Data Streams: Algorithms and Applications

by S. Muthukrishnan

Presentation by Ramesh Sridharan and Matthew Johnson

1 So what is a streaming algorithm?

- From Wikipedia: "A streaming algorithm is a method of managing a flow of data by examining arriving items once and then discarding them. The benefit of a streaming algorithm is that it can be used to manage data that is continuously generated and and where the total volume is extremely large, even if the volume is too large for memory."
- The relevant problems often involve computing a relatively simple function of a large sequential stream under severe space and sometimes time constraints. According to the monograph, relevant applications include IP network traffic analysis, mining text message streams, and tackling huge observation streams such as new particle physics experiments with 40TB/s.

2 Some Illustrative Examples [Sec. 1]

Finding Missing Numbers

Let π be a permutation of $\{1, \ldots, n\}$, and π_{-1} be π with one arbitrary element missing. Paul shows Carole $\pi_{-1}[i]$ in increasing i, and Carole is tasked with determining the missing integer. But she can only use $O(\log n)$ bits of memory!

$$s = \frac{n(n+1)}{2} - \sum_{j \leq i} \pi_{-1}[j]$$

What if n is unknown in advance? Just keep track of the maximum.

What if Carole must solve the puzzle with two elements missing, i.e. she observes π_{-2} and must find both missing elements? Can it be generalized?

One solution would be to keep both s as above and ss as

$$ss = \frac{n(n+1)(2n+1)}{6} - \sum_{j \leq i} (\pi_{-2}[j])^2$$

.

Such formulae continue for arbitrary power sums.

Fishing

Paul goes fishing, and in his lake there is a set of possible fish $U = \{1, \ldots, u\}$. Paul catches one fish at a time, recording his catches as the sequence $(a_t)_{t \in \mathbb{N}}$ with $a_t \in U$. $c_t[j]$ is the number of times he catches species j up to time t.

We might say species j is *rare* at time t if it appears precisely once in his catch up to time t, and the *rarity* of his current catch might be the proportion of all fish that are rare in his catch:

$$\rho[t] = \frac{|\{j | c_t[j] = 1\}|}{u}$$

Paul wants to maintain a calculation of the rarity of his catch using only the few bits he brought along with him: $o(u)$, preferably $O(1)$.

It can't be done, by an "information" lower-bound proof. But it can be approximated in a straightforward way.

Suppose we consider a new definition of rarity, where rarity is defined relative to Paul's catch-so-far instead of the entire population:

$$\gamma[t] = \frac{|\{j|c_t[j] = 1\}|}{|\{j|c_t[j] \neq 0\}|}$$

The problem is now harder since more information about the catch is needed, but an approximation can be built using random min-wise independent hash functions.

With a probabilistic description of fish samples, this problem can be related to Good-Turing estimators for missing mass.

Pointer and Chaser

In this puzzle, we have $n+1$ starting positions, and n destinations. Each starting position points to a single destination; by the pigeonhole principle, there must be at least one destination that is pointed to by multiple starting positions. We want to find this duplicate position.

Aside from the obvious iterative solution, which requires $O(\log n)$ bits of space and $O(n^2)$ time, we can apply a more clever multiple-pass approach. Here, we loop over all the starting positions. In the first pass, we maintain two counters, a "low" counter and a "high" counter. The low counter is incremented whenever the position points to a destination with value less than $n/2$, and the high counter is incremented whenever the position points to a destination with value greater than $n/2$. After we complete one pass, whichever of the two counters is bigger will tell us where in the destination set our duplicate lies. We can then repeat the process, over the appropriate subsection. We thus only require $O(\log n)$ space and $O(n \log n)$ time.

A final solution is to treat the starting positions and the destinations as the same set. In this case, the problem reduces to finding the loop point of a cyclic linked list that starts at node $n+1$. This can be solved by maintaining two pointers. At even-numbered time steps, we advance the first pointer, or "slow" pointer by one node, and at odd-numbered time steps we advance the second pointer, or "fast" pointer, by two nodes. After each time step, we check if the two pointers point to the same place; if they do, we have found our cycle. This takes $O(\log n)$ bits of space as before, but now only requires linear time. However, it also requires constant-time random access over the input stream, which may not always be available in practice.

3 Formalism [Sec. 4]

We consider input streams, which represent underlying shorter signals. We will use $a_1, a_2, \ldots, a_t, \ldots$ to represent the input stream, where a_t arrives at time t. This stream describes some underlying signal, $A[i]$ for $i \in [1, N]$ for some dimensionality N, which we would like to query. There are three typical models used:

- **Time Series:** In this model, $a_t = A[t]$. This is the simplest model, and we will not see it often in this paper.
- **Cash Register:** In this model, each $a_t = (j, I_t)$, and at time t, we update A_t, the value of the signal at time t, as $A_t[j] = A_{t-1}[j] + I_t$. In this model, we restrict I_t to be **nonnegative**. We will often see the case where $I_t = 1 \forall t$. In this case, the signal contains counts of elements j that we see in our signal.
- **Turnstile:** In this model, we again receive our signal $a_t = (j, I_t)$ and update $A[j]$ accordingly, but now we do not restrict the values of I_t. In the **strict turnstile** model, $A[j] \geq 0 \forall j$ at all times.

4 Basic Mathematical Techniques [Sec. 5]

This section overviews the mathematical and probabilistic arguments that are common in data streaming problems. It employs many examples, most of which are summarized below. There are two main categories that the author has identified: *sampling*, which refers to problems where only a polylog size subset of the input is of interest, and *random projections*, which make use of randomly-chosen summary data to estimate quantities of interest with probabilistic performance guarantees.

4.1 Sampling

Heavy Hitters

Consider the cash register model. Estimating $\max_i A[i]$ is impossible in $o(N)$ space, and estimating the k most frequent items is at least as hard. Instead, researchers consider a slightly relaxed problem of finding the *heavy hitters*: those items where the multiplicity exceeds the fraction ϕ of the total size of the data stream. Note that if we did not have such a tight space constraint, a heap structure would solve our problem, but in fact we can show that any algorithm that guarantees to find all and only items i such that $A[i] > (1/k+1)\,||A||_1$ must store $\Omega(N)$ bits.

Thus, we allow some approximation, and define the (ϕ, ϵ)-heavy hitters as all i such that $A[i] \geq \phi\,||A||_1$ with no "ϵ-bad false positives", i.e. no i such that $A[i] \leq (\phi - \epsilon)\,||A||_1$.

We can design an algorithm with no false negatives but bad false positives by keeping a set of K counters (where K does not depend on N but only ϕ). Each counter corresponds to an element recently seen in the input stream, so the size of K grows as we receive the stream. When $|K| > 1/\phi$, decrement each of the counts by 1 and eliminate the ones with count 0. There is a way to augment this algorithm with $O(1/\epsilon)$ space usage to eliminate the bad false positives.

Distinct Sampling

Here, we define the "inverse signal":

$$A^{-1}[i] = \frac{|\{j | A[j] = i, i \neq 0\}|}{|\{j | A[j] \neq 0\}|}$$

It is the proportion of values i such that $A[j] = i$. In the cash register model with $I_i = 1$, it corresponds to the proportion of values (js) that occurred i times.

For an error bound ϵ and probability of error δ, we can approximate A^{-1} with $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ samples. The key to this algorithm is that the samples are chosen in such a way that each distinct item in the input stream j is equally likely to be sampled. We can then approximate $A^{-1}[i]$ by the fraction of samples that occurred i times within the sample.

4.2 Random Projections

Moments estimation

Here, we want to estimate the kth moment of a stream: $F_k \sum_i A[i]^k$. This is useful in many practical settings, as we will see over the next few weeks. In this section, we focus on F_2.

We consider the random vectors $\mathbf{X}_{ij}[i]$ of length N whose elements are ± 1 and fourwise independent. We also define $X_{ij} = \langle A, \mathbf{X}_{ij}\rangle = \sum_\ell A[\ell]\mathbf{X}_{ij}[\ell]$.

We can show $\mathbb{E}[X_{ij}^2] = F_2$ by considering the square of the sum above, and noting that in expectation, the cross terms between $\mathbf{X}_{ij}$ are 0. We can also show that $\text{var}(X_{ij}^2) = 2F_2^2$ using a similar approach for X_{ij}^4, the second moment of the random variable X_{ij}^2.

To obtain an approximation that lies within $(1 \pm \epsilon)F_2$ with probability greater than $(1 - \delta)$, we consider i in the range $\{1, \ldots, \log\frac{1}{\epsilon^2}\}$, and j in the range $\{1, \ldots, \log\frac{1}{\delta}\}$.

Count-min sketch

We often want to keep track of $A[i]$ for all i, but this violates our space constraints. So, instead of maintaining $A[i]$ for all i, we instead maintain a 2-dimensional $d \times w$ array called count, where $w = \lceil\frac{e}{\epsilon}\rceil$ and $d = \lceil\ln\frac{1}{\delta}\rceil$. Associated with the array are d hash functions $h_1, \ldots, h_d : \{1, \ldots, N\} \to \{1, \ldots, w\}$. When we receive an update $a_i = (j, I_i)$, for each hash function h_k, we update $\text{count}[k, h_k(j)]$ to be $\text{count}[k, h_k(j)] + I_i$; that is, each cell maintains the cumulative sum of all updates whose index hashes to that value.

This allows us to efficiently solve the point-estimation problem, i.e. find $A[i]$ for an arbitrary i. Our estimate is

$$\hat{A}[i] = \min_j \text{count}[j, h_j(i)]$$

This is (certainly) bounded from below by $A[i]$ and (with probability at least $1 - \delta$) from above by $A[i] + \epsilon ||A||_1$.

Note that count$[j, h_j(i)]$ has not only the I_ks corresponding to index i, but also the I_ks corresponding to any other index that hashes to the same value. So, $\hat{A}[i]$ is bounded from below because of these "extra values." The bound from above comes from applying the Markov inequality to the probability $\mathbb{P}(\hat{A}[i] \leq A[i] + \epsilon ||A||_1)$. This is equivalent to $\mathbb{P}(\text{count}[j, h_j(i)] \leq A[i] + \epsilon ||A||_1 \, \forall j)$. This is equivalent to the probability that the sum of the "extra values" is less than $\epsilon ||A||_1$. The expectation of this "extra weight" is $||A||_1 / w$, and since they are pairwise independent, we can obtain a bound by multiplying their probabilities. Using the Markov inequality then gives the desired result.

Note that many of the problems expressed in earlier sections can be solved using this technique.

Estimating Number of Distinct Elements

The problem is to estimate $D = |\{i | A[i] \neq 0\}|$. If $A[i]$ is the number of occurrences of i in the stream, D is the number of distinct items. More generally, D is the support of $A[i]$.

One way of estimating D in the cash register model keeps a bit vector c of length $\log_2 N$ and uses a hash function $f : [1, N] \to \{1, 2, \ldots, \log_2 N\}$ such that $\mathbb{P}[f(i) = j] = 2^{-j}$ and any update j to item i sets $c[f(i)]$ to 1. An unbiased estimate of the number of distinct items is given by $2^{k(c)}$, where $k(c)$ is the lowest index j such that $c[j] = 0$. Intuitively, if the probability that any item is mapped into the counter at index j is 2^{-j}, then if there are D distinct items, we expect $D/2$ of them to be mapped to $c[1]$, $D/4$ to be mapped to $c[2]$, etc. However, that relies on the existence of a fully random hash function, and so it has been extended to allow a hash function that can be stored in $O((\frac{1}{\epsilon^2} \log \log m + \log m \log(1/\epsilon)) \log(1/\delta))$. For the turnstile model, the methods for estimating D uses L_p-sum estimation for small p.