [(1-\epsilon)F_0, (1+\epsilon)F_0]\} < \frac{1}{3} \quad (1)$$

Let P be the probability in equation 1. Let ℓ^* denote the first level such that $E[X_{\ell^*}] \leq \frac{2}{3}\tau$. The instance produces an estimate not within the target range if B_f is true for the level f such that E_f is true. Thus,

$$P = \sum_{i=0}^{\log n} \Pr\{E_i \wedge B_i\} < \sum_{i=0}^{\ell^*} \Pr\{B_i\} + \sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} \quad (2)$$

The idea behind using the inequality to separate the B_i terms from the E_i terms is that the lower levels (until ℓ^*) are likely to have good estimates and the algorithm is unlikely to keep going beyond level ℓ^*.

As in the proof for the AMS algorithm (Theorem 2), we have that for $\ell = 0, \ldots, \log n - 1$,

$$\mathrm{E}[X_\ell] = \frac{F_0}{2^\ell}, \quad (3)$$

and

$$\mathrm{var}[X_\ell] < \frac{F_0}{2^\ell} \quad (4)$$

We will now show that

$$\sum_{i=0}^{\ell^*} \Pr\{B_i\} < \frac{6}{\epsilon^2 \tau} \tag{5}$$

To see this, we first express $\Pr\{B_i\}$ in terms of equation 3: $\Pr\{B_i\} = \Pr\{|X_i - \frac{F_0}{2^i}| \geq \epsilon \frac{F_0}{2^i}$. Then, from equation 4 and using Chebyshev's inequality, we have $\Pr\{B_i\} < \frac{2^i}{F_0 \epsilon^2}$. Hence, $\sum_{i=0}^{\ell^*} \Pr\{B_i\} < \sum_{i=0}^{\ell^*} \frac{2^i}{F_0 \epsilon^2} = \frac{2^{\ell^*+1}}{F_0 \cdot \epsilon^2}$ Now, because ℓ^* is the first level such that $\frac{F_0}{2^{\ell^*}} \leq \frac{2}{3}\tau$, we have that $F_0 > 2^{\ell^*-1} \cdot \frac{2}{3}\tau$. Thus, $\sum_{i=0}^{\ell^*} \Pr\{B_i\} < \frac{2^{\ell^*+1}}{F_0 \cdot \epsilon^2} < \frac{6}{\epsilon^2 \tau}$, establishing equation 5.

Next, we will show that

$$\sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} < \frac{6}{\tau} \tag{6}$$

To see this, we first observe that $\sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} = \Pr\{X_{\ell^*} > \tau\}$, because the E_i's are mutually exclusive. Because $E[X_{\ell^*}] < \frac{2}{3}\tau$, we have $\Pr\{X_{\ell^*} > \tau\} < \Pr\{X_{\ell^*} - E[X_{\ell^*}] > \frac{\tau}{3}\}$. By Chebyshev's inequality and equation 4, this latter probability is less than $\frac{9}{\tau^2} \cdot \frac{F_0}{2^{\ell^*}}$. Plugging in the fact that $\frac{2}{3}\tau > E[X_{\ell^*}] = \frac{F_0}{2^{\ell^*}}$, we obtain $\sum_{i=\ell^*+1}^{\log n} \Pr\{E_i\} < \frac{9}{\tau^2} \cdot \frac{2\tau}{3}$, establishing equation 6.

Plugging into equation 2 the results from equations 5 and 6, and setting $\tau = 36/\epsilon^2$, we have $P < \frac{1}{6} + \frac{\epsilon^2}{6} < \frac{1}{3}$. Thus, equation 1 is established.

Finally, the median fails to be an (ϵ, δ) estimator of F_0 if at least $k/2$ instances of the algorithm fail. By equation 1, we expect $< k/3$ to fail, and hence by Chernoff bounds, the probability the algorithm fails is less than $\exp(-k/36)$. Setting $k = 36 \log(1/\delta)$ makes this probability less than δ, completing the proof of the theorem. ■

4.2 Improving the Space Bound

Bar-Yossef et al. [2] showed how to adapt the Coordinated Sampling algorithm in order to improve the space bound. Specifically, their algorithm, which we call the *BJKST algorithm*, stores the elements in S_j using less space, as follows. Instead of storing the pair (v, b), as in Coordinated Sampling, the BJKST algorithm stores $g(v)$, for a suitably chosen hash function $g()$. Namely, $g()$ is a (randomly chosen) uniform pairwise independent hash function that maps values from $[0..n-1]$ to the range $[0..R-1]$, where $R = 3((\log n+1)\tau)^2$. Thus only $O(\log\log n + \log(1/\epsilon))$ bits are needed to store $g(v)$. The level b for v is represented implicitly by storing the hashed values as a collection of balanced binary search trees, one tree for each level.

The key observation is that for any given instance of the algorithm, $g()$ is applied to at most $(\log n + 1) \cdot \tau$ distinct values. Thus, the choice of R ensures that with probability at least 5/6, $g()$ is injective on these values. If $g()$ is indeed injective, then using $g()$ did not alter the basic progression of the instance. The alternative occurs with probability at most 1/6. To compensate, the BJKST algorithm uses a larger τ, namely, $\tau = 576/\epsilon^2$, such that the probability of a bad estimate can be bounded by 1/6. Because $\frac{1}{6} + \frac{1}{6} = \frac{1}{3}$, a result akin to equation 1 can be established. Finally, taking the median over $k = 36\log(1/\delta)$ instances results in an (ϵ, δ)-approximation.

The error guarantee, space bound, and time bound are summarized in the following theorem. The space bound includes the space for representing the hash functions. The time bound assumes that computing hash functions and b are constant expected time operations.

Theorem 6. [2] *The BJKST algorithm provides an (ϵ, δ)-approximation scheme, using $O((\frac{1}{\epsilon^2}(\log(1/\epsilon) + \log\log n) + \log n)\log(1/\delta))$ memory bits. For each item, the algorithm performs $O(\log(1/\delta))$ operations on $(\log_2 n)$-bit words plus at most $O(\frac{\log(1/\delta)}{\epsilon^2})$ operations on $(\log_2(1/\epsilon) + \log_2\log_2 n)$-bit words.*

5 Lower Bounds

This section presents five key lower bound results for distinct-values estimation.

The first lower bound shows that *observing (nearly) the entire stream* is essential for obtaining good estimation error guarantees for all input streams.

Theorem 7. [5] *Consider any (possibly adaptive and randomized) estimator for the number of distinct values F_0 that examines at most r items in a stream of N items. Then, for any $\gamma > e^{-r}$, there exists a worst case input stream such that with probability at least γ, the ratio error of the estimate $\hat{F}_0$ output by the estimator is at least $\sqrt{\frac{N-r}{2r}\ln\frac{1}{\gamma}}$.*

Thus when $r = o(N)$, the ratio error is non-constant with high probability. Even when 1% of the input is examined, the ratio error is at least 5 with probability $> 1/2$.

The second lower bound shows that *randomization* is essential for obtaining low estimation error guarantees for all input streams, if we hope to use sublinear space. For this lower bound, we also provide the proof, as a representative example of how such lower bounds are proved.

Theorem 8. [1] *Any deterministic algorithm that outputs, given one pass through a data stream of $N = n/2$ elements of $U = \{1, 2, \ldots, n\}$, an estimate with at most 10% relative error requires $\Omega(n)$ memory bits.*

Proof. Let G be a family of $t = 2^{\Omega(n)}$ subsets of U, each of cardinality $n/4$ so that any two distinct members of G have at most $n/8$ elements in common. (The existence of such a G follows from standard results in coding theory, and can be proved by a simple counting argument). Fix a deterministic algorithm that approximates F_0. For every two members G_1 and G_2 of G let $A(G_1, G_2)$ be the stream of length $n/2$ starting with the $n/4$ members of G_1 (in a sorted order) and ending with the set of $n/4$ members of G_2 (in a sorted order). When the algorithm runs, given a stream of the form $A(G_1, G_2)$, the memory configuration after it reads the first $n/4$ elements of the stream depends only on G_1. By the pigeonhole principle, if the memory has less than $\log_2 t$ bits, then there are two distinct sets G_1 and G_2 in G, so that the content of the memory after reading the elements of G_1 is equal to that content after reading the elements of G_2. This means that the algorithm must give the same final output to the two streams $A(G_1, G_1)$ and $A(G_2, G_1)$. This, however, contradicts the assumption, because $F_0 = n/4$ for $A(G_1, G_1)$ and $F_0 \geq 3n/8$ for $A(G_2, G_1)$. Therefore, the answer of the algorithm makes a relative error that exceeds 0.1 for at least one of these two streams. It follows that the space used by the algorithm must be at least $\log_2 t = \Omega(n)$, completing the proof. ∎

The third lower bound shows that *approximation* is essential for obtaining low estimation error guarantees for all input streams, if we hope to use sublinear space.

Theorem 9. [**1**] *Any randomized algorithm that outputs, given one pass through a data stream of at most $N = 2n$ items of $U = \{1, 2, \ldots, n\}$, a number Y such that $Y = F_0$ with probability at least $1 - \delta$, for some fixed $\delta < 1/2$, requires $\Omega(n)$ memory bits.*

The fourth lower bound shows that $\Omega(\log n)$ memory bits are required for obtaining low estimation error.

Theorem 10. [**1**] *Any randomized algorithm that outputs, given one pass through a data stream of items from $U = \{1, 2, \ldots, n\}$, an estimate with at most a 10% relative error with probability at least 3/4 must use at least $\Omega(\log n)$ memory bits.*

The final lower bound shows that $\Omega(1/\epsilon^2)$ memory bits are required in order to obtain an (ϵ, δ)-approximation scheme (even for constant δ).

Theorem 11. [**34**] *For any δ independent of n and any ϵ, any randomized algorithm that outputs, given one pass through a data stream of items from $U = \{1, 2, \ldots, n\}$, an estimate with at most an ϵ relative error with probability at least $1 - \delta$ must use at least $\Omega(\min(n, 1/\epsilon^2))$ memory bits.*

Thus we have an $\Omega(1/\epsilon^2 + \log n)$ lower bound for obtaining arbitrary relative error for constant δ and, by the BJKST algorithm, a nearly matching upper bound of $O(1/\epsilon^2(\log(1/\epsilon) + \log\log n) + \log n)$.

6 Extensions

In this section, we consider distinct-values estimation in a variety of important scenarios beyond the basic data stream set-up. In Sections 6.1–6.5, we focus on sampling, sliding windows, update streams, distributed streams, and sensor networks (ODI), respectively, as summarized in Table 2. Finally, Section 6.6 highlights three additional settings studied in the literature.

Algorithm	**Cite & Section**	**Sampling Distinct**	**Sliding Windows**	**Update Streams**	**Distributed Streams**	**ODI**
FM	[13]; 3.1	no	no	no	yes	yes
PCSA	[13]; 3.4	no	no	no	yes	yes
FM with $\log^2$ space	6.2, 6.3	no	yes	yes	yes	no
AMS	[1]; 3.2	no	no	no	yes	yes
Cohen	[7]; 3.3	no	no	no	yes	yes
LogLog	[11]; 3.4	no	no	no	yes	yes
Coordinated Sampling	[17]; 4.1	yes	no	no	yes	yes
BJKST	[2]; 4.2	no	no	no	yes	yes
Distinct Sampling	[16]; 6.1	yes	no	no	yes	yes
Randomized Wave	[18]; 6.2	yes	yes	no	yes	yes
l_0 Sketch	[9]; 6.2	no	no	yes	yes	no
Ganguly	[14]; 6.3	yes	no	yes	yes	no
CLKB	[8]; 6.5	no	no	no	yes	yes

Table 2. Scenarios handled by the main algorithms discussed in this section

6.1 Sampling Distinct

In addition to providing an estimate of the number of distinct values in the stream, several algorithms provide a *uniform sample of the distinct values* in the stream. Such a sample can be used for a variety of sampling-based estimation procedures, such as estimating the mean, the variance, and the quantiles over the distinct values. Algorithms that retain only hashed values, such as FM, PCSA, AMS, Cohen, LogLog, BJKST, l_0 Sketch (Section 6.2) and CLKB (Section 6.5), do not provide such samples. In some cases, such as Cohen, the algorithm can be readily adapted to produce a uniform sample (with replacement): For each instance (i.e., each hash function) of the algorithm, maintain not just the current minimum hashed value but also the original value associated with this minimum hashed value. As long as two different values do not hash to the same minimum value for a given hash function, each parallel instance produces one sample point. In contrast, Coordinated Sampling, Randomized Wave (Section 6.2) and Ganguly's algorithm (Section 6.3) all directly provide a uniform sample of the distinct values.

Gibbons [16] extended the sampling goal to a multidimensional data setting that arises in a class of common databases queries. Here, the goal is to extract a uniform sample of the distinct values in a primary dimension, as before, but instead of retaining only the randomly selected values V, the algorithm retains a "same-value" sample for each value in V. Specifically, for each $v \in V$, the algorithm maintains a uniform random sample chosen from all the stream items with value v. A user-specified parameter t determines the size of each of these same-value samples; if there are fewer than t stream items with a particular value, the algorithm retains them all. The algorithm, called *Distinct Sampling*, is similar to Coordinated Sampling (Fig. 9) in having $\log n$ levels, maintaining all values whose levels are above a current threshhold, and incrementing the level

```
select count(distinct target-attr)
from Table
where P
```

(a)

```
select count(distinct o_custkey)
from orders
where o_orderdate ≥ '2006-01-01'
```

(b)

Fig. 10. (a) Distinct Values Query template (b) Example query

threshhold whenever a space bound is reached. However, instead of retaining one (v, b) pair for the value v, it starts by retaining each of the first t items with value v in the primary dimension, as well as a count, n_v, of the number of items in the stream with value v (including the current item). Then, upon observing any subsequent items with value v, it maintains a uniform same-value sample for v by adding the new item to the sample with probability t/n_v, making room by discarding a random item among the t items currently in the sample for v. Fig. 10 gives an example of the type of SQL query that can be well estimated by the Distinct Sampling algorithm, where target-attr in Fig. 10(a) is the primary dimension and the predicate P is typically on one or more of the other dimensions, as in Fig. 10(b). The estimate is obtained by first applying the predicate to the same-value samples, in order to estimate what fraction of the values in V would be eliminated by the predicate, and then outputting the overall query estimate based on the number of remaining values.

6.2 Sliding Windows

The sliding windows setting is motivated by the desire to estimate the number of distinct values over only the most recent stream items. Specifically, we are given a window size W, and the problem is to estimate the number of distinct values over a sliding window of the W most recent items. The goal is to use space that is logarithmic in W (linear in W would be trivial). Datar et al. [10] observed that the FM algorithm can be extended to solve the sliding windows problem, by keeping track of the stream position of the most recent item that set each FM bit. Then, when estimating the number of distinct values within the current sliding window, only those FM bits whose associated positions are within the window are considered to be set. This increases the space needed for the FM algorithm by a logarithmic factor.

Gibbons and Tirthapura [18] developed an (ϵ, δ)-approximation scheme for the sliding windows scenario. Their algorithm, called *Randomized Wave*, is depicted in Fig. 11. In the algorithm, there are $k = \Theta(\log(1/\delta))$ instances of the same procedure, differing only in their use of different hash functions $h_j()$. Any uniform, pairwise independent hash function can be used for $h_j()$. Let $b_j(v)$ be the value of b computed in the algorithm for $h_j(v)$.

Whereas Coordinated Sampling maintained a single uniform sample of the distinct values, Randomized Wave maintains $\approx \log W$ uniform samples of the

```
// Note: All additions and comparisons in this algorithm are done modulo $W'$
$W'$ := the smallest power of 2 greater than or equal to $2W$
$pos := 0$
for $j := 1, \ldots, k$ do {
    initialize $V_j$ to be an empty list   // value list
    for $\ell := 0, \ldots, \log(W')$ do
        initialize $L_j(\ell)$ to be an empty list   // level lists
}
// Process the stream items
foreach (stream item with value $v$) do {
    $pos := pos + 1$
    for $j := 1, \ldots, k$ do {
        if the tail $(v', p')$ of $V_j$ has expired (i.e., $p' = pos - W$)
            discard $(v', p')$ from $V_j$ and from any level list $L_j()$
        $b$ := the largest $i \geq 0$ such that the $i$ rightmost bits in $h_j(v)$ are all 0
        for $\ell := 0, \ldots, b$ do {
            if $v$ is already in $L_j(\ell)$
                remove current entry for $v$ and insert $(v, pos)$ at the head of $L_j(\ell)$
            else do {
                if $|L_j(\ell)| = \tau$ then discard the pair at the tail of $L_j(\ell)$
                insert $(v, pos)$ at the head of $L_j(\ell)$
            }
        }
        if $v$ is in $V_j$
            remove current entry for $v$ from $V_j$
        insert $(v, pos)$ at the head of $V_j$
    }
}
// Compute an estimate for a sliding window of size $w \leq W$
$s := \max(0, pos - w + 1)$   // $[s, pos]$ is the desired window
for $j := 1, \ldots, k$ do {
    $\ell_j$ := min level such that the tail of $L_j(\ell_j)$ contains a position $p \leq s$
    $c_j$ := number of pairs in $L_j(\ell_j)$ with $p \geq s$
}
return $\text{median}_{j=1,\ldots,k}(c_j \cdot 2^{\ell_j})$
```

Fig. 11. The Randomized Wave algorithm [18]. The values of k and τ depend on the desired ϵ and δ: $k = 36 \log_2(1/\delta)$ and $\tau = 36/\epsilon^2$, where the constant 36 is determined by the worst case analysis and can be much smaller in practice.

distinct values. Each of these "level" samples corresponds to a different sampling probability, and retains only the $\tau = \Theta(1/\epsilon^2)$ most recent distinct values sampled into the associated level. (In the figure, $L_j(\ell)$ is the level sample for level ℓ of instance j.) An item with value v is selected into levels $0, \ldots, b_j(v)$, and stored as the pair (v, pos), where *pos* is the stream position when v most recently occurred.

Each level sample $L_j(\ell)$ can be maintained as a doubly linked list. The algorithm also maintains a (doubly linked) list V_j of all the values in any of the level samples $L_j()$, together with the position of their most recent occurrences, ordered by position. This list enables fast discarding of items no longer within the sliding window. Finally, there is a hash table H_j (not shown in the figure) that holds triples $(v, Vptr, Lptr)$, where v is a value in V_j, *Vptr* is a pointer to the entry for v in V_j, and *Lptr* is a doubly linked list of pointers to each of the occurrences of v in the level samples $L_j()$. These triples are stored in H_j hashed by their value v.

Consider an instance j. For each stream item, we first check the oldest value v' in V_j to see if its position is now outside of the window, and if so, we discard it. We use the triple in $H_j(v')$ to locate all occurrences of v' in the data structures; these occurrences are spliced out of their respective doubly linked lists. Second, we update the level samples L_j for each of the levels $0 \ldots b_j(v)$, where v is the value of the stream item. There are two cases. If v is not in the level sample, we insert it, along with its position *pos*, at the head of the level sample. Otherwise, we perform a move-to-front: splicing out v's current entry and inserting (v, pos) at the head of the level sample. In the former case, if inserting the new element would make the level sample exceed τ elements, we discard the oldest element to make room. Finally, we insert (v, pos) at the head of V_j, and if v was already in V_j, we splice out the old entry for v. Because the expected value of $b_j(v)$ is less than 2, v occurs in an expected constant number (in this case, 2) of levels. Thus, all of the above operations can be done in constant expected time.

Let (v', p') denote the pair at the tail of a level sample $L_j(\ell)$. Then $L_j(\ell)$ contains all the distinct values with stream positions in the interval $[p', pos]$ whose $b_j()$'s are at least ℓ. Thus, similar to Coordinated Sampling, an estimate of the number of distinct values within a window can be obtained by taking the number of elements in $L_j(\ell)$ in this interval and multiplying by 2^ℓ, the inverse of the sampling probability for the level.

The error guarantee, space bound, and time bound are summarized in the following theorem. The space bound includes the space for representing the hash functions. The time bound assumes that computing hash functions and b are constant expected time operations.

Theorem 12. **[18]** *The Randomized Wave algorithm provides an* (ϵ, δ)*-approximation scheme for estimating the number of distinct values in any sliding window of size* $w \leq W$*, using* $O(\frac{\log n \log W \log(1/\delta)}{\epsilon^2})$ *memory bits, where the values are in* $[0..n)$*. For each item, the algorithm performs an expected* $O(\log(1/\delta))$ *operations on* $\max(\log n, 2 + \log W)$*-bit memory words.*

Note that by setting W to be N, the length of the stream, the algorithm provides an (ϵ, δ)-approximation scheme for all possible window sizes.

6.3 Update Streams

Another important scenario is where the stream contains both new items and the deletion of previous items. Examples include estimating the current number of distinct network connections, phone connections or IP flows, where the stream contains both the start and the end of each connection or flow. Most of the distinct-values algorithms discussed thus far are not designed to handle deletions. For example, algorithms that retain only the maximum of some quantity, such as AMS, Cohen and LogLog, or even the top few highest priority items, such as Coordinated Sampling and BJKST, are unable to properly account for the deletion of the current maximum or a high priority item. Similarly, once a bit i is set in the FM algorithm, the subsequent deletion of an item mapped to bit i does not mean the bit can be unset: there are likely to have been other un-deleted stream items that also mapped to bit i. In the case of FM, deletions can be handled by replacing each bit with a running counter that is incremented on insertions and decremented on deletions—at a cost of increasing the space needed by a logarithmic factor.

Update streams generalize the insertion and deletion scenario by having each stream item being a pair (v, Δ), where $\Delta > 0$ specifies Δ insertions of the value v and $\Delta < 0$ specifies $|\Delta|$ deletions of the value v. The resulting frequency $f_v = \sum_{(v,\Delta)\in S} \Delta$ of value v is assumed to be nonnegative. The metric F_0 is the number of distinct values v with $f_v > 0$. The above variant of FM with counters instead of bits readily handles update streams. Cormode et al. [9] devised an (ϵ, δ)-approximation scheme, called l_0 *sketch*, for distinct-values estimation over update streams. Unlike any of the approaches discussed thus far, the l_0 sketch algorithm uses properties of *stable distributions*, and requires floating point arithmetic. The algorithm uses $O(\frac{1}{\epsilon^2}\log(1/\delta))$ floating point numbers and $O(\frac{1}{\epsilon^2}\log(1/\delta))$ floating point operations per stream item.

Recently, Ganguly [14] devised two (ϵ, δ)-approximation schemes for update streams. One uses $O(\frac{1}{\epsilon^2}(\log n + \log N)\log N \log(1/\delta))$ memory bits and $O(\log(1/\epsilon)\cdot \log(1/\delta))$ operations to process each stream update. The other uses a factor of $(\log(1/\epsilon) + \log(1/\delta))$ times more space but reduces the number of operations to only $O(\log(1/\epsilon) + \log(1/\delta))$. Both algorithms return a uniform sampling of the distinct values, as well as an estimate.

6.4 Distributed Streams

In a number of the motivating scenarios, the goal is to estimate the number of distinct values over a collection of *distributed streams*. For example, in network monitoring, each router observes a stream of packets and the goal is to estimate the number of distinct “values” (e.g., destination IP addresses, source-destination pairs, requested urls, etc.) across all the streams. Formally, we have $t \geq 2$ data streams, $S_1, S_2, \ldots, S_t$, of items, where each item is from a universe of n possible values. Each stream S_i is observed and processed by a party, P_i, independently of the other streams, in one pass and with limited working space memory. The working space can be initialized (prior to observing any stream data) with data shared by all parties, so that, for example, all parties can use the same random hash function(s). The goal is to estimate the number of distinct values in the multi-set arising from concatenating all t streams. For example, in the $t = 2$ streams in Fig. 1, there are 15 distinct values in the two streams altogether.

In response to a request to produce an estimate, each party sends a message (containing its current synposis or some function of it) to a Referee, who outputs the estimate based on these messages. Note that the parties do not communicate directly with one another, and the Referee does not directly observe any stream data. We are primarily interested in minimizing: (1) the workspace used by each party, and (2) the time taken by a party to process a data item.

As shown in Table 2, each of the algorithms discussed in this chapter can be readily adapted to handle the distributed streams setting. For example, the FM algorithm (Fig. 3) can be applied to each stream independently, using the exact same hash function across all streams, to generate a bit vector $M[\cdot]$ for each stream. These bit vectors are sent to the Referee. Because the same hash function was used by all parties, the bit-wise OR of these t bit vectors yields exactly the bit vector that would have been produced by running the FM algorithm on the concatenation of the t streams (or any other interleaving of the stream data). Thus, the Referee computes this bit-wise OR, and then computes Z, the least significant 0-bit in the result. As in the original FM algorithm, we reduce the variance in the estimator by using k hash functions instead of just 1, where all parties use the same k hash functions. The Referee computes the Z corresponding to each hash function, then computes the average, $\bar{Z}$, of these Z's, and finally, outputs $\lfloor 2^{\bar{Z}}/.77351 \rfloor$. The error guarantees of this distributed streams algorithm match the error guarantees in Theorem 1 for the single-stream algorithm. Moreover, the *per-party* space bound and the *per-item* time bound also match the space and time bounds in Theorem 1.

Similarly, PCSA, FM with $\log^2$ space, AMS, Cohen, and LogLog can be adapted to the distributed streams setting in a straightforward manner, preserving the error guarantees, per-party space bounds, and per-item time bounds of the single-stream algorithm.

A bit less obvious, but still relatively straightforward, is adapting algorithms that use dynamic threshholds on what to keep and what to discard, where the threshhold adjusts to the locally-observed data distribution. The key observation for why these algorithms do not pose a problem is that we can match the error guarantees of the single-stream algorithm by having the Referee use the strictest threshhold among all the local threshholds. (Here, "strictest" means that the smallest fraction of the data universe has its items kept.) Namely, if ℓ is the strictest threshhold, the Referee "subsamples" the synopses from all the parties by applying the threshhold ℓ to the synopses. This unifies all the synopses to the same threshhold, and hence the Referee can safely combine these synopses and compute an estimate.

Consider, for example, the Coordinated Sampling algorithm (Fig. 9). Each party sends its sets $S_1, \ldots, S_k$ and levels $\ell_1, \ldots, \ell_k$ to the Referee. For $j =$

$1, \ldots, k$, the Referee computes ℓ_j^*, the maximum value of the ℓ_j's from all the parties. Then, for each j, the Referee subsamples each of the S_j from the t parties, by discarding from S_j all pairs (v', b') such that $b' < \ell_j^*$. Next, for each j, the Referee determines the union, S_j^*, of all the subsampled S_j's. Finally, the Referee outputs the median over all j of $|S_j^*| \cdot 2^{l_j^*}$. The error guarantees, per-party space bound, and per-item time bound match those in Theorem 5 for the single-stream algorithm [17]. The error guarantees follow because (1) S_j^* contains *all* pairs (v, b) with $b \geq \ell_j^*$ across all t streams, and (2) the size of each S_j^* is at least as big as the size of the S_j at a party with level $\ell_j = \ell_j^*$ (i.e., at a party with no subsampling), and this latter size was already sufficient to get a good estimate in the single-stream setting.

6.5 Order- and Duplicate-Insensitive (ODI)

Another interesting setting for distinct-values estimation algorithms arises in robust aggregation in wireless sensor networks. In sensor network aggregation, an aggregate function (e.g., count, sum, average) of the sensors' readings is often computed by having the wireless sensor nodes organize themselves into a tree (with the base station as the root). The aggregate is computed bottom-up starting at the leaves of the tree: each internal node in the tree combines its own reading with the partial results from its children, and sends the result to its parent. For a sum, for example, the node's reading is added to the sum of its childrens' respective partial sums. This conserves energy because each sensor node sends only one short message, in contrast to the naive approach of having all readings forwarded hop-by-hop to the base station.

Aggregating along a tree, however, is not robust against message loss, which is common in sensor networks, because each dropped message loses a subtree's worth of readings. Thus, Considine et al. [8] and Nath et al. [24] proposed using multi-path routing for more robust aggregation. In one scheme, the nodes organize themselves into "rings" around the base station, where ring i consists of all nodes that are i hops from the base station. As in the tree, aggregation is done bottom-up starting with the nodes in the ring furthest from the base station (the "leaf" nodes). In contrast to the tree, however, when a node sends its partial result, there is no designated parent. Instead, all nodes in the next closest ring that overhear the partial result incorporate it into their accumulating partial results. Because of the added redundancy, the aggregation is highly robust to message loss, yet the energy consumption is similar to the (non-robust) tree because each sensor node sends only one short message.

On the other hand, because of the redundancy, partial results are accounted for multiple times. Thus, the aggregation must be done in a duplicate-insensitive fashion. This is where distinct-values estimation algorithms come in. First, if the goal is to count the number of distinct "values" (e.g., the number of distinct temperature readings), then a distinct-values estimation algorithm can be used, as long as the algorithm works for distributed streams and is insensitive to the duplication and observation re-ordering that arises in the scheme. An aggregation algorithm with the combined properties of order- and duplicate-insensitivity is called *ODI*-correct [24]. Second, if the goal is to count the number of sensor nodes whose readings satisfy a given boolean predicate (e.g., nodes with temperature readings below freezing), then again a distinct-values estimation algorithm can be used, as follows. Each sensor node whose reading satisfies the predicate uses its unique sensor id as its "value". Then the number of distinct values in the sensor network is precisely the desired count. Thus any distributed, ODI-correct distinct-values estimation algorithm can be used.

As shown in Table 2, most of the algorithms discussed in this chapter are ODI-correct. For example, the FM algorithm (Fig. 3) is insensitive to both re-ordering and duplication: the bits that are set in an FM bit vector are independent of both the order in which stream items are processed and any duplication of "partial-result" bit vectors (i.e., bit vectors corresponding to a subset of the stream items). Moreover, Considine et al. [8] showed how the FM algorithm can be effectively adapted to use only $O(\log\log n)$ bit messages in this setting. Similarly, most of the other algorithms are ODI-correct, as can be proved formally using the approach described in [24].

6.6 Additional Settings

We conclude this chapter by briefly mentioning three additional important settings considered in the literature.

The first setting is distinct-values estimation when each value is *unique*. This setting occurs, for example, in distributed census taking over mobile objects (e.g., [31]). Here, there are a large number of objects, each with a unique id. The goal is to estimate how many objects there are despite the constant motion of the objects, while minimizing the communication. Clearly, any of the distributed distinct-values estimation algorithms discussed in this chapter can be used. Note, however, that the setting enables a practical optimization: hash functions are not needed to map values to bit positions or levels. Instead, independent coin tosses can be used at each object; the desired exponential distribution can be obtained by flipping a fair coin until the first heads and counting the number of tails observed prior to the first head. (The unique id is not even used.) Obviating the need for hash functions eliminates their space and time overhead. Thus, for example, only $O(\log\log n)$-bit synopses are needed for the AMS algorithm. Note that hash functions were needed before to ensure that the multiple occurrences of the same value all map to the same bit position or level; this feature is not needed in the setting with unique values.

A second setting, studied by Bar-Yossef et al. [3] and Pavan and Tirthapura [28], seeks to estimate the number of distinct values where each stream item is a *range* of integers. For example, in the 4-item stream $[2, 5], [10, 12], [4, 8], [6, 7]$, there are 10 distinct values, namely, 2, 3, 4, 5, 6, 7, 8, 10, 11, and 12. Pavan and Tirthapura present an (ϵ, δ)-approximation scheme that uses $O(\frac{\log n \log(1/\delta)}{\epsilon^2})$ memory bits, and performs an amortized $O(\log(1/\delta)\log(n/\epsilon))$ operations per stream item. Note that although a single stream item introduces up to n distinct values into the stream, the space and time bounds have only a logarithmic (and not a linear) dependence on n.

Finally, a third important setting generalizes the distributed streams setting from just the union (concatenation) of the data streams to arbitrary set-expressions among the streams (including intersections and set differences). In this setting the number of distinct values corresponds to the *cardinality* of the resulting set. Ganguly et al. [15] showed how techniques for distinct values estimation can be generalized to handle this much richer setting.

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. of Computer and System Sciences **58** (1999) 137–147
2. Bar-Yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D., Trevisan, L.: Counting distinct elements in a data stream. In: Proc. 6th International Workshop on Randomization and Approximation Techniques. (2002) 1–10
3. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA). (2002)
4. Bunge, J., Fitzpatrick, M.: Estimating the number of species: A review. J. of the American Statistical Association **88** (1993) 364–373
5. Charikar, M., Chaudhuri, S., Motwani, R., Narasayya, V.: Towards estimation error guarantees for distinct values. In: Proc. 19th ACM Symp. on Principles of Database Systems. (2000) 268–279
6. Chaudhuri, S., Motwani, R., Narasayya, V.: Random sampling for histogram construction: How much is enough? In: Proc. ACM SIGMOD International Conf. on Management of Data. (1998) 436–447
7. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. J. of Computer and System Sciences **55** (1997) 441–453
8. Considine, J., Li, F., Kollios, G., Byers, J.: Approximate aggregation techniques for sensor databases. In: Proc. 20th International Conf. on Data Engineering. (2004) 449–460
9. Cormode, G., Datar, M., Indyk, P., Muthukrishnan, S.: Comparing data streams using Hamming norms (how to zero in). In: Proc. 28th International Conf. on Very Large Data Bases. (2002) 335–345
10. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM Journal on Computing **31** (2002) 1794–1813
11. Durand, M., Flajolet, P.: Loglog counting of large cardinalities. In: Proc. 11th European Symp. on Algorithms. (2003) 605–617
12. Estan, C., Varghese, G., Fisk, M.: Bitmap algorithms for counting active flows on high speed links. In: Proc. 3rd ACM SIGCOMM Conf. on Internet Measurement. (2003) 153–166
13. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. of Computer and System Sciences **31** (1985) 182–209
14. Ganguly, S.: Counting distinct items over update streams. In: Proc. 16th International Symp. on Algorithms and Computation. (2005) 505–514
15. Ganguly, S., Garofalakis, M., Rastogi, R.: Tracking set-expression cardinalities over continuous update streams. VLDB J. **13** (2004) 354–369

16. Gibbons, P.B.: Distinct sampling for highly-accurate answers to distinct values queries and event reports. In: Proc. 27th International Conf. on Very Large Data Bases. (2001) 541–550
17. Gibbons, P.B., Tirthapura, S.: Estimating simple functions on the union of data streams. In: Proc. 13th ACM Symp. on Parallel Algorithms and Architectures. (2001) 281–291
18. Gibbons, P.B., Tirthapura, S.: Distributed streams algorithms for sliding windows. In: Proc. 14th ACM Symp. on Parallel Algorithms and Architectures. (2002) 63–72
19. Haas, P.J., Naughton, J.F., Seshadri, S., Stokes, L.: Sampling-based estimation of the number of distinct values of an attribute. In: Proc. 21st International Conf. on Very Large Data Bases. (1995) 311–322
20. Haas, P.J., Stokes, L.: Estimating the number of classes in a finite population. J. of the American Statistical Association **93** (1998) 1475–1487
21. Hou, W.C., Özsoyoğlu, G., Taneja, B.K.: Statistical estimators for relational algebra expressions. In: Proc. 7th ACM Symp. on Principles of Database Systems. (1988) 276–287
22. Hou, W.C., Özsoyoğlu, G., Taneja, B.K.: Processing aggregate relational queries with hard time constraints. In: Proc. ACM SIGMOD International Conf. on Management of Data. (1989) 68–77
23. Kumar, A., Xu, J., Wang, J., Spatscheck, O., Li, L.: Space-code bloom filter for efficient per-flow traffic measurement. In: Proc. IEEE INFOCOM. (2004)
24. Nath, S., Gibbons, P.B., Seshan, S., Anderson, Z.: Synopsis diffusion for robust aggregation in sensor networks. In: Proc. 2nd ACM International Conf. on Embedded Networked Sensor Systems. (2004) 250–262
25. Naughton, J.F., Seshadri, S.: On estimating the size of projections. In: Proc. 3rd International Conf. on Database Theory. (1990) 499–513
26. Olken, F.: Random Sampling from Databases. PhD thesis, Computer Science, U.C. Berkeley (1993)
27. Palmer, C.R., Gibbons, P.B., Faloutsos, C.: ANF: A fast and scalable tool for data mining in massive graphs. In: Proc. 8th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining. (2002) 81–90
28. Pavan, A., Tirthapura, S.: Range-efficient computation of F0 over massive data streams. In: Proc. 21st IEEE International Conf. on Data Engineering. (2005) 32–43
29. Poosala, V.: Histogram-based Estimation Techniques in Databases. PhD thesis, Univ. of Wisconsin-Madison (1997)
30. Poosala, V., Ioannidis, Y.E., Haas, P.J., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: Proc. ACM SIGMOD International Conf. on Management of Data. (1996) 294–305
31. Tao, Y., Kollios, G., Considine, J., Li, F., Papadias, D.: Spatio-temporal aggegration using sketches. In: Proc. 20th International Conf. on Data Engineering. (2004) 214–225
32. Venkataraman, S., Song, D., Gibbons, P.B., Blum, A.: New streaming algorithms for high speed network monitoring and internet attacks detection. In: Proc. 12th ISOC Network and Distributed Security Symp. (2005)
33. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Transactions on Database Systems **15** (1990) 208–229
34. Woodruff, D.: Optimal space lower bounds for all frequency moments. In: Proc. 15th ACM-SIAM Symp. on Discrete Algorithms. (2004) 167–175