A Tutorial on Network Data Streaming

Jun (Jim) Xu
Networking and Telecommunications Group
College of Computing
Georgia Institute of Technology

Motivation for new network monitoring algorithms

Problem: we often need to monitor network links for quantities such as

- Elephant flows (traffic engineering, billing)
- Number of distinct flows, average flow size (queue management)
- Flow size distribution (anomaly detection)
- Per-flow traffic volume (anomaly detection)
- Entropy of the traffic (anomaly detection)
- Other “unlikely” applications: traffic matrix estimation, P2P routing, IP traceback

The challenge of high-speed network monitoring

- Network monitoring at high speed is challenging
  - packets arrive every 25ns on a 40 Gbps (OC-768) link
  - has to use SRAM for per-packet processing
  - per-flow state too large to fit into SRAM
  - traditional solution of sampling is not accurate due to the low sampling rate dictated by the resource constraints (e.g., DRAM speed)

Network data streaming – a smarter solution

- **Computational model:** process a long stream of data (packets) in one pass using a small (yet fast) memory
- **Problem to solve:** need to answer some queries about the stream at the end or continuously
- **Trick:** try to remember the most important information about the stream *pertinent to the queries* – learn to forget unimportant things
- **Comparison with sampling**: streaming peruses every piece of data for most important information while sampling digests a small percentage of data and absorbs all information therein.

The “hello world” data streaming problem

- Given a long stream of data (say packets) $d_1, d_2, \cdots$, count the number of distinct elements (F_0) in it
- Say in a, b, c, a, c, b, d, a – this number is 4
- Think about trillions of packets belonging to billions of flows
- A simple algorithm: choose a hash function h with range (0,1)
- $\hat{X} := \min(h(d_1), h(d_2), ...)$
- We can prove $E[\hat{X}] = 1/(F_0 + 1)$ and then estimate F_0 using method of moments
- Then averaging hundreds of estimations of F_0 up to get an accurate result

Another solution to the same problem [Whang et al., 1990]

- Initialize a bit array A of size m to all 0 and fix a hash function h that maps data items into a number in $\{1, 2, ..., m\}$.
- For each incoming data item x_t, set $A[h(x_t)]$ to 1
- Let m_0 be the number of 0's in A
- Then $\hat{F}_0 = m \times \ln(m/m_0)$
  - Given an arbitrary index i, let Y_i the number of elements mapped to it and let X_i be 1 when $Y_i = 0$. Then $E[X_i]$ = $Pr[Y_i = 0]$ = $(1 - 1/m)^{F_0} \approx e^{-F_0/m}$.
  - Then $E[X] = \sum_{i=1}^{m} E[X_i] \approx m \times e^{-F_0/m}$.
  - By the method of moments, replace $E[X]$ by m_0 in the above equation, we obtain the above unbiased estimator (also shown to be MLE).

Cash register and turnstile models [Muthukrishnan,]

- The implicit state vector (varying with time t) is the form $\vec{a} =< a_1, a_2, ..., a_n >$
- Each incoming data item x_t is in the form of $< i(t), c(t) >$, in which case $a_{i(t)}$ is incremented by $c(t)$
- Data streaming algorithms help us approximate functions of $\vec{a}$ such as $F_0(\vec{a}) = \sum_{i=1}^{n} |a_i|^0$ (number of distinct elements).
- Cash register model: $c(t)$ has to be positive (often is 1 in networking applications)
- Turnstile model: $c(t)$ can be both positive and negative

Estimating the sample entropy of a stream [Lall et al., 2006]

- Note that $\sum_{i=1}^{n} a_i = N$
- The *sample entropy* of a stream is defined to be

$$H(\vec{a}) \equiv -\sum_{i=1}^{n} (a_i/N) \log (a_i/N)$$

- All logarithms are base 2 and, by convention, we define $0 \log 0 \equiv 0$
- We extend the previous algorithm ([Alon et al., 1999]) to estimate the entropy
- Another team obtained similar results simultaneously

The concept of entropy norm

We will focus on computing the entropy norm value $S \equiv \sum_{i=1}^{n} a_i \log a_i$ and note that

$$\begin{aligned} H &= -\sum_{i=1}^{n} \frac{a_i}{N} \log\left(\frac{a_i}{N}\right) \\ &= \frac{-1}{N}\left[\sum_i a_i \log a_i - \sum_i a_i \log N\right] \\ &= \log N - \frac{1}{N}\sum_i a_i \log a_i \\ &= \log N - \frac{1}{N} S, \end{aligned}$$

so that we can compute H from S if we know the value of N.

(ϵ, δ)-Approximation

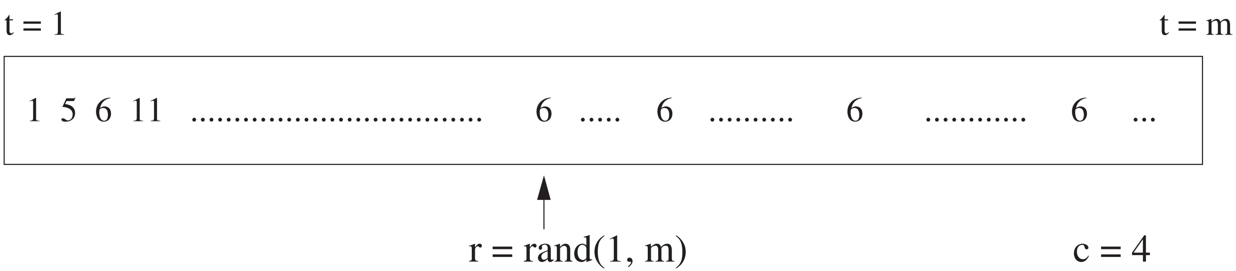
An (ϵ, δ)-approximation algorithm for X is one that returns an estimate X' with relative error more than ϵ with probability at most δ. That is

$$Pr(|X - X'| \geq X\epsilon) \leq \delta.$$

For example, the user may specify $\epsilon = 0.05, \delta = 0.01$ (i.e., at least 99% of the time the estimate is accurate to within 5% error). These parameters affect the space usage of the algorithm, so there is a tradeoff of accuracy versus space.

The Algorithm

The strategy will be to sample as follows:



and compute the following estimating variable:

$$X = N\left(c\log c - (c-1)\log(c-1)\right).$$

can be viewed as $f'(x)|_{x=c}$ where $f(x) = x\log x$

This estimator $X = N\left(c\log c - (c-1)\log(c-1)\right)$ is an unbiased estimator of S since

$$
\begin{aligned}
E[X] &= \frac{N}{N}\sum_{i=1}^{n}\sum_{j=1}^{a_i}\left(j\log j - (j-1)\log(j-1)\right) \\
&= \sum_{i=1}^{n} a_i \log a_i \\
&= S.
\end{aligned}
$$

Next, we bound the variance of X:

$$
\begin{aligned}
Var(X) &= E(X^2) - E(X)^2 \leq E(X^2) \\
&= \frac{N^2}{N}[\sum_{i=1}^{n}\sum_{j=2}^{a_i}(j\log j - (j-1)\log(j-1))^2] \\
&\leq N\sum_{i=1}^{n}\sum_{j=2}^{a_i}(2\log j)^2 \leq 4N\sum_{i=1}^{n} a_i \log^2 a_i \\
&\leq 4N\log N(\sum_i a_i \log a_i) \leq 4(\sum_i a_i \log a_i)\log N(\sum_i a_i \log a_i) \\
&= 4S^2 \log N,
\end{aligned}
$$

assuming that, on average, each item appears in the stream at least twice.

Algorithm contd.

If we compute $s_1 = (32 \log N)/\epsilon^2$ such estimators and compute their average Y, then by Chebyschev's inequality we have:

$$\begin{aligned} Pr(|Y - S| > \epsilon S) &\leq \frac{Var(Y)}{\epsilon^2 S^2} \\ &\leq \frac{4S^2 \log N}{s_1 \epsilon^2 S^2} = \frac{4 \log N}{s_1 \epsilon^2} \\ &\leq \frac{1}{8}. \end{aligned}$$

If we repeat this with $s_2 = 2 \log(1/\delta)$ groups and take their median, by a Chernoff bound we get more than ϵS error with probability at most δ.

Hence, the median of averages is an (ϵ, δ)-approximation.

The Sieving Algorithm

- KEY IDEA: Separating out the elephants decreases the variance, and hence the space usage, of the previous algorithm.
- Each packet is now sampled with some fixed probability p.
- If a particular item is sampled *two or more* times, it is considered an elephant and its exact count is estimated.
- For all items that are not elephants we use the previous algorithm.
- The entropy is estimated by adding the contribution from the elephants (from their estimated counts) and the mice (using the earlier algorithm).

Estimating the k_{th} moments [Alon et al., 1999]

- Problem statement (cash register model with increments of size 1): approximating $F_k = \sum_{i=1}^{n} a_i^k$
- Given a stream of data $x_1, x_2, ..., x_N$, the algorithm samples an item uniformly randomly at $s_1 \times s_2$ locations like before
- If it is already in the hash table, increment the corresponding counter, otherwise add a new entry $< i, 1 >$ to it
- After the measurement period, for each record $< i, c_i >$, obtain an estimate as c_i^k - c_i^{k-1} ($f'(x)|_{x=c}$ where $f(x) = x^k$)
- Median of the means of these $s_1 \times s_2$ estimates like before
- Our algorithm is inspired by this one

Tug-of-War sketch for estimating the 2nd moment [Alon et al., 1999]

- Fix an explicit set $V = \{v_1, v_2, ..., v_h\}$ of $h = O(n^2)$ vectors of length n with +1 and -1 entries
- These vectors are 4-wise independent, that is, for every four distinct indices i_1, ..., i_4 and every choice of ϵ_1, ..., $\epsilon_4 \in \{-1, +1\}$, exactly 1/16 of the vectors in V take these values – they can be generated using BCH codes using a small seed
- randomly choose $v =< \epsilon_1, \epsilon_2, ..., \epsilon_n >$ from V, and let X be square of the dot product of v and the implicit state vector, i.e., $X = (\sum_{i=1}^{n} \epsilon_i \times a_i)^2$. This can be calculated in one pass.
- Then take the median of a bunch of such $X's$

Elephant detection algorithms

- Problem: finding all the elements whose frequency is over θN
- There are three types of solutions:
  - Those based on "intelligent sampling"
  - Those based on a sketch that provides a "reading" on the approximate size of the flow that an incoming packet belongs to, in combination with a heap (to keep the largest ones).
  - The hybrid of them
- We will not talk about change detection, as it can be viewed as a variant of the elephant detection problem

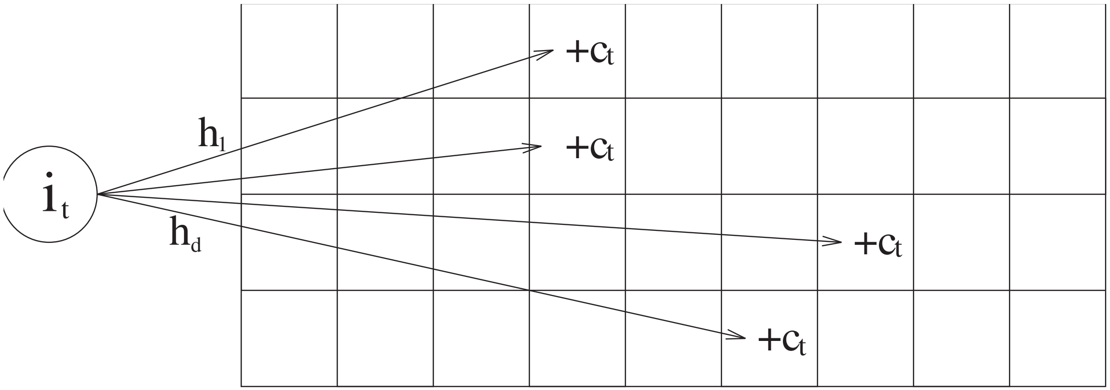
Karp-Shenker-Papadimitriou Algorithm

- A deterministic algorithm to guarantee that all items whose frequency count is over θN are reported:

  1. maintain a set of $< e, f >$ tuples
  2. foreach incoming data x_j
  3.   search/increment/create an item in the set
  4.   if the set has more than $1/\theta$ items then
  5.   decrement the count of each item in the set by 1,
  6.   remove all zero-count items from the set
  7. Output all the survivors at the end

- Not suitable for networking applications

Count-Min or Cormode-Muthukrishnan sketch



- The count is simply the minimum of all the counts
- One can answer several different kinds of queries from the sketch (e.g., point estimation, range query, heavy hitter, etc.)
- It is a randomized algorithm (with the use of hash functions)

Elephant detection algorithm with the CM sketch

- maintain a heap H of "small" size

1. for each incoming data item x_t
2. get its approximate count f from the CM sketch
3. if $f \geq \theta t$ then
4. increment and/or add x_t to H
5. delete $H.min()$ if it falls under θt
6. output all above-threshold items from H

- Suitable for networking applications

Charikar-Chen-(Farach-Colton) sketch

- It is a randomized algorithm (with the use of hash functions)
- Setting: An $m \times b$ counter array C, hash functions h_1, ..., h_m that map data items to $\{1, ..., b\}$ and s_1, ..., s_m that map data items to $\{-1, +1\}$.
- Add(x_t): compute $i_j := h_j(x_t)$, $j = 1, ..., m$, and then increment $C[j][i_j]$ by $s_j(x_t)$.
- Estimate(x_t): return the median$_{1 \leq j \leq m}$ $\{C[j][i_j] \times s_j(x_t)\}$
- Suitable for networking applications

Sticky sampling algorithm [Manku and Motwani, 2002]

- sample (and hold) initially with probability 1 for first $2t$ elements
- sample with probability $1/2$ for the next $2t$ elements and resample the first $2t$ elements
- sample with probability $1/4$ for the next $4t$ elements, resample, and so on ...
- A little injustice to describe it this way as it is earlier than [Estan and Varghese, 2002]
- Not suitable for networking applications due to the need to resample

Lossy counting algorithm [Manku and Motwani, 2002]

- divide the stream of length N into buckets of size $\omega = \lceil 1/\theta \rceil$ each
- maintain a set D of entries in the form $< e, f, \Delta >$

1. foreach incoming data item x_t
2. $\quad b := \lceil \frac{t}{\omega} \rceil$
3. $\quad$ if x_t is in D then increment its f accordingly
4. $\quad$ else add entry $< x_t, 1, b - 1 >$ to D
5. $\quad$ if t is divisible by ω then
6. $\qquad$ delete all items e whose $f + \Delta \leq b$
7. return all items whose $f \geq (\theta - \epsilon)N$.

- Not suitable for networking applications

Sample-and-hold [Estan and Varghese, 2002]

- maintain a set D of entries in the form $< e, f >$

1. foreach incoming data item x_t
2. if it is in D then increment its f
3. else insert a new entry to D with probability $b * 1/(N\theta)$
4. return all items in D with high frequencies

Multistage filter [Estan and Varghese, 2002]

- maintain multiple arrays of counters C_1, C_2, ..., C_m of size b and a set D of entries $< e, f >$, and let h_1, h_2, ..., h_m be hash functions that map data items to $\{1, 2, ..., b\}$.

  1. for each incoming data item x_t
  2.   increment $C_i[h_i(x_t)]$, $i = 1, ..., m$ by 1 if possible
  3.   if these counters reach value MAX
  4.   then insert/increment x_t into D
  5. Output all items with count at least $N \times \theta - MAX$

- Conservative update: only increment the minimum(s)
- Serial version is more memory efficient, but increases delay

Estimating L_1 norm [Indyk, 2006]

- Recall the turnstile model (increments can be both positive and negative)
- L_1 norm is exactly $L_1(\vec{a}) = \sum_{i=1}^{n} |a_i|$ and is more general than frequency moments, under the turnstile model
- Algorithm to estimate the L_1 norm:

1. prescribe independent hash functions h_1, ..., h_m that maps a data item into a Cauchy random variable distributed as $f(x) = \frac{1}{\pi}\frac{1}{1+x^2}$ and initialize real-valued registers r_1, ..., r_m to 0.0
2. for each incoming data item $x_t =< i(t), c_i(t) >$
3. obtain $v_1 = h_1(i(t))$, ..., $v_m = h_m(i(t))$
4. increment r_1 by v_1, r_2 by v_2, ..., and r_m by v_m
5. return median($|r_1|$, $|r_2|$, ..., $|r_m|$)

Why this algorithm works [Indyk, 2006]

- Property of Cauchy distribution: if X_1, X_2, X are standard Cauchy RV's, and X_1 and X_2 are independent, then $aX_1 + bX_2$ has the same distribution as $(|a| + |b|)X$
- Given the actual state vector as $< a_1, a_2, ..., a_n >$, after the execution of this above algorithm, we get in each r_i a random variable of the following format $a_1 \times X_1 + a_2 \times X_2 + ... + a_n \times X_n >$, which has the same distribution as $(\sum_{i=1}^{n} |a_i|)X$
- Since median($|X|$) = 1 (or $F_X^{-1}(0.75) = 1$), the estimator simply uses the sample median to approximate the distribution median
- Why not "method of moments"?

The theory of stable distributions

- The existence of p-stable distributions ($S(p)$, $0 < p \leq 2$) is discovered by Paul Levy about 100 years ago (p replaced with α in most of the mathematical literature).
- Property of p-stable distribution: let X_1, ..., X_n denote mutually independent random variables that have distribution $S(p)$, then $a_1 X_1 + a_2 X_2 + \ldots + a_n X_n$ and $(|a_1|^p + |a_2|^p + \ldots + |a_n|^p)^{1/p} X$ are identically distributed.
- Cauchy is 1-stable as shown above and Gaussian ($f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$) is 2-stable

The theory of stable distributions, contd.

Although analytical expressions for the probability density function of stable distributions do not exist (except for $p = 0.5, 1, 2$), random variables with such distributions can be generated through the following formula:

$$X = \frac{\sin(p\theta)}{\cos^{1/p}\theta}\left(\frac{\cos(\theta(1-p))}{-\ln r}\right)^{1/p-1},$$

where θ is chosen uniformly in $[-\pi/2, \pi/2]$ and r is chosen uniformly in $[0, 1]$ [Chambers et al., 1976].

Fourier transforms of stable distributions

- Each $S(p)$ and correspondingly $f_p(x)$ can be uniquely characterized by its characteristic function as

$$E[e^{itX}] \equiv \int_{-\infty}^{\infty} f_p(x)(\cos(tx) + i \cdot \sin(tx)) = e^{-|t|^p}. \quad (1)$$

- It is not hard to verify that the fourier inverse transform of the above is a distribution function (per Polya's criteria)

- Verify the stableness property of $S(p)$:

$$\begin{aligned}
& E[e^{it(a_1X_1+a_2X_2+\ldots+a_nX_n)}] \\
= \; & E[e^{ita_1X_1}] \cdot E[e^{ita_2X_2}] \cdot \ldots \cdot E[e^{ita_nX_n}] \\
= \; & e^{-|a_1t|^p} \cdot e^{-|a_2t|^p} \cdot \ldots \cdot e^{-|a_2t|^p} \\
= \; & e^{-[(|a_1|^p+|a_2|^p+\ldots+|a_n|^p)^{1/p}|t|]^p} \\
= \; & E[e^{it((|a_1|^p+|a_2|^p+\ldots+|a_n|^p)^{1/p}X)}].
\end{aligned}$$

Estimating L_p norms for $0 < p \leq 2$

- L_p norm is defined as $L_p(\vec{a}) = (\sum_{i=1}^{n} |a_i|^p)^{1/p}$, which is equivalent to the p-th root of F_p (p_{th} moment) under the cash register model (not equivalent under the turnstile model)
- Simply modify the L_1 algorithm by changing the output of these hash functions h_1, ..., h_m from Cauchy (i.e., $S(1)$) to $S(p)$, and divide by distribution median of $S^+(p)$
- Moments of $S(p)$ may not exist but median estimator will work when m is reasonably large (say ≥ 5).
- Indyk's algorithms focus on reducing space complexity and some of these tricks may not be relevant to networking applications

Estimating entropy of OD Flows [Zhao et al., 2007]

- We can approximate $x \ln x$ by linear combinations of x^p for x on a fixed interval $[0, N]$ within relative error ϵ:

$$x \ln x \approx \frac{1}{2\alpha}(x^{1+\alpha} - x^{1-\alpha}), \text{ where } \alpha = \frac{\sqrt{\frac{6\epsilon}{1+6\epsilon}}}{\ln N}$$

- Proof: By Taylor expansion of $x^\alpha = e^{\alpha \ln x}$
- Therefore we can use $L_{1+\alpha}$ and $L_{1-\alpha}$ norms to estimate the entropy norm S

$$S = \sum a_i \ln a_i \approx \frac{1}{2\alpha}(\sum a_i^{1+\alpha} - \sum a_i^{1-\alpha})$$

- In parallel, we have an elephant-detection module that handles (with high probability) all the flows of size greater than N.

Estimating L_1 norm

- To estimate entropy, we need the entropy norm and the L_1 norm.
- We can utilize $L_{1+\alpha}$ and $L_{1-\alpha}$ norm estimations to avoid the overhead of L_1 norm estimation.

$$x \approx \frac{1}{2}(x^{1+\alpha} + x^{1-\alpha})$$

$$L_1(\vec{a}) = \sum a_i \approx \frac{1}{2}(\sum a_i^{1+\alpha} + \sum a_i^{1-\alpha})$$

Estimating L_p norm of OD flows

- Indyk's algorithm has Intersection Measurable Property (IMP).
- If we denote the L_p sketch at origin as $\vec{O}$, the one at destination as $\vec{D}$, and the median estimator as $\Lambda()$, then the L_p norm of the cross-traffic between origin and destination can be estimated by $\left(\frac{\Lambda(\vec{O})^p+\Lambda(\vec{D})^p-\Lambda(\vec{O}-\vec{D})^p}{2}\right)^{1/p}$ or $\left(\frac{\Lambda(\vec{O}+\vec{D})^p-\Lambda(\vec{O}-\vec{D})^p}{2^p}\right)^{1/p}$, where $\vec{O}+\vec{D}$ and $\vec{O}-\vec{D}$ are component-wise additions and subtractions of the sketches.
- $L_{1+\alpha}$ and $L_{1-\alpha}$ norm estimations of OD flows give us entropy estimation of OD flows.

Estimating L_p norm of OD flows

$$\Lambda(\vec{O})^p \approx |f_1|^p + \ldots + |f_k|^p + |g_1|^p + \ldots + |g_l|^p$$

$$\Lambda(\vec{D})^p \approx |f_1|^p + \ldots + |f_k|^p + |h_1|^p + \ldots + |h_m|^p$$

$$\Lambda(\vec{O} - \vec{D})^p \approx |g_1|^p + \ldots + |g_l|^p + |h_1|^p + \ldots + |h_m|^p$$

$$\Lambda(\vec{O} + \vec{D})^p \approx |2f_1|^p + \ldots + |2f_k|^p + |g_1|^p + \ldots + |g_l|^p + |h_1|^p + \ldots + |h_m|$$

Hence,

$$\frac{\Lambda(\vec{O})^p + \Lambda(\vec{D})^p - \Lambda(\vec{O} - \vec{D})^p}{2} \approx |f_1|^p + \ldots + |f_k|^p$$

$$\frac{\Lambda(\vec{O} + \vec{D})^p - \Lambda(\vec{O} - \vec{D})^p}{2^p} \approx |f_1|^p + \ldots + |f_k|^p.$$

Modifications to Indyk's Sketch

- Note that for every packet we have to perform hundreds or thousands of updates per packet (infeasible at line speeds).
- Solution: Hash packets into many (thousands of) buckets. For packets mapped to each bucket we apply Indyk's sketch with only a small number (tens) of registers and estimate the L_p norm of those packets. We add those results together to get the L_p norm of all the packets.
- The overall relative error is much small than the relative error of each bucket.
- We also use large lookup tables for the stable distribution RV's.

Data Streaming Algorithm for Estimating Flow Size Distribution [Kumar et al., 2004]

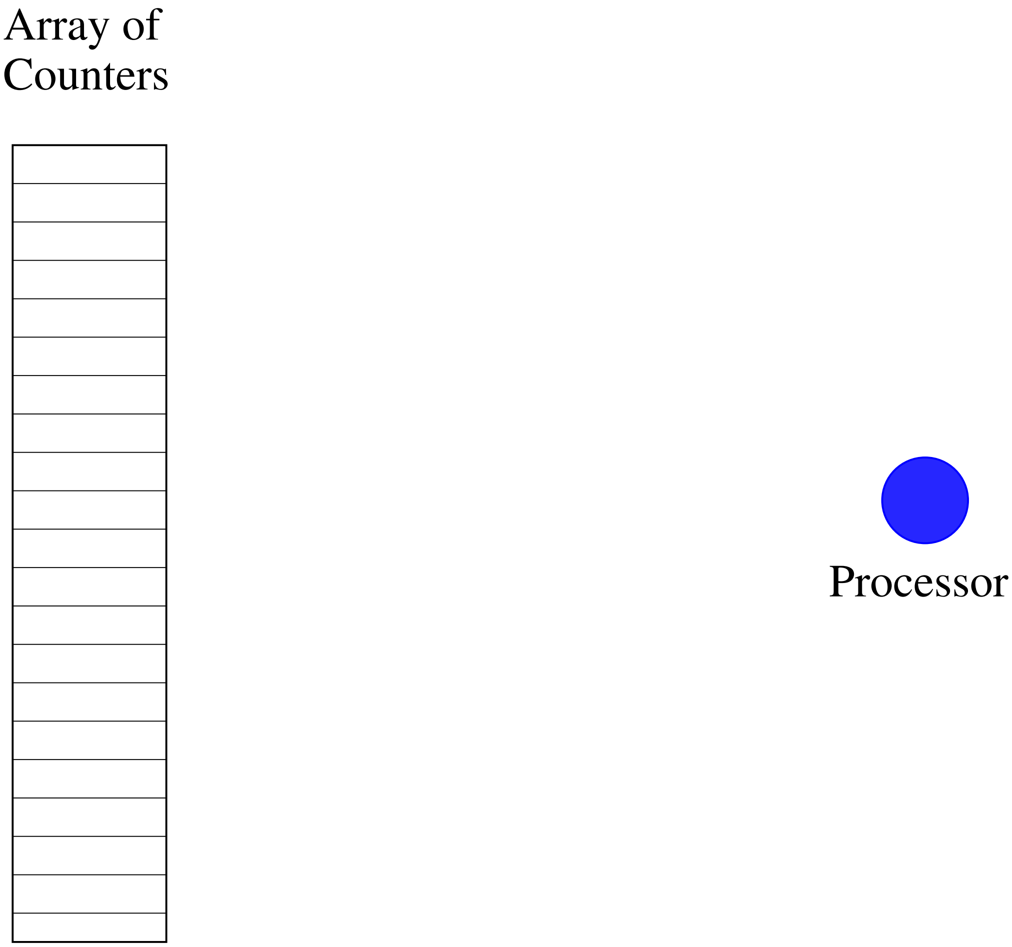
- **Problem:** To estimate the probability distribution of flow sizes. In other words, for each positive integer i, estimate n_i, the number of flows of size i.
- **Applications:** Traffic characterization and engineering, network billing/accounting, anomaly detection, etc.
- **Importance:** The mother of many other flow statistics such as average flow size (first moment) and flow entropy

- **Definition of a flow:** All packets with the same flow-label. The flow-label can be defined as any combination of fields from the IP header, e.g., $<$Source IP, source Port, Dest. IP, Dest. Port, Protocol$>$.

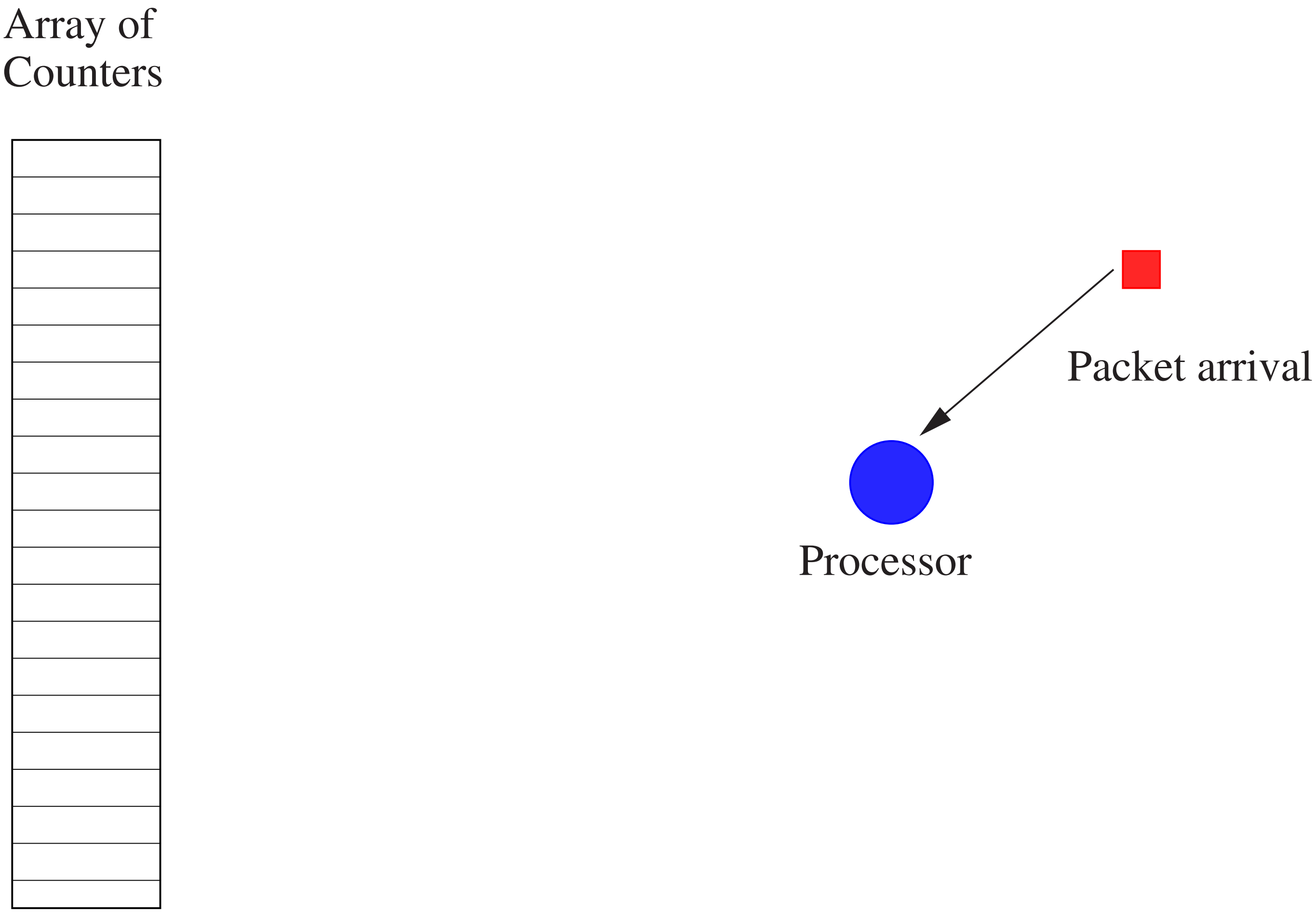
Architecture of our Solution — Lossy data structure

- Maintain an array of counters in fast memory (SRAM).
- For each packet, a counter is chosen via hashing, and incremented.
- No attempt to detect or resolve collisions.
- Each 64-bit counter only uses 4-bit of SRAM (due to [Zhao et al., 2006b])
- Data collection is lossy (erroneous), but very fast.

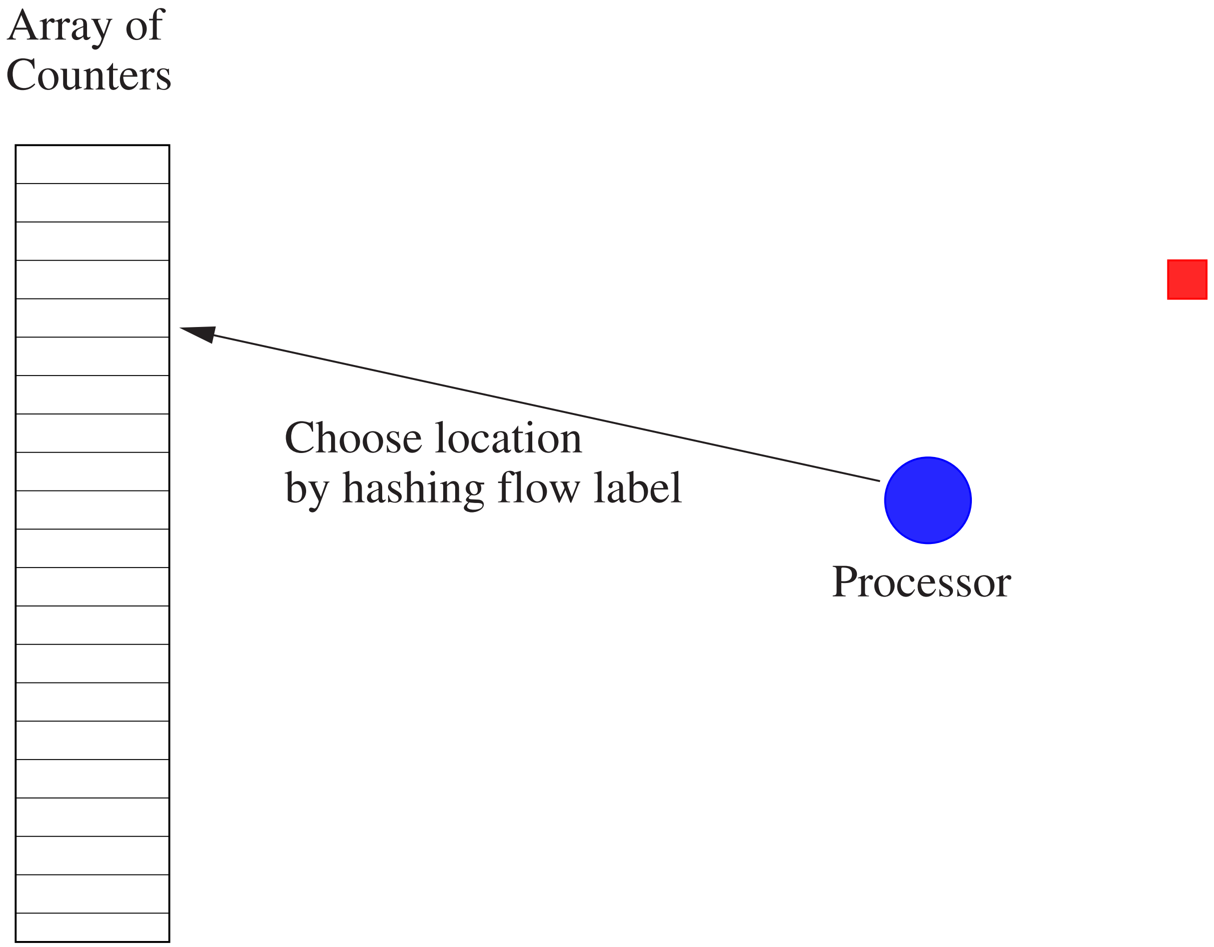
Counting Sketch: Array of counters



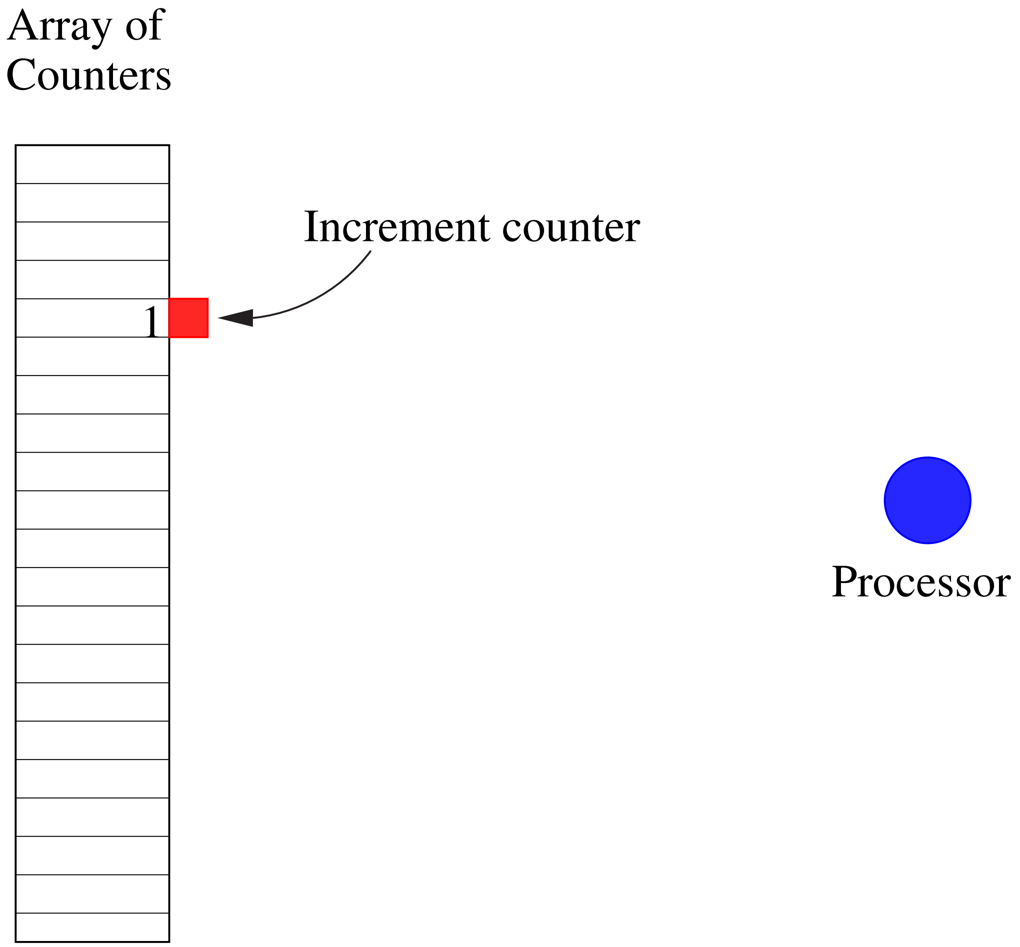
Counting Sketch: Array of counters



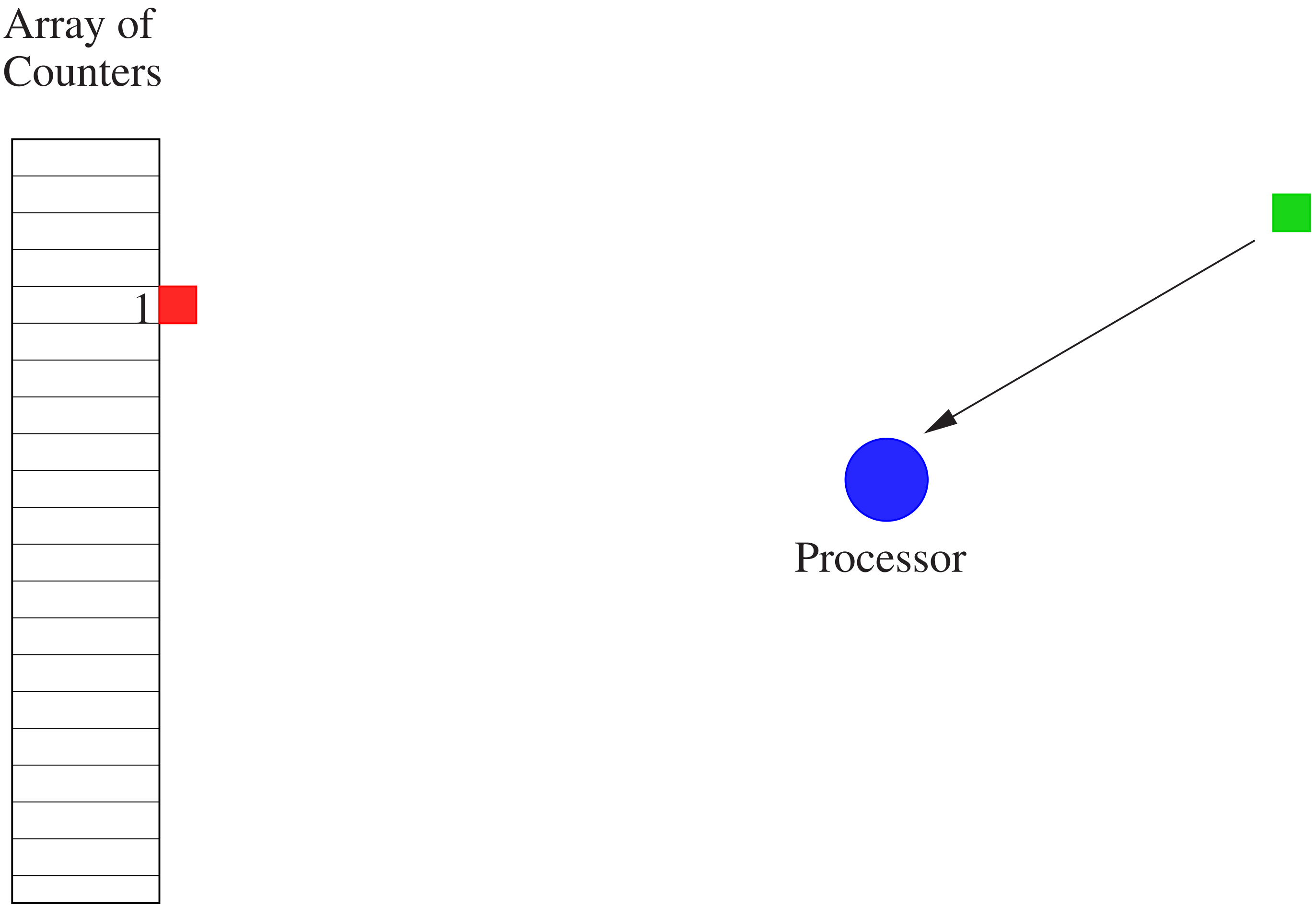
Counting Sketch: Array of counters



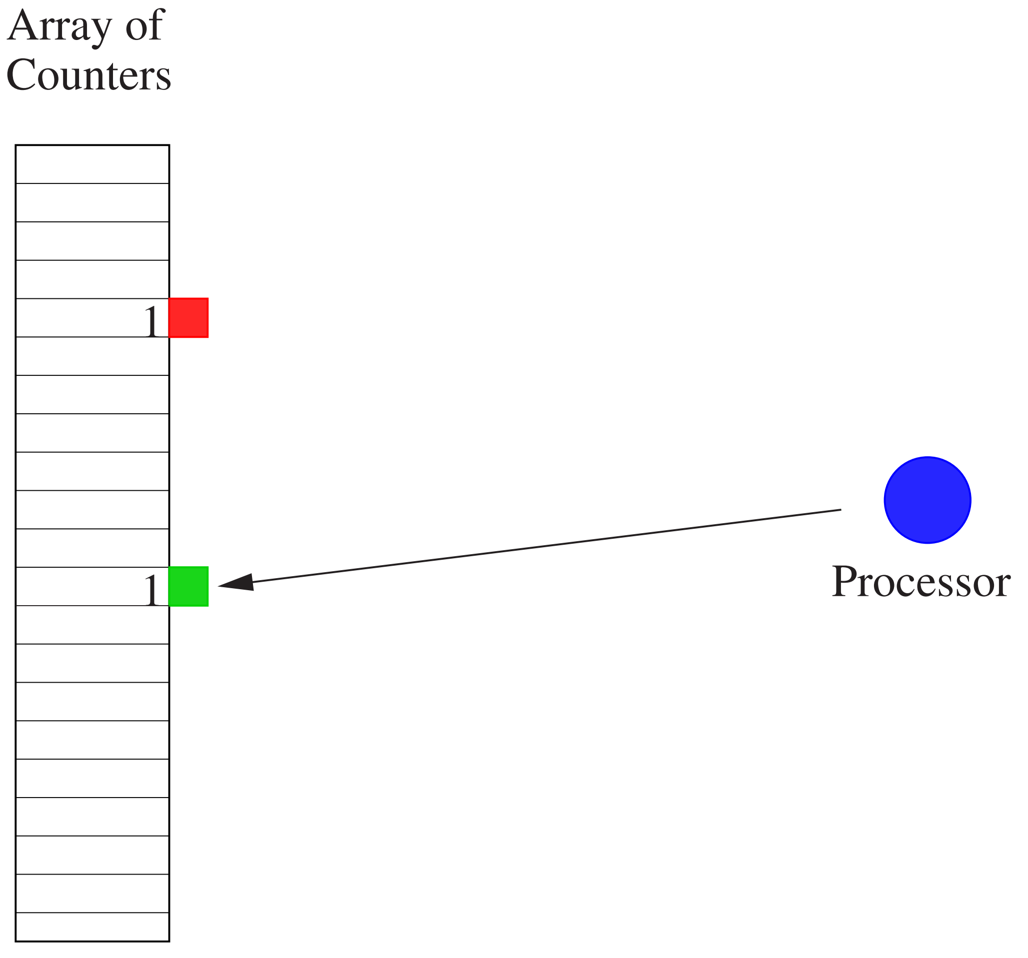
Counting Sketch: Array of counters



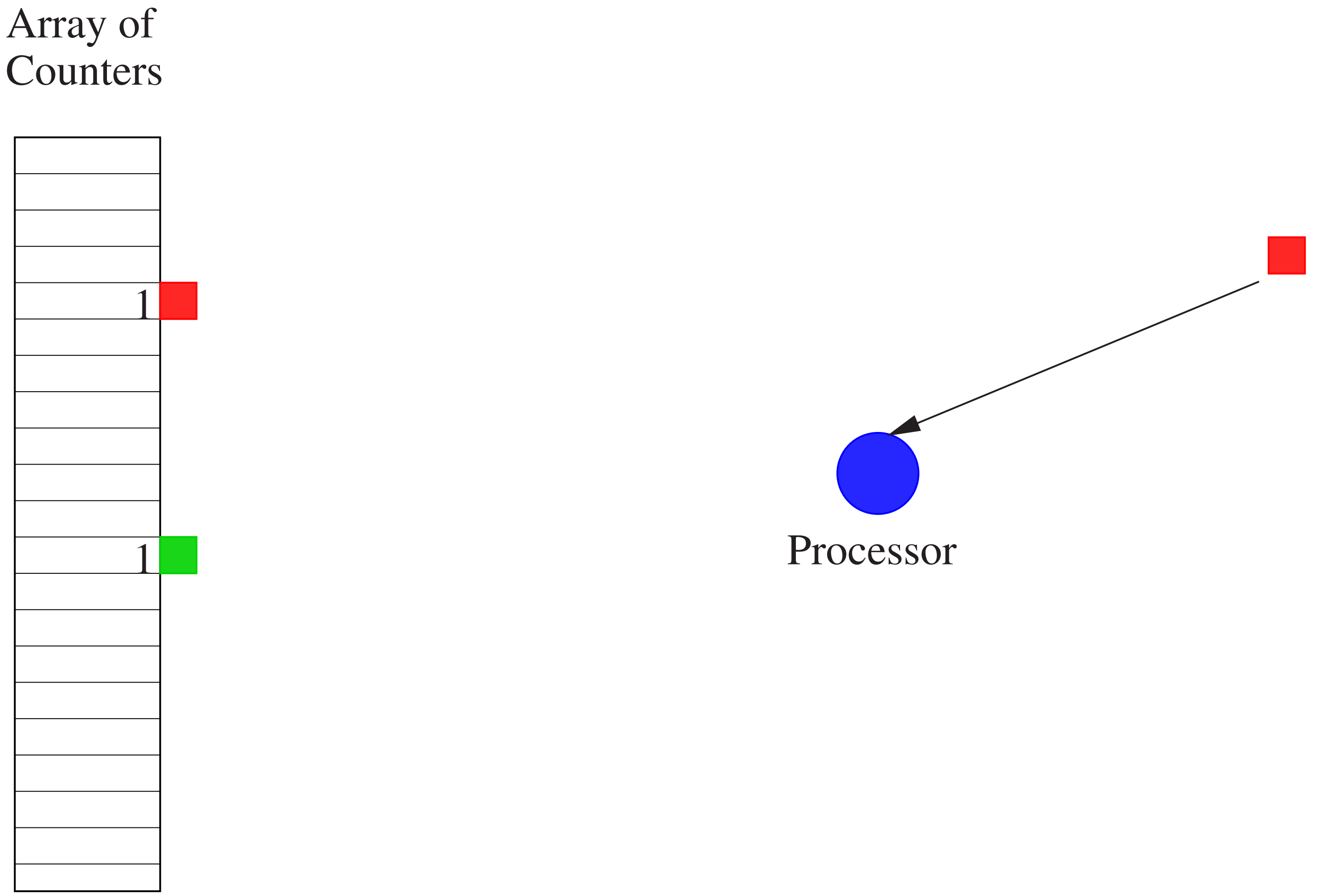
Counting Sketch: Array of counters



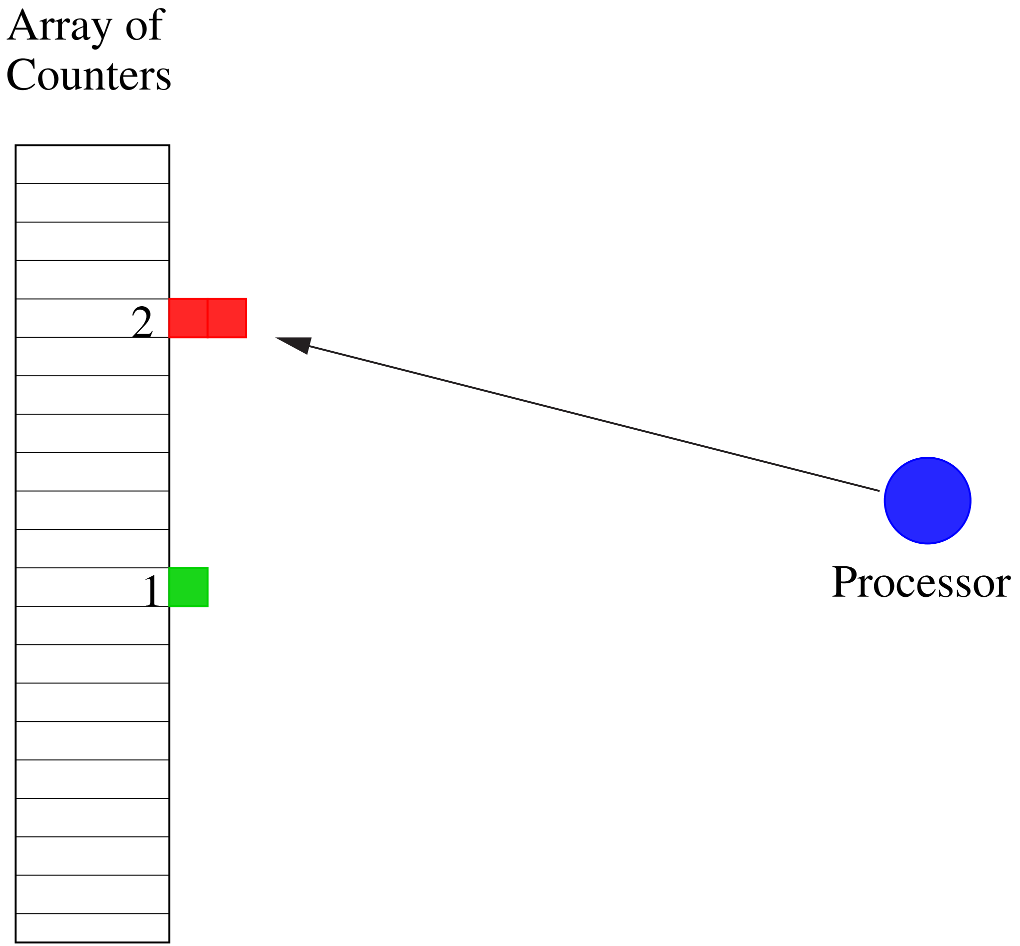
Counting Sketch: Array of counters



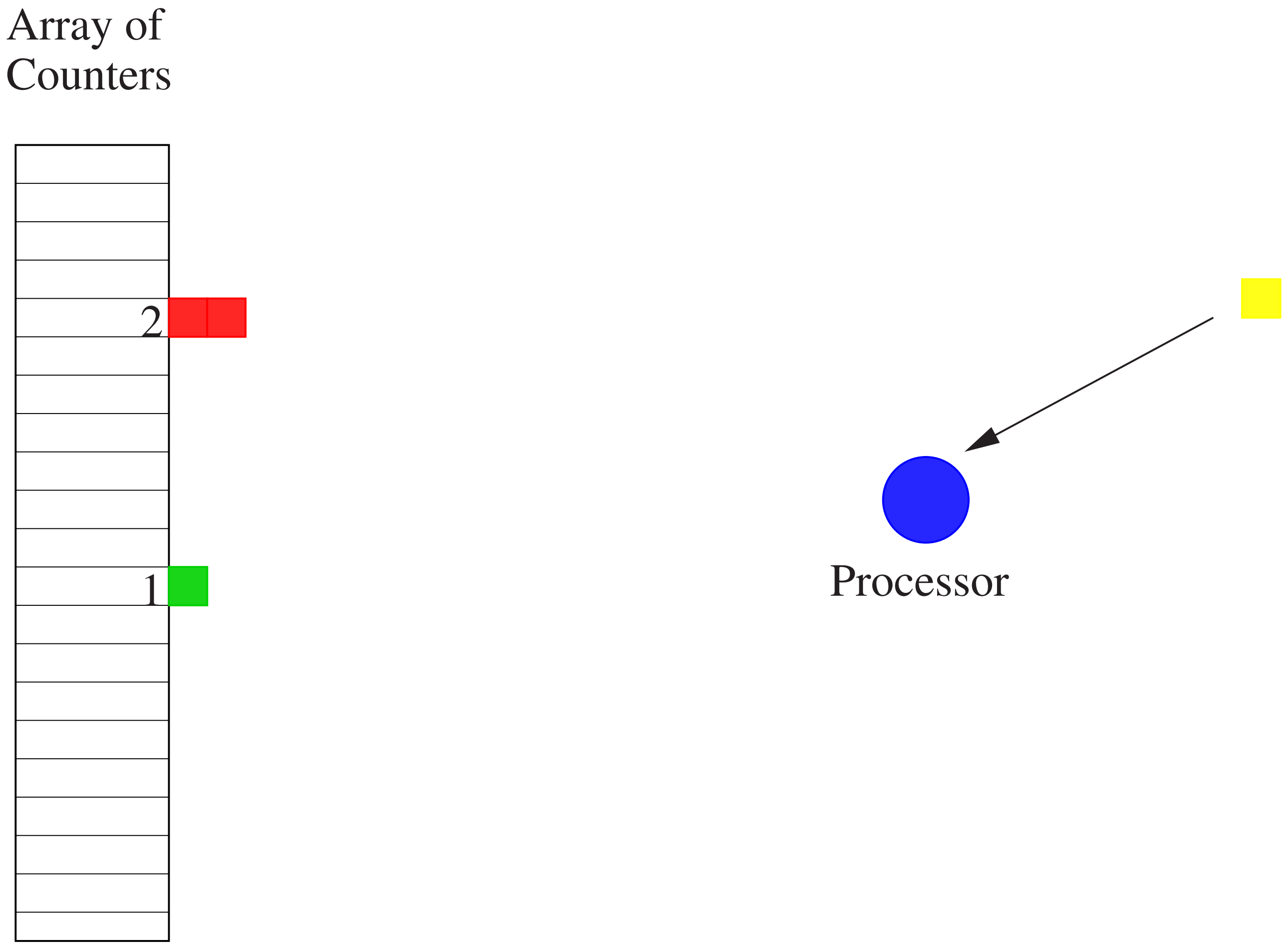
Counting Sketch: Array of counters



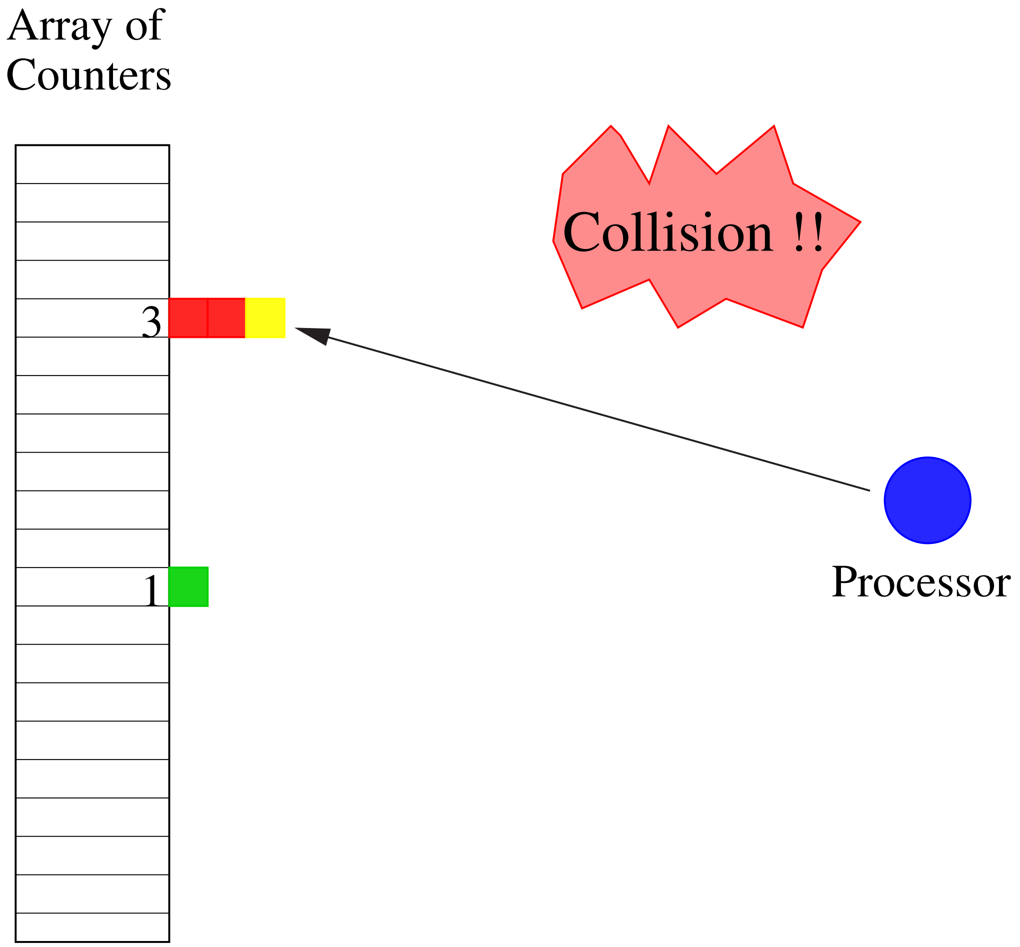
Counting Sketch: Array of counters



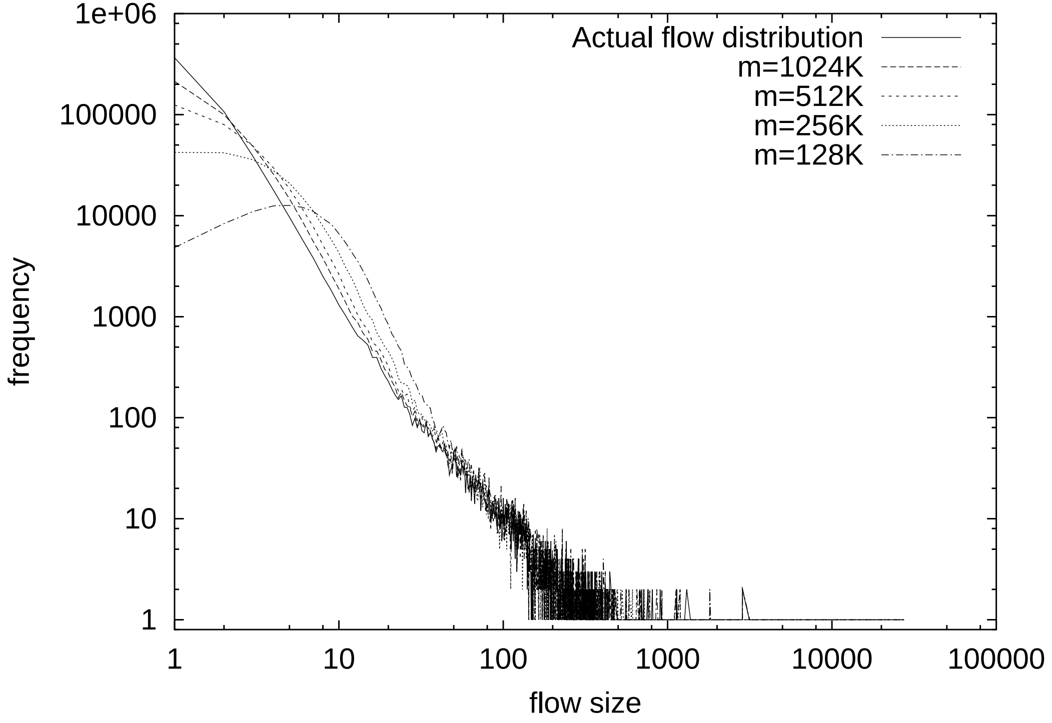
Counting Sketch: Array of counters



Counting Sketch: Array of counters



The shape of the "Counter Value Distribution"



The distribution of flow sizes and raw counter values (both x and y axes are in log-scale). m = *number of counters*.

Estimating n and n_1

- Let total number of counters be m.
- Let the number of value-0 counters be m_0
- Then $\hat{n} = m * ln(m/m_0)$ as discussed before
- Let the number of value-1 counters be y_1
- Then $\hat{n_1} = y_1 e^{\hat{n}/m}$
- Generalizing this process to estimate n_2, n_3, and the whole flow size distribution will not work
- Solution: joint estimation using Expectation Maximization

Estimating the entire distribution, ϕ, using EM

- Begin with a guess of the flow distribution, ϕ^{ini}.
- Based on this ϕ^{ini}, compute the various possible ways of "splitting" a particular counter value and the respective probabilities of such events.
- This allows us to compute a refined estimate of the flow distribution ϕ^{new}.
- Repeating this multiple times allows the estimate to converge to a *local maximum*.
- This is an instance of *Expectation maximization*.

Estimating the entire flow distribution — an example

- For example, a counter value of 3 could be caused by three events:
  - 3 = 3 (no hash collision);
  - 3 = 1 + 2 (a flow of size 1 colliding with a flow of size 2);
  - 3 = 1 + 1 + 1 (three flows of size 1 hashed to the same location)
- Suppose the respective probabilities of these three events are 0.5, 0.3, and 0.2 respectively, and there are 1000 counters with value 3.
- Then we estimate that 500, 300, and 200 counters split in the three above ways, respectively.
- So we credit 300 * 1 + 200 * 3 = 900 to n_1, the count of size 1 flows, and credit 300 and 500 to n_2 and n_3, respectively.

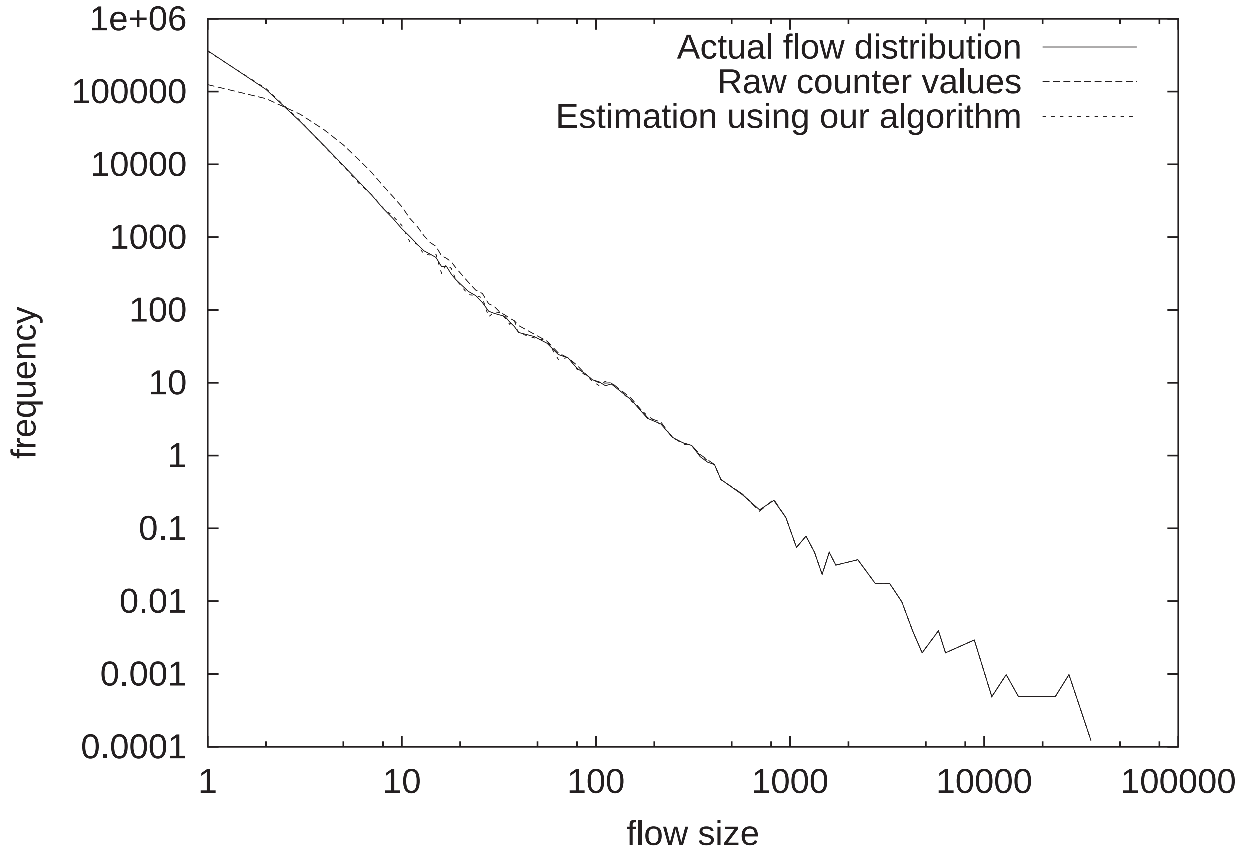
How to compute these probabilities

- Fix an arbitrary index ind. Let β be the event that f_1 flows of size s_1, f_2 flows of size s_2, ..., f_q flows of size s_q collide into slot ind, where $1 \leq s_1 < s_2 < \ldots < s_q \leq z$, let λ_i be n_i/m and λ be their total.
- Then, the a priori (i.e., before observing the value v at ind) probability that event β happens is

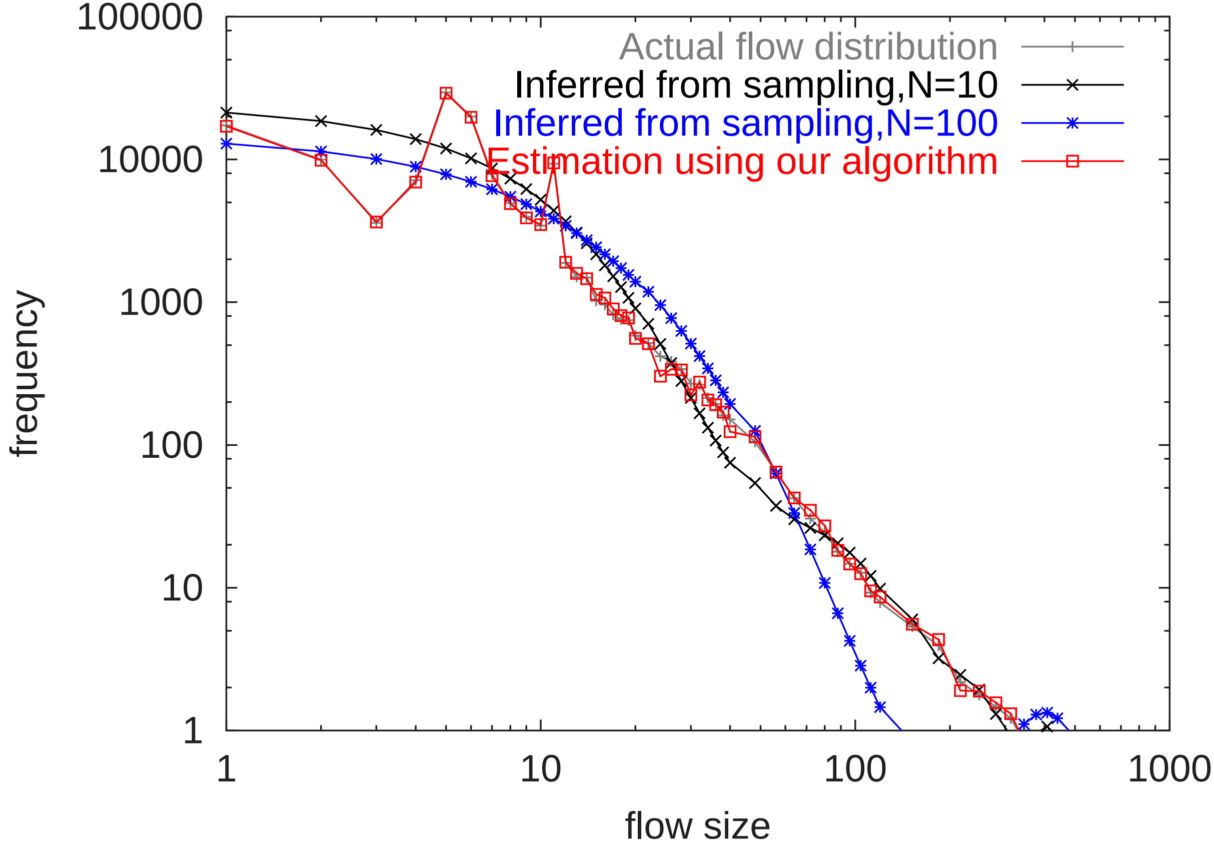
$$p(\beta|\phi, n) = e^{-\lambda} \prod_{i=1}^{q} \frac{\lambda_{s_i}^{f_i}}{f_i!}.$$

- Let Ω_v be the set of all collision patterns that add up to v. Then by Bayes' rule, $p(\beta|\phi, n, v) = \frac{p(\beta|\phi,n)}{\sum_{\alpha \in \Omega_v} p(\alpha|\phi,n)}$, where $p(\beta|\phi, n)$ and $p(\alpha|\phi, n)$ can be computed as above

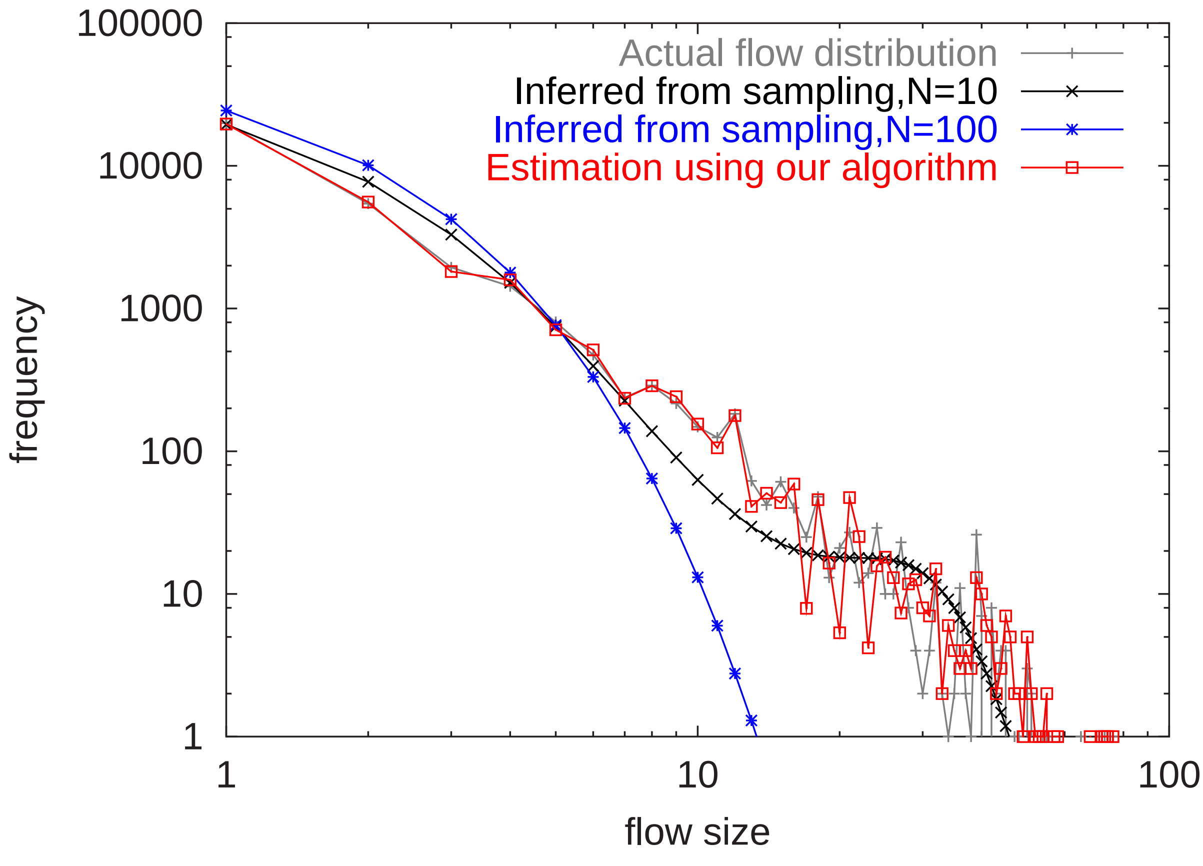
Evaluation — Before and after running the Estimation algorithm



Sampling vs. array of counters – Web traffic.



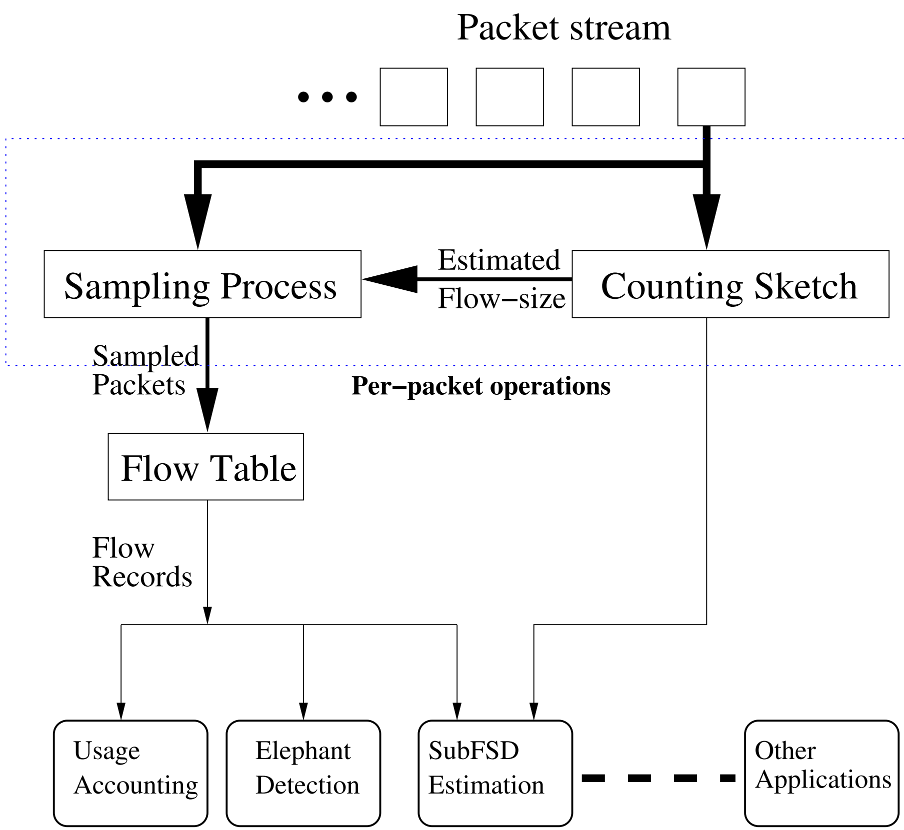
Sampling vs. array of counters – DNS traffic.



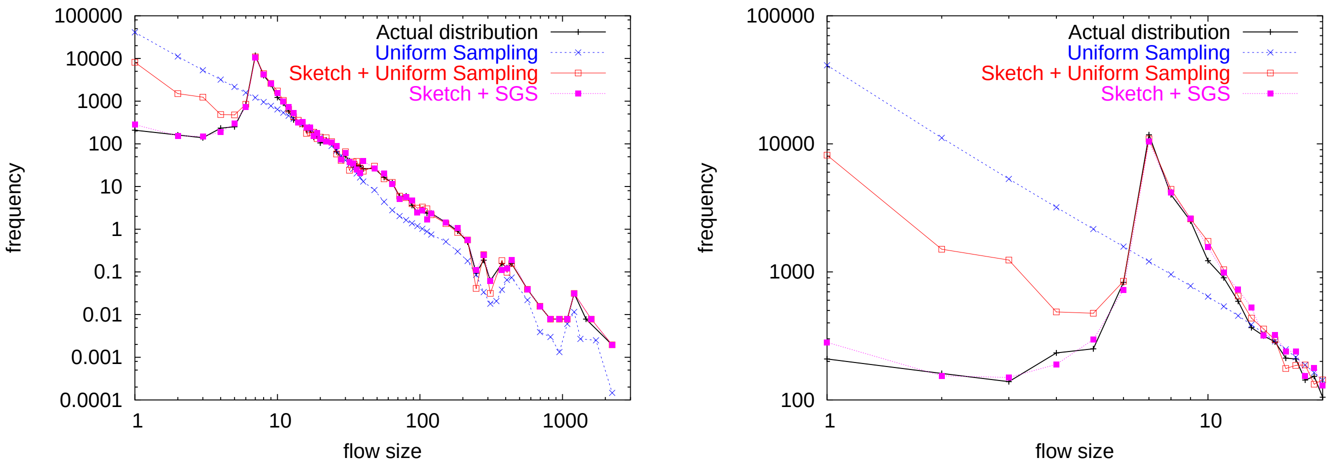
Extending the work to estimating subpopulation FSD [Kumar et al., 2005a]

- Motivation: there is often a need to estimate the FSD of a subpopulation (e.g., "what is FSD of all the DNS traffic").
- Definitions of subpopulation not known in advance and there can be a large number of potential subpopulation.
- Our scheme can estimate the FSD of any subpopulation defined after data collection.
- Main idea: perform both data streaming and sampling, and then correlate these two outputs (using EM).

Streaming-guided sampling [Kumar and Xu, 2006]



Estimating the Flow-size Distribution: Results



(a) Complete distribution. (b) Zoom in to show impact on small flows.

Figure 1: Estimates of FSD of https flows using various data sources.

A hardware primitive for counter management [Zhao et al., 2006b]

- Problem statement: To maintain a large array (say millions) of counters that need to be incremented (by 1) in an arbitrary fashion (i.e., $A[i_1]++$, $A[i_2]++$, ...)
- Increments may happen at very high speed (say one increment every 10ns) – has to use high-speed memory (SRAM)
- Values of some counters can be very large
- Fitting everything in an array of "long" (say 64-bit) SRAM counters can be expensive
- Possibly lack of locality in the index sequence (i.e., i_1, i_2, ...) – forget about caching

Motivations

- A key operation in many network data streaming algorithms is to "hash and increment"
- Routers may need to keep track of many different counts (say for different source/destination IP prefix pairs)
- To implement millions of token/leaky buckets on a router
- Extensible to other non-CS applications such as sewage management
- Our work is able to make 16 SRAM bits out of 1 (Alchemy of the 21st century)

Main Idea in Previous Approaches [Shah et al., 2002, Ramabhadran and Va

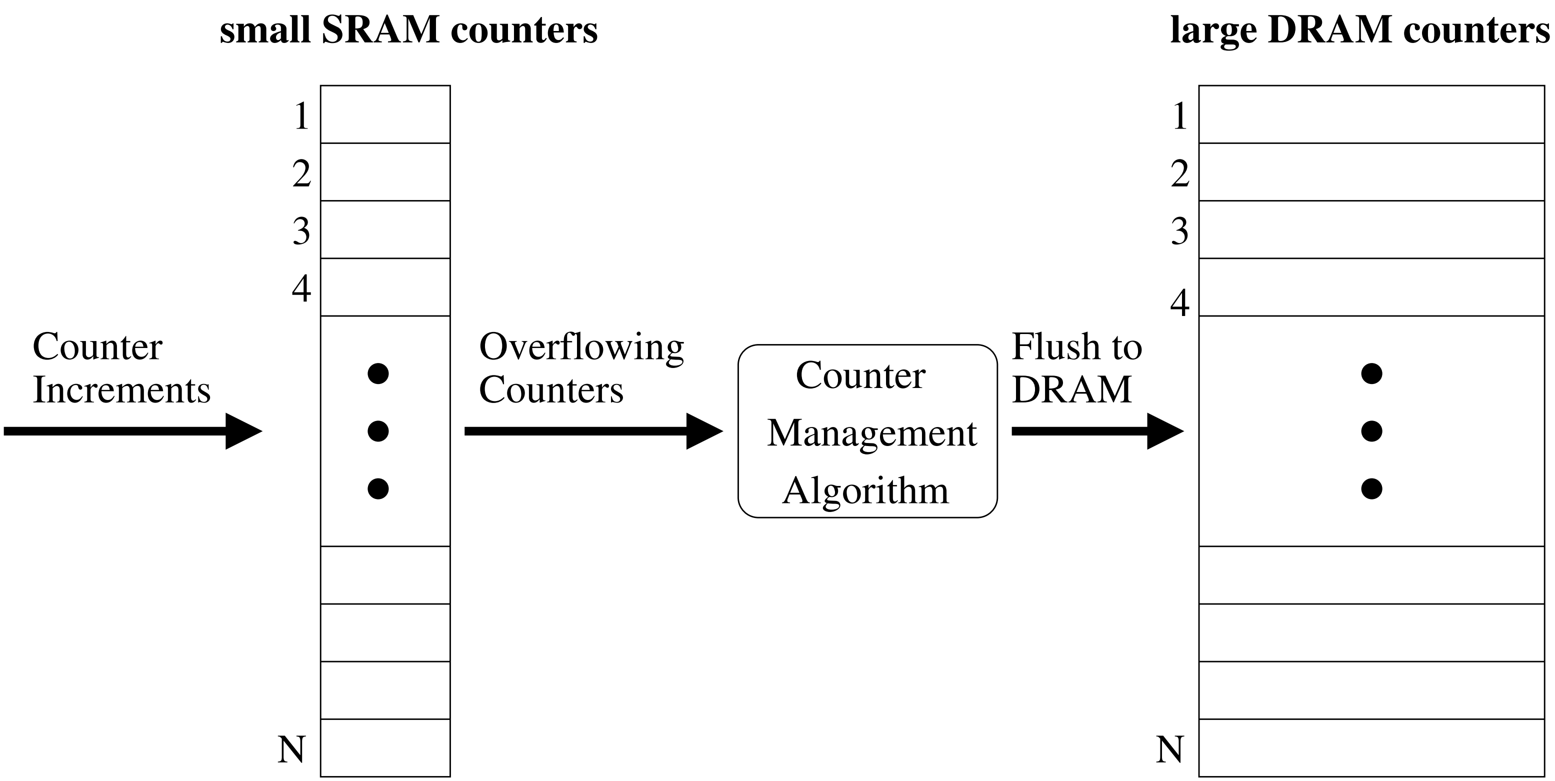


Figure 2: Hybrid SRAM/DRAM counter architecture

CMA used in [Shah et al., 2002]

- Implemented as a priority queue (fullest counter first)
- Need 28 = 8 + 20 bits per counter (when S/D is 12) – the theoretical minimum is 4
- Need pipelined hardware implementation of a heap.

CMA used in [Ramabhadran and Varghese, 2003]

- SRAM counters are tagged when they are at least half full (implemented as a bitmap)
- Scan the bitmap clockwise (for the next "1") to flush (half-full)$^+$ SRAM counters, and pipelined hierarchical data structure to "jump to the next 1" in O(1) time
- Maintain a small priority queue to preemptively flush the SRAM counters that rapidly become completely full
- 8 SRAM bits per counter for storage and 2 bits per counter for the bitmap control logic, when S/D is 12.

Our scheme

- Our scheme only needs 4 SRAM bits when S/D is 12.
- Flush only when an SRAM counter is "completely full" (e.g., when the SRAM counter value changes from 15 to 16 assuming 4-bit SRAM counters).
- Use a small (say hundreds of entries) SRAM FIFO buffer to hold the indices of counters to be flushed to DRAM
- Key innovation: a simple randomized algorithm to ensure that counters do not overflow in a burst large enough to overflow the FIFO buffer, with overwhelming probability
- Our scheme is provably space-optimal

The randomized algorithm

- Set the initial values of the SRAM counters to independent random variables uniformly distributed in $\{0, 1, 2, ..., 15\}$ (i.e., $A[i] := uniform\{0, 1, 2, ..., 15\}$).
- Set the initial value of the corresponding DRAM counter to the negative of the initial SRAM counter value (i.e., $B[i] := -A[i]$).
- Adversaries know our randomization scheme, but not the initial values of the SRAM counters
- We prove rigorously that a small FIFO queue can ensure that the queue overflows with very small probability

A numeric example

- One million 4-bit SRAM counters (512 KB) and 64-bit DRAM counters with SRAM/DRAM speed difference of 12
- 300 slots (≈ 1 KB) in the FIFO queue for storing indices to be flushed
- After 10^{12} counter increments in an arbitrary fashion (like 8 hours for monitoring 40M packets per second links)
- The probability of overflowing from the FIFO queue: less than 10^{-14} in the worst case (MTBF is about 100 billion years) – proven using minimax analysis and large deviation theory (including a new tail bound theorem)

Distributed coordinated data streaming – a new paradigm

- A network of streaming nodes
- Every node is both a producer and a consumer of data streams
- Every node exchanges data with neighbors, “streams” the data received, and passes it on further
- We applied this kind of data streaming to P2P [Kumar et al., 2005b] and sensor network query routing, and the RPI team has applied it to Ad-hoc networking routing.

Finding Global Icebergs over Distributed Data Sets [Zhao et al., 2006a]

- An **iceberg**: the item whose frequency count is greater than a certain threshold.
- A number of algorithms are proposed to find icebergs at a single node (i.e., local icebergs).
- In many real-life applications, data sets are physically distributed over a large number of nodes. It is often useful to find the icebergs over aggregate data across all the nodes (i.e., **global icebergs**).
- Global iceberg $\neq$ Local iceberg
- We study the problem of finding global icebergs over distributed nodes and propose two novel solutions.

Motivations: Some Example Applications

- Detection of distributed DoS attacks in a large-scale network
  - The IP address of the victim appears over many ingress points. It may not be a local iceberg at any ingress points since the attacking packets may come from a large number of hosts and Internet paths.
- Finding globally frequently accessed objects/URLs in CDNs (e.g., Akamai) to keep tabs on current "hot spots"
- Detection of system events which happen frequently across the network during a time interval
  - These events are often the indication of some anomalies. For example, finding DLLs which have been modified on a large number of hosts may help detect the spread of some unknown worms or spyware.

Problem statement

- A system or network that consists of N distributed nodes
- The data set S_i at node i contains a set of $\langle x, c_{x,i} \rangle$ pairs.
  - Assume each node has enough capacity to process incoming data stream. Hence each node generates a list of the arriving items and their exact frequency counts.
- The flat communication infrastructure, in which each node only needs to communicate with a central server.
- Objective: Find $\{x | \sum_{i=1}^{N} c_{x,i} \geq T\}$, where $c_{x,i}$ is the frequency count of the item x in the set S_i, with the minimal communication cost.

Our solutions and their impact

- Existing solutions can be viewed as "hard-decision codes" by finding and merging local icebergs
- We are the first to take the "soft-decision coding" approach to this problem: encoding the "potential" of an object to become a global iceberg, which can be decoded with overwhelming probability if indeed a global iceberg
- Equivalent to the minimax problem of "corrupted politician"
- We offered two solution approaches (sampling-based and bloom-filter-based)and discovered the beautiful mathematical structure underneath (discovered a new tail bound theory on the way)
- Sprint, Thomson, and IBM are all very interested in it

Direct Measurement of Traffic Matrices [Zhao et al., 2005a]

- Quantify the aggregate traffic volume for every origin–destination (OD) pair (or ingress and egress point) in a network.
- Traffic matrix has a number of applications in network management and monitoring such as
  - capacity planning: forecasting future network capacity requirements
  - traffic engineering: optimizing OSPF weights to minimize congestion
  - reliability analysis: predicting traffic volume of network links under planned or unexpected router/link failures

Previous Approaches

- Direct measurement [Feldmann et al., 2000]: record traffic flowing through at all ingress points and combine with routing data
  - storage space and processing power are limited: sampling
- Indirect inference such as [Vardi, 1996, Zhang et al., 2003]: use the following information to construct a highly **under-constrained linear inverse problem** $\mathrm{B} = \mathbf{A}\mathrm{X}$
  - SNMP link counts B (traffic volume on each link in a network)
  - routing matrix ($\mathbf{A}_{i,j} = \begin{cases} 1 & \text{if traffic of OD flow } j \text{ traverses link } i, \\ 0 & \text{otherwise.} \end{cases}$)

Data streaming at each ingress/egress node

- Maintain a bitmap (initialized to all 0's) in fast memory (SRAM)
- Upon each packet arrival, input the invariant packet content to a hash function; choose the bit by hashing result and set it to 1.
  - variant fields (e.g., TTL, CHECKSUM) are marked as 0's
  - adopt the equal sized bitmap and the same hash function
- No attempt to detect or resolve collisions caused by hashing
- Ship the bitmap to a central server at the end of a measurement epoch

How to Obtain the Traffic Matrix Element $TM_{i,j}$?

- Only need the bitmap B_i at node i and the bitmap B_j at node j for $TM_{i,j}$.
- Let T_i denote the set of packets hashed into B_i: $TM_{i,j} = |T_i \cap T_j|$.
  - Linear counting algorithm [Whang et al., 1990] estimates $|T_i|$ from B_i, i.e., $\widehat{|T_i|} = b \log \frac{b}{U}$ where b is the size of B_i and U is the number of "0"s in B_i.
  - $|T_i \cap T_j| = |T_i| + |T_j| - |T_i \cup T_j|$.
    * $|T_i|$ and $|T_j|$: estimate directly
    * $|T_i \cup T_j|$: infer from the bitwise-OR of B_i and B_j.

Some theoretical results

- Our estimator is almost unbiased and we derive its approximate variance

$$Var[\widehat{TM_{i,j}}] = b(2e^{t_{T_i \cap T_j}} + e^{t_{T_i \cup T_j}} - e^{t_{T_i}} - e^{t_{T_j}} - t_{T_i \cap T_j} - 1)$$

- Sampling is integrated into our streaming algorithm to reduce SRAM usage

$$Var[\widehat{TM_{i,j}}] = \frac{b}{p^2}\left((e^{\frac{Tp}{b} - \frac{Xp}{2b}} - e^{\frac{Xp}{2b}})^2 + e^{\frac{Xp}{b}} - \frac{Xp}{b} - 1\right) + \frac{X(1-p)}{p}$$

- The general forms of the estimator and variance for the intersection of $k \geq 2$ sets from the corresponding bitmaps is derived in [Zhao et al., 2005b].

Pros and Cons

- **Pros**
  - multiple times better than the sampling scheme given the same amount of data generated.
  - for estimating $TM_{i,j}$, only the bitmaps from nodes i and j are needed.
    - support submatrix estimation using minimal amount of information
    - allow for incremental deployment
- **Cons**
  - need some extra hardware addition (hardwired hash function and SRAM)
  - only support estimation in packets (not in bytes)

References

[Alon et al., 1999] Alon, N., Matias, Y., and Szegedy, M. (1999). The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–143.

[Chambers et al., 1976] Chambers, J. M., Mallows, C. L., and Stuck, B. W. (1976). A method for simulating stable random variables. *Journal of the American Statistical Association*, 71(354).

[Estan and Varghese, 2002] Estan, C. and Varghese, G. (2002). New Directions in Traffic Measurement and Accounting. In *Proc. ACM SIGCOMM*.

[Feldmann et al., 2000] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and F.True (2000). Deriving Traffic Demand for Operational IP Networks: Methodology and Experience. In *Proc. ACM SIGCOMM*.

[Indyk, 2006] Indyk, P. (2006). Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323.

[Kumar et al., 2004] Kumar, A., Sung, M., Xu, J., and Wang, J. (2004). Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc. ACM SIGMETRICS*.

[Kumar et al., 2005a] Kumar, A., Sung, M., Xu, J., and Zegura, E. (2005a). A data streaming algorithms for estimating subpopulation flow size distribution. In *Proc. ACM SIGMETRICS*.

[Kumar and Xu, 2006] Kumar, A. and Xu, J. (2006). Sketch guided sampling–using on-line estimates of flow size for adaptive data collection. In *Proc. IEEE INFOCOM*.

[Kumar et al., 2005b] Kumar, A., Xu, J., and Zegura, E. W. (2005b). Efficient and scalable query routing for unstructured peer-to-peer networks. In *Proc. of IEEE Infocom*, Miami, Florida, USA.

[Lall et al., 2006] Lall, A., Sekar, V., Ogihara, M., Xu, J., and Zhang, H. (2006). Data streaming algorithms for estimating entropy of network traffic. In *Proc. ACM SIGMETRICS*.

[Manku and Motwani, 2002] Manku, G. and Motwani, R. (2002). Approximate frequency counts over data streams. In *Proc. 28th International Conference on Very Large Data Bases (VLDB)*.

[Muthukrishnan,] Muthukrishnan, S. Data streams: algorithms and applications. available at http://athos.rutgers.edu/∼muthu/.

[Ramabhadran and Varghese, 2003] Ramabhadran, S. and Varghese, G. (2003). Efficient implementation of a statistics counter architecture. In *Proc. ACM SIGMETRICS*.

[Shah et al., 2002] Shah, D., Iyer, S., Prabhakar, B., and McKeown, N. (2002). Maintaining statistics counters in router line cards. In *IEEE Micro*.

[Vardi, 1996] Vardi, Y. (1996). Internet tomography: estimating source-destination traffic intensities from link data. *Journal of American Statistics Association*, pages 365–377.

[Whang et al., 1990] Whang, K., Vander-zanden, B., and Taylor, H. (1990). A linear-time probabilistic counting algorithm for database applications. *IEEE transaction of Database Systems*, pages 208–229.

[Zhang et al., 2003] Zhang, Y., Roughan, M., Lund, C., and Donoho, D. (2003). An information-theoretic approach to traffic matrix estimation. In *Proc. ACM SIGCOMM*.

[Zhao et al., 2007] Zhao, H., Lall, A., Ogihara, M., Spatscheck, O., Wang, J., and Xu, J. (2007). A data streaming algorithm for estimating entropies of od flows. In *IMC '07, Proceedings of the Internet Measurement Conference 2007*.

[Zhao et al., 2005a] Zhao, Q., Kumar, A., Wang, J., and Xu, J. (2005a). Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices. In *Proc. ACM SIGMETRICS*.

[Zhao et al., 2005b] Zhao, Q., Kumar, A., and Xu, J. (2005b). Joint data streaming and sampling techniques for accurate identification of super sources/destinations. In *Proc. ACM IMC*.

[Zhao et al., 2006a] Zhao, Q., Ogihara, M., Wang, H., and Xu, J. (2006a). Finding global icebergs over distributed data sets. In *ACM PODS*.

[Zhao et al., 2006b] Zhao, Q., Xu, J., and Liu, Z. (2006b). Design of a novel statistics counter architecture with optimal space and time efficiency. In *Proc. ACM SIGMETRICS*.