7
The Data Streaming Model

7.1 The Model

Today's lecture will be about a slightly different computational model called the *data streaming* model. In this model you see elements going past in a "stream", and you have very little space to store things. For example, you might be running a program on an Internet router, the elements might be IP Addresses, and you have limited space. You certainly don't have space to store all the elements in the stream. The question is: which functions of the input stream can you compute with what amount of time and space? (For this lecture, we will focus on space, but similar questions can be asked for update times.)

We will denote the stream elements by

$$a_1, a_2, a_3, \ldots, a_t, \ldots$$

We assume each stream element is from alphabet U and takes b bits to represent. For example, the elements might be 32-bit integers IP Addresses. We imagine we are given some function, and we want to compute it continually, on every prefix of the stream. Let us denote $a_{[1:t]} = \langle a_1, a_2, \ldots, a_t \rangle$.

7.1.1 Examples

Let us consider some examples. Suppose we have seen the integers

$$3, 1, 17, 4, -9, 32, 101, 3, -722, 3, 900, 4, 32, \ldots \qquad (\diamond)$$

- Computing the sum of all the integers seen so far? $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. We want the outputs to be

  $$3, 4, 21, 25, 16, 48, 149, 152, -570, -567, 333, 337, 369, \ldots$$

  If we have seen T numbers so far, the sum is at most $T2^b$ and hence needs at most $O(b + \log T)$ space. So we can just keep

a counter, and when a new element comes in, we add it to the counter.

- How about the maximum of the elements so far? $F(a_{[1:t]}) = \max_{i=1}^{t} a_i$. Even easier. The outputs are:

$$3, 1, 17, 17, 17, 32, 101, 101, 101, 101, 900, 900, 900$$

  We just need to store b bits.

- The median? The outputs on the various prefixes of ($\diamond$) now are

$$3, 1, 3, 3, 3, 3, 4, 3, \ldots$$

  And doing this will small space is a lot more tricky.

- ("distinct elements") Or the number of distinct numbers seen so far? You'd want to output:

$$1, 2, 3, 4, 5, 6, 7, 7, 8, 8, 9, 9, 9 \ldots$$

- ("heavy hitters") Or the elements that have appeared most often so far? Hmm...

You can imagine the applications of the data-stream model. An Internet router might see a lot of packets whiz by, and may want to figure out which data connections are using the most space? Or how many different connections have been initiated since midnight? Or the median (or the 90^{th} percentile) of the file sizes that have been transferred. Which IP connections are "elephants" (say the ones that have used more than 0.01% of your bandwidth)? Even if you are not working at "line speed", but just looking over the server logs, you may not want to spend too much time to find out the answers, you may just want to read over the file in one quick pass and come up with an answer. Such an algorithm might also be cache-friendly. But how to do this?

Such a router might see tens of millions of packets per second.

Two of the recurring themes in this chapter will be:

- ***Approximate solutions***: in several cases, it will be impossible to compute the function exactly using small space. Hence we'll explore the trade-offs between approximation and space.
- ***Hashing***: this will be a very powerful technique.

7.2 *Sampling vs. Hashing*

It is natural that we use sampling for some of these problems: after all, we want to whittle down the amount of data to some manageable

size. But sometimes you should be careful about how you sample (and why hashing may be good). Here's an example from the MMDS book.

Suppose you want to figure out the number of "uniques", i.e., *the elements that occur exactly once*. One way is: pick 10% of the stream *by picking each element of the stream independently at random with probability* 0.1, look at the number of uniques in the sample, and scale up the answer by 10. But this will mis-calcluate the answer. To see why, suppose the stream of length n has $n/2$ distinct elements that appear just once (the "uniques"), and $n/4$ more distinct elements that appear exactly twice (the "doubles"). So the correct number of uniques is $n/2$.

The sampled stream has expected length $\frac{1}{10}n$. So we sample a unique element with probability 0.1. But we also see a double element once if we sample either of its copies not not both, which happens with probability:

$$\begin{aligned}\Pr[\text{copy 1 sampled but not 2}] + \Pr[\text{copy 2 sampled but not 1}] \\ = (0.1)(0.9) + (0.9)(0.1) = 0.18.\end{aligned}$$

This is almost twice as often as the uniques! It means that now we expect to see $0.1 \times n/2 + (0.18) \times n/4 = \frac{n}{10} \cdot \frac{19}{20}$ unique elements in the sample. And our estimate will be that we have $\frac{19}{20}n$ unique elements in the stream, which is very wrong! In fact, by a concentration bound, you can show that you are incorrect not just in expectation, but also with very high probability (as n gets large).

The problem, of course, was that we were making *independent* sampling decisions for each element of the stream. What we should have done is to make sure that if an element was sampled, all copies of it were sampled too. And one way of doing this: pick a hash function that maps the universe to the range $[10] = \{0, 1, \ldots, 9\}$. And take the elements that map to 0, say, as part of your sample. Now, if the hash function is (at least) 1-wise independent (i.e., each value in the range is equally likely), then we'll get a 10% sample of the stream in a way that maintains the fraction of duplicates. *Everything is in expectation*, of course, and the variance of this estimator gets higher, so we have to work around that.

7.3 Streams as Vectors, and Additions/Deletions

An important abstraction will be to view the stream as a vector (in high dimensional space). Since each element in the stream is an element of the universe U, you can imagine the stream at time t as a

vector $\mathbf{x}^t \in \mathbb{Z}^{|U|}$. Here

$$\mathbf{x}^t = (x_1^t, x_2^t, \ldots, x_{|U|}^t)$$

and x_i^t is the number of times the i^{th} element in U has been seen until time t. (Hence, $x_i^0 = 0$ for all $i \in U$.) When the next element comes in and it is element j, we increment x_j by 1.

This brings us a extension of the model: we could have another model where each element of the stream is either a new element, or an old element departing. Formally, each time we get an *update* a_t, it looks like $(\texttt{add}, e)$ or $(\texttt{del}, e)$. We usually assume that for each element, the number of deletes we see for it is at most the number of adds we see — the running counts of each element is non-negative.

In data stream jargon, the addition-only model is called the *cash-register* model, whereas the model with both additions and deletions is called the *turnstile model*. I will not use this jargon.

As an example, suppose $U = \{A, B, C\}$ and the stream looked like:

$$(\texttt{add}, A), (\texttt{add}, B), (\texttt{add}, A), (\texttt{del}, B), (\texttt{del}, A), (\texttt{add}, C), \ldots$$

then the vector $x \in \mathbb{Z}^{|U|}$ evolves as follows:

$$(1,0,0), (1,1,0), (2,1,0), (2,0,0), (1,0,0), (1,0,1), \ldots$$

This vector notation allows us to formulate some of the problems more easily:

1. The question of "heavy hitters" is to estimate the "large" entries in the vector $\mathbf{x}$.
2. The total number of elements currently in the system is just $\|\mathbf{x}\| := \sum_{i=1}^{|U|} x_i$. (This is easy.)
3. The number of distinct elements is the number of non-zero entries in $\mathbf{x}$.
4. We might want to estimate the norms $\|x\|_2, \|x\|_p$ of the vector $\mathbf{x}$.

Let's consider the (non-trivial) problems one by one.

7.4 *Heavy Hitters*

There are many ways of formalizing the heavy-hitters problem. Here is one that is fairly useful. At any time, the ε-heavy-hitters are the indices i such that $x_i > \varepsilon\|\mathbf{x}\|_1$. Instead of asking to output the set of exactly those elements that are heavy-hitters, let us work with the following problem instead.

Problem Count-Query: At any time t, given an index i, output the value of x_i^t with an error of at most $\varepsilon\|\mathbf{x}^t\|_1$. I.e., output an estimate

$$\widetilde{x_i^t} \in x_i^t \pm \varepsilon\|\mathbf{x}^t\|_1.$$

Given an algorithm for Count-Query, if i is a heavy-hitter it will have $x_i^t > \varepsilon \|\mathbf{x}^t\|_1$, and hence the estimate $\widetilde{x}_i^t$ will be strictly positive. (Of course saying i is a heavy-hitter any time $\widetilde{x}_i^t > 0$ is dangerous, since might give us false positives.) But this is a step in the right direction.

So how should we solve the Count-Query problem? Would sampling work? Suppose we want to find the ε-heavy-hitters: should we just sample some small fraction of the elements and hope to see all the heavy hitters in there? This is not clear how to maintain a small sample size when we allow the stream to contain deletions. Imagine that N copies of a arrive, then they all depart, then we get a stream of length $\sqrt{N}$ containing just b. The only heavy hitter at the end is b. But unless we sample each element with probability more than $1/\sqrt{N}$, we don't expect to see any b's. Here we will see how to get away with storing only $\text{poly}(1/\varepsilon)$ elements.

7.4.1 *A Hashing-Based Solution to Count-Query*

We're going to be using hashing for this approach, simple and effective. We'll worry about what properties we need for our hash functions later, for now assume we have a hash function $h : U \to [M] = \{0, 1, \ldots, M-1\}$ for some suitably large integer M. Maintain an array $A[1 \ldots M]$ capable of storing non-negative integers.

This algorithm is called the CountMin sketch, and is due to Graham Cormode and S. Muthukrishnan.

Algorithm 6: CountMin-Basic

```
6.1 foreach update a_t do
6.2     if a_t == (add, i) then
6.3         A[h(i)]++
6.4     if a_t == (del, i) then
6.5         A[h(i)]--
```

This was the update procedure. And our estimate for x_i^t is

$$\widetilde{x}_i^t := A(h(i)).$$

In words, we look at the location $h(i)$ where i gets mapped using the hash function h, and look at the count $A[h(i)]$ stored at that location. What does it contain? It contains the current count x_i for element i for sure. But added to it is the current count for any other element that gets mapped to that location. In math:

$$A(h(j)) = \sum_{j \in U} x_j^t \cdot \mathbf{1}(h(j) = h(i)),$$

where $\mathbf{1}(\text{some condition})$ is a function that evaluates to 1 when the condition in the parentheses is true, and 0 if it is false. We can rewrite

this as

$$A(h(e)) = x_i^t + \sum_{j \neq i} x_j^t \cdot \mathbf{1}(h(j) = h(i)), \tag{7.1}$$

or using the definition of the estimate, the error is

$$\widetilde{x}_i^t - x_i^t = \sum_{j \neq i} x_j^t \cdot \mathbf{1}(h(j) = h(i)). \tag{7.2}$$

It is too much to hope that no other elements $j \neq i$ hashed to location $h(i)$ and the error evaluates to zero. What's the expected error? Now we need to assume something good about the hash functions. Assume that the hash function h is a random draw from a universal family. Recall the definition of universal:

Definition 7.1. A family H of hash functions from $U \to [M]$ is *universal* if for any pair of distinct keys $x_1, x_2 \in U$ with $x_1 \neq x_2$,

$$\Pr[h(x_1) = h(x_2)] \leq \frac{1}{M}.$$

Recall: we gave a matrix-based construction where each hash function in the family used $(\lg M) \cdot (\lg |U|)$ bits to specify.

Good. So we drew a hash function h from this universal hash family H, and we used it to map elements to locations $\{0, 1, \ldots, M-1\}$. What is its expected error? Taking expectations in (7.2),

$$\begin{aligned} E\left[\sum_{j \neq i} x_i^t \cdot \mathbf{1}(h(j) = h(i))\right] &= \sum_{j \neq i} x_i^t \cdot E\left[\mathbf{1}(h(j) = h(e))\right] && (7.3) \\ &= \sum_{j \neq i} x_i^t \cdot \Pr[h(j) = h(i)] \\ &\leq \sum_{j \neq i} x_i^t \cdot (1/M) && (7.4) \\ &= \frac{\|\mathbf{x}^t\|_1 - x_i^t}{M} \leq \frac{\|\mathbf{x}^t\|_1}{M}. \end{aligned}$$

We used linearity of expectations in equality (7.3). To get (7.4) from the previous line, we used the definition of a universal hash family.

That's pretty awesome. (But perhaps not so surprising, once you think about it.) If we just hash the vector down into a smaller vector of size $M = 1/\varepsilon$ then we get expected error $\varepsilon \|\mathbf{x}^t\|_1$, which sounds great. Actually, we'd like to do better—instead of just low *expected error*, we'd like to get low error with high probability. So let's see how to improve things.

7.4.2 *Amplification of the Success Probability*

Any ideas how to *amplify* the probability that we are close to the expectation? Very often independent repetition is a great idea.

Let us pick m hash functions $h_1, h_2, \ldots, h_\ell$ *independently* from the universal hash family H. Each $h_i : U \to \{0, 1, \ldots, M-1\}$. We now

If we use the matrix-based hash function construction given in the previous lecture, this means the $(\lg M) \cdot (\lg |U|)$-bit matrices for each of the ℓ hash functions must be filled with independent random bits.

also have ℓ arrays $A_1, A_2, \ldots, A_\ell$, one for each hash function. The algorithm now just uses the k^{th} hash function to choose a location in the k^{th} array, and increments or decrements the same as before.

Algorithm 7: CountMin

```
7.1 foreach update a_t do
7.2     foreach table k = 1...ℓ do
7.3         if a_t == (add, i) then
7.4             A_k[h(i)] ++
7.5         if a_t == (del, i) then
7.6             A_k[h(i)] --
```

And what is our new estimate for the number of copies of element e in our active set? It is

$$\widetilde{x}_i^t := \min_{k=1}^{\ell} A_k(h_k(i)).$$

In other words, each (h_k, A_k) pair gives us an estimate, and we take the least of these. It makes perfect sense — the estimates are all *over*estimates, so taking the least of these is the right thing to do.

This is very much like a Bloom filter, which just maintains membership, whereas this maintains counts. But the idea is very similar.

But how much better is this estimator than the single-table version? Let's do the math.

1. What is the chance that one single estimator has error more than $2\|\mathbf{x}^t\|_1/M$? The expected error is at most $\|\mathbf{x}^t\|_1/M$, so by Markov's inequality,
$$\Pr\left[error > 2 \cdot \frac{\|\mathbf{x}^t\|_1}{M}\right] \leq \frac{1}{2}.$$
2. So the chance that all of the ℓ repetitions have more than $2\|\mathbf{x}^t\|_1/M$ error is
$$\begin{aligned}&\Pr[\text{each of } \ell \text{ repetitions have error } \geq 2\|\mathbf{x}^t\|_1/M] \\ &= \prod_{k=1}^{\ell} \Pr[k^{th} \text{ repetition had error } \geq 2\|\mathbf{x}^t\|_1/M] \\ &\leq (1/2)^{\ell}.\end{aligned}$$
The first equality there used the independence of the hash function choices.
3. And so our estimate $\widetilde{x}_i^t$ has error at most $2\|\mathbf{x}^t\|_1/M$ with probability at least $1 - (1/2)^{\ell}$.

7.4.3 *Final Bookkeeping*

Let's set the parameters now. Set $M = 2/\varepsilon$, so that the error bound $2\|\mathbf{x}^t\|_1/M = \varepsilon\|\mathbf{x}^t\|_1$. Set $\ell = \lg 1/\delta$, then the failure probability is

$(1/2)^\ell = \delta$, and our query will succeed with probability at least $1 - \delta$. Then for any t and i, the estimate $\widetilde{x}_i^t$ satisfies

$$\Pr\left[\left|\widetilde{x}_i^i - x_i^t\right| \leq \varepsilon \|\mathbf{x}^t\|_1\right] \geq 1 - \delta.$$

How small should you make δ? Depends on how many queries you want to do. Suppose you want to make a query a million times a day, then you could make $\delta = 1/10^9 \approx 1/2^{30}$ to get a 1-in-1000 chance that even one of your answers has high error. Our space varies linearly as $\lg 1/\delta$, so setting $\delta = 1/10^{18}$ instead of $1/10^9$ doubles the space usage, but drops the error probability by a factor of billion.

Just as we wanted. And the total space usage is

$$\begin{aligned}\ell \cdot M \text{ counters} &= O(\log 1/\delta) \cdot O(1/\varepsilon) \\ &= O(1/\varepsilon \log 1/\delta) \text{ counters.}\end{aligned}$$

Each counter has to store at most $\lg T$-bit numbers after T time steps.

So a 32-counter can handle a data stream of length 4 billion. If that is not enough, there are ways to reduce this space usage as well, look online for "approximate counters".

Space for Hash Functions: We need to store the ℓ hash functions as well. How much space does that use? The construction from the previous lecture used $s := (\lg M) \cdot (\lg U)$ bits per hash function. Since $M = 2/\varepsilon$, the total space used for all the ℓ functions is

$$\ell \cdot s = O(\log 1/\delta) \cdot (\lg 1/\varepsilon) \cdot (\lg U) \text{ bits.}$$

In summary, using about $1/\varepsilon \times$ poly-logarithmic factors space, and very simple hashing ideas, we could maintain the counts of elements in a data stream under both arrivals and departures (up to an error of $\varepsilon \|\mathbf{x}^t\|_1$).

7.5 Optional: Distinct Elements

Our second example today will be to compute the number of distinct elements seen in the data stream. (Imagine there are no deletions, we are in the addition-only model.) So this number is the number of non-zeroes in the vector $\mathbf{x}^t$. Often this is called the zero-norm of $\mathbf{x}^t$, where we define

$$\|x^t\|_0 := \text{ number of non-zeroes in } \mathbf{x}^t.$$

How should we do this?

7.5.1 An Exact Solution and a Lower Bound

Of course, if we store $\mathbf{x}$ explicitly (using $|U|$ space), we can trivially solve this problem exactly. Or we could store the (at most) t elements seen so far, again we could give an exact answer. And indeed, we cannot do much better if we want no errors. Here's a proof sketch for deterministic algorithms (one can extend this to randomized algorithms with some more work).

Lemma 7.2 (A Lower Bound)**.** *Suppose a deterministic algorithm correctly reports the number of distinct elements for each sequence of length at most N. Suppose $N \leq 2|U|$. Then it must use at least $\Omega(N)$ bits of space.*

Proof. Consider the situation where first we send in some subset S of $N-1$ elements distinct elements of U. Look at the information stored by the algorithm. We claim that we should be able to use this information to identify exactly which of the $\binom{|U|}{N-1}$ subsets of U we have seen so far. This would require

$$\log_2 \binom{|U|}{N-1} \geq (N-1)\big(\log_2 |U| - \log_2(N-1)\big) = \Omega(N)$$

bits of memory.

We used the approximation that

$$\binom{m}{k} \geq \left(\frac{m}{k}\right)^k,$$

and hence

$$\log_2 \binom{m}{k} \geq k(\log_2 m - \log_2 k).$$

Why should we be able to uniquely identify the set of elements until time $N-1$? For a contradiction, suppose we could not tell whether we'd seen S_1 or S_2 after $N-1$ elements had come in. Pick any element $e \in S_1 \setminus S_2$. Now if we gave the algorithm e as the N^{th} element, the number of distinct elements seen would be N if we'd already seen S_2, and $N-1$ if we'd seen S_1. But the algorithm could not distinguish between the two cases, and would return the same answer. It would be incorrect in one of the two cases. This contradicts the claim that the algorithm always correctly reports the number of distinct elements on streams of length N. □

7.5.2 *Much Lower Space using Hashing*

So we need an approximation if we want to use little space. Let's use some hashing magic. Here is the essential idea.

> Suppose there are $d = \|\mathbf{x}\|_0$ distinct elements. If we randomly map d distinct elements onto the line $[0,1]$, we expect to see the smallest mapped value at location $\approx \frac{1}{d}$. (I am assuming that we map these elements *consistently*, so that multiple copies of an element go to the same place.) So if the smallest value is δ, one estimator for the number of elements is $1/\delta$.

To make this work (and analyze it), we change it slightly: The variance of the above estimator is large. However, by the same argument, for any integer s we expect the s^{th} smallest mapped value at $\frac{s}{d}$. We use a larger value of s to reduce the variance.

7.5.3 *The Algorithm*

Assume we have a hash family H with hash functions $h : U \to \{0, 1, \ldots, M-1\}$, and we pick a hash function from this family. We'll soon figure out the precise properties we'll want from this hash family. Moreover, we will later fix the value of the parameter s to be some large constant. Here's the algorithm:

The crucial observation is: it does not matter if you see an element e once or multiple times — the algorithm will behave the same, since

7.1 **foreach** *query, say at time t* **do**
7.2 consider hash values $h(a_1), h(a_2), \ldots, h(a_t)$ seen so far.
7.3 let L_t be the s^{th} smallest distinct hash value $h(a_i)$ in this set.
7.4 output the estimate $D_t = \frac{M \cdot s}{L_t}$.

the output depends on what *distinct* elements we've seen so far. Also, maintaining the s^{th} smallest element can be done by remembering at most s elements. (So we want to make s small.)

How does this help? As a thought experiment, if you had d distinct darts and threw them in the continuous interval $[0, M]$, you would expect the location of the s^{th} smallest dart to be about $\frac{s \cdot M}{d}$. So if the s^{th} smallest dart was at location ℓ in the interval $[0, M]$, you would be tempted to equate $\ell = \frac{s \cdot M}{d}$ and hence guessing $d = \frac{s \cdot M}{\ell}$ would be a good move. Which is precisely why we used the estimate

$$D_t = \frac{M \cdot s}{L_t}.$$

Of course, all this is in expectation—the following theorem formally reasons that this estimate is any good.

Theorem 7.3. *Consider some time t. If H is a 2-universal hash family mapping $U \to \{0, 1, \ldots, M-1\}$, and M is large enough, then both the following guarantees hold:*

$$\Pr[D_t > 2 \|\mathbf{x}^t\|_0] \leq \frac{3}{s}, \text{ and} \tag{7.5}$$

$$\Pr[D_t < \frac{\|\mathbf{x}^t\|_0}{2}] \leq \frac{3}{s}. \tag{7.6}$$

We will prove this in the next section. First, let us make some observations.

1. Setting $s = 8$ means that the estimate D_t lies within $[\frac{\|\mathbf{x}^t\|_0}{2}, 2\|\mathbf{x}^t\|_0]$ with probability at least $1 - (1/4 + 1/4) = 1/2$. (And we can boost the success probability by repetitions.)

2. We will see that the estimation error of a factor of 2 can be made $(1 + \varepsilon)$ by changing the parameters s and k.

3. Finally, observe we now use the stronger assumption that that the hash family is 2-*universal* or *pairwise-independent*. Recall the definition?

   Definition 7.4 (2-Universal Hash Family)**.** A family H of hash functions from $U \to R$ is 2-universal if for any pair of distinct keys $x_1 \neq x_2$ and any set of values $v_1, v_2 \in R$,

   $$\Pr[h(x_1) = v_1 \wedge h(x_2) = v_2] = \frac{1}{|R|^2}.$$

In other words, if we just look at two keys, the probability that they map to two particular values v_1, v_2 in the range R is the same as what we would get if we were to map these elements completely randomly and independently to locations in the R.

7.5.4 *Proof of Theorem 7.3*

Now for the proof of the theorem. We'll prove bound (7.6), the other bound (7.5) is proved identically. Some shorter notation may help. Let $d := \|\mathbf{x}^t\|_0$. Let these d distinct elements be $T = \{e_1, e_2, \ldots, e_d\} \subseteq U$.

The random variable L_t is the s^{th} smallest distinct hash value seen until time t. Our estimate is $\frac{sM}{L_t}$, and we want this to be at least $d/2$. So we want L_t to be *at most* $\frac{2sM}{d}$. In other words,

$$\Pr[\text{ estimate too low }] = \Pr[D_t < d/2] = \Pr[L_t > \frac{2sM}{d}].$$

Recall T is the set of all d $(= \|\mathbf{x}^t\|_0)$ distinct elements in U that have appeared so far. How many of these elements in T hashed to values greater than $2sM/d$? The event that $L_t > 2sM/d$ (which is what we want to bound the probability of) is the same as saying that fewer than s of the elements in T hashed to values smaller than $2sM/d$. For each $i = 1, 2, \ldots, d$, define the indicator

$$X_i = \begin{cases} 1 & \text{if } h(e_i) \leq 2sM/d \\ 0 & \text{otherwise} \end{cases} \tag{7.7}$$

Then $X = \sum_{i=1}^{d} X_i$ is the number of elements seen that hash to values below $2sM/d$. By the discussion above, we get that

$$\Pr\left[L_t < \frac{2sM}{d}\right] \leq \Pr[X < s].$$

We will now estimate the RHS.

Next, what is the chance that $X_i = 1$? The hash $h(e_i)$ takes on each of the M integer values with equal probability, so

$$\Pr[X_i = 1] = \frac{\lfloor sM/2d \rfloor}{M} \geq \frac{s}{2d} - \frac{1}{M}. \tag{7.8}$$

By linearity of expectations,

$$E[X] = E\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} E\left[X_i\right] = \sum_{i=1}^{d} \Pr\left[X_i = 1\right] \geq d \cdot \left(\frac{s}{2d} - \frac{1}{M}\right) = \left(\frac{s}{2} - \frac{d}{M}\right).$$

Let's imagine we set M large enough so that d/M is, say, at most $\frac{s}{100}$. Which means

$$E[X] \geq \left(\frac{s}{2} - \frac{s}{100}\right) = \frac{49\,s}{100}.$$

So by Markov's inequality,

$$\Pr\left[X > s\right] = \Pr\left[X > \frac{100}{49}E[X]\right] \leq \frac{49}{100}.$$

Good? Well, not so good. We wanted a probability of failure to be smaller than $2/s$, we got it to be slightly less than $1/2$. Good try, but no cigar. Yet.

7.5.5 *Enter Chebyshev*

To do better, the final ingredient is Chebyshev's inequality, which you recall from the previous lecture. For a random variable Z with mean μ and variance σ^2,

$$\Pr\left[\,|Z-\mu| \geq c\,\sigma\,\right] \leq \frac{1}{c^2}.$$

A convenient way to rewrite Chebyshev's Inequality is

$$\Pr\left[\,|Z-\mu| \geq C\,\mu\,\right] \leq \frac{\sigma^2}{(C\,\mu)^2}. \tag{7.9}$$

Applying Chebyshev's inequality is useful when the variance of the random variable Z is small. Fortunately, we have some useful facts about variances we can use.

- $\mathrm{Var}(\sum_i Z_i) = \sum_i \mathrm{Var}(Z_i)$ for pairwise-independent random variables Z_i. (Why?)
- And when Z_i is a $\{0,1\}$ random variable, $\mathrm{Var}(Z_i) \leq E[Z_i]$. (Why?)

Applying this to our random variables $X = \sum_i X_i$, we get

$$\mathrm{Var}(X) = \sum_i \mathrm{Var}(X_i) \leq \sum_i E[X_i] = E(X).$$

(The first inequality used that the X_i were pairwise independent, since the hash function was 2-universal.)

Is this variance "low" enough? Let's plug into Chebyshev's inequality (7.9) and find out.

$$\begin{aligned}\Pr[X > s] = \Pr[X > \frac{100}{49}\mu_X] &\leq \Pr[|X-\mu_X| > \frac{50}{49}\mu_X] \\ &\leq \frac{\sigma_X^2}{(50/49)^2\mu_X^2} \leq \frac{1}{(50/49)^2\mu_X} \leq \frac{3}{s}.\end{aligned}$$

Which is precisely what we want for the bound (7.5). The proof for the bound (7.6) is similar and left as an exercise.

If you want the estimate to be at most $\frac{\|\mathbf{x}'\|_0}{(1+\varepsilon)}$, then you would want to bound

$$\Pr[X < \frac{E[X]}{(1+\varepsilon)}].$$

A similar calculation should give this to be at most $\frac{3}{\varepsilon^2 s}$, as long as M was large enough. In that case you would set $s = O(1/\varepsilon^2)$ to get some non-trivial guarantees.

7.5.6 Final Bookkeeping

Excellent. We have a hashing-based data structure that answers "number of distinct elements seen so far" queries, such that each answer is within a multiplicative factor of 2 of the actual value $\|\mathbf{x}^t\|_0$, with small error probability.

Let's see how much space we actually used. Recall that for failure probability $1/2$, we could set $s = 12$, say. And the space to store the s smallest hash values seen so far is $O(s \lg M)$ bits. For the hash functions themselves, the construction from previous lectures (and the homework) uses $O((\lg M) + (\lg U))$ bits per hash function. So the total space used for the entire data structure is

$$O(\log M) + (\lg U) \text{ bits.}$$

What is M? Recall we needed to M large enough so that $d/M \leq s/100$. Since $d \leq |U|$, the total number of elements in the universe, set $M = \Theta(U)$. Now the total number of bits stored is

$$O(\log U).$$

And the probability of our estimate D_t being within a factor of 2 of the correct answer $\|\mathbf{x}^t\|_0$ is at least $1/2$.

7.6 Takeaways

Here are some of the ideas from this chapter:

1. *Data streaming* models the settings where data can be big: too big for the device to store all of its input. "Big" is relative, of course: it can be that you are using your laptop to handle terabytes of data, or your phone to handle even gigabytes, or some low-powered device handing handle megabytes.
2. Streaming algorithms can use only a tiny amount of space, hence they are often very simple and fast! In fact, you may use these algorithms even in settings where you have enough space—just because you want a fast algorithm. It is as though constraining the space usage forces us to avoid complicated solutions.
3. Hashing (and randomization) is a powerful idea: in case you were not convinced by the previous chapters, this is yet another application where it gives surprising, simple, and smart algorithms!
4. Approximation is another powerful idea: if you allow approximate answers, you may be able to use a lot less space and time than if you need exact answers. Of course, it is not suitable for all settings (as is randomization), but consider it when you need to solve a problem.

5. There is a difference between naïve *independent sampling* and *hashing*: the latter is a specific way of sampling where you make the same (correlated!) decisions for each element. You should be careful, and see what is right for your application.

6. If you have an estimator that is weak, running multiple copies of such an estimator and aggregating them (by taking the minimum, or average, or something else) is a common way to increase its strength. (We will call this idea ***parallel repetition***.)

7. Fashions are cyclic. Streaming was fashionable in the 1970s when data was stored on tapes. Reading data from tapes was slow (and there was no random-access), so streaming was the right model to use. Once we had disk drives, these questions went out of fashion. But with the era of big data, these are back. (In fact, we often use some algorithms from the old days!)