A Course in Graduate Algorithms **Lecture 10**

Bloom Filters and Count-Min Sketches

Designed by Prof. Bo Waggoner for the University of Colorado-Boulder *Updated: 2023*

Having discussed hash tables, we will now consider two more data structures that utilize randomness and hash functions. Both of them have very good time and space complexity, but give only approximate correctness guarantees.[1]

Objectives:

- Understand how Bloom filters and count-min setches work.
- Identify scenarios where one would use each type of data structure.
- Work with probability to analyze the data structures.

1 Bloom Filters and Streaming Problems

In *streaming* problems, data or items x arrive sequentially ("online") one at a time. We would like data structures that can answer questions about the history of items, *without* storing the entire history.

In the case of Bloom Filters, we simply want to be able to check which items have arrived:

Operation	**Time**
Add(x)	$O(1)$
CheckIfAdded(x)	$O(1)$

In our setting:

- x is part of a known universe U of possible items (presumably large).
- n of the items will arrive in the stream.
- We will use an array A of m bits for our data structure.

Our key tool will be **hash functions** $h : U \to \{0, \dots, m-1\}$. Recall that an *ideal* hash function is one that maps uniformly at random to that range for every x. We will assume for analysis that we are using ideal hash functions.

Exercise 1. Can we use a hash table to solve this problem? What are the upsides and downsides?

1.1 Deriving Bloom filters

Suppose we try this: using our array A of m bits, we implement Add(x) by setting $A[h(x)] = 1$, and we implement CheckIfAdded(x) by returning true if $A[h(x] = 1$, false otherwise.

Of course, the problem would be *collisions*. As we know from our study of hash tables, after only $n = \Theta(\sqrt{m})$ items arrive, we are likely to have two different items $x \neq x'$ with $h(x) = h(x')$. This will cause a **false positive:** if we query CheckIfAdded(x'), then we will get true even if only x has arrived and not x'.

The solution is to use *multiple* hash functions $h_1, \dots, h_k$. Then:

[1] Content inspired by [1], "Director's Cut" lecture notes, Lecture 6.

- Initially, set $A[i] = 0$ for $i = 0, \ldots, m-1$.
- When each x arrives, call Add(x).
- Add(x): set $A[h_j(x)] = 1$ for all $j = 1, \ldots, k$.
- CheckIfAdded(x): return true if $A[h_j(x)] = 1$ for all $j = 1, \ldots, k$. If any of those bits are zero, return false.

The hope here is that we will need many collisions before we return a false positive. Each item x hashes to j different locations, and if any of those bits are still zero, we know x has not arrived.

1.2 Analysis

There are two things to analyze: false negatives (does CheckIfAdded return false when the correct answer is true?) and false positives (does it return true when the correct answer is false?). Recall that

- k = number of hash functions.
- m = number of bits in our array A.
- n = number of items arriving.

Fact 1 (No False Negatives)**.** *If x has arrived, then CheckIfAdded(x) returns true.*

Proof. When x arrives, we set $A[h_j(x)] = 1$ for all $j = 1, \ldots, k$. Note that we never set any bits back to zero. Therefore, when we call CheckIfAdded(x), it returns true. □

Analyzing false positives is a bit more involved. We will give a series of facts analyzing the process, assuming that we use ideal hash functions.

Lemma 1. *Given any particular "bin" $i \in \{0, \ldots, m-1\}$, we have* $\Pr[A[i] = 1] = 1 - (1 - \frac{1}{m})^{nk}$.

Proof. With n items and k hashes per item, we have nk total hashes. By assumption, each is independently and uniformly drawn from $\{0, \ldots, m-1\}$. So each hash has a probability of $(1 - \frac{1}{m})$ of "missing" bin i. By independence, the probability of all of them missing is the product, or $(1 - \frac{1}{m})^{nk}$. □

Note that, if $nk \approx m$, then $(1 - \frac{1}{m})^{nk} \approx e^{-nk/m}$. Therefore, this lemma leads indirectly to the following claim. However, this claim is informal. In addition to the approximation above, it makes the approximation that each bin is independent. In reality, they are not. However, the approximation is still quite accurate.

Claim 1 (Informal)**.** *Suppose x has not arrived. Then* $\Pr[$*CheckIfAdded(x) returns true*$] \approx (1-p)^k$*, where $p = e^{-nk/m}$.*

Our goal is to make this false positive rate as small as possible. If k is larger, then $(1-p)^k$ is smaller if p is fixed. But if k is larger, then $p = e^{-nk/m}$ is smaller, which increases the error rate. So there is a tradeoff: k should not be too large nor too small.

This tradeoff makes sense if we consider n and m fixed. If k is small, then we are more likely to have k collisions and a false positive. But if k is large, then the array will "fill up" with ones. This makes the chance of false positives higher also. We navigate the tradeoff as follows, using Claim 1.

Claim 2 (Informal)**.** *With an optimal choice of $k \approx \frac{m}{n}\ln(2)$,* $\Pr[$*false positive*$] \approx 2^{-k}$.

Sketch. Recall that we set $p = e^{-nk/m}$, or $k = -\frac{m}{n}\ln(p)$. So

$$\begin{aligned}(1-p)^k &= \exp\left(k\ln(1-p)\right) \\ &= \exp\left(-\frac{m}{n}\ln(p)\ln(1-p)\right).\end{aligned}$$

This is minimized at $p = \frac{1}{2}$. In this case, $(1-p)^k = \left(\frac{1}{2}\right)^k$. □

To understand this further, recall that p was approximately $\Pr[A[i] = 0]$ for any fixed bin i (see Lemma 1). From Claim 2, we get $\frac{nk}{m} \approx \ln(2)$, so $p = e^{-nk/m} \approx \frac{1}{2}$. Information-theoretically, this makes sense: the array stores the most information when about half the bits are ones and the other half zeroes.

Implications for design. In a given scenario, typically we cannot control n, the number of items arriving. However, we can choose m (the number of bits in our array) and k. We can reverse the above analysis: Given a target false positive rate δ, set $k = \log_2 \frac{1}{\delta}$ and $m = \frac{nk}{\ln(2)}$.

Complexity We usually think of the false positive rate as a fixed constant δ, e.g. $\delta = 10^{-8}$. This implies that k is a constant, so Add(x) and CheckIfAdded(x) do run in constant time (check). Also, note that the space used is $m = O(n)$.

2 Count-Min Sketch

Another task in streaming is to estimate how many times a given item has arrived so far. We will be able to do so at least for the "heavy hitters", i.e. significantly common items, using very little memory.

Operation	**Time**
Increment(x)	$O(1)$
Estimate(x)	$O(1)$

Exercise 2. Can we use a hash table to solve *this* problem? What are the upsides and downsides?

Approach. The idea is to use an array of *counters* rather than bits. It'll be a two-dimensional array:

- Height w, a parameter to be chosen later.
- Width d, where d is the number of hash functions used (analogous to k in the previous section).
- To implement Increment(x): we set $A[j, h_j(x)] += 1$ for each $j = 1, \ldots, d$.
- To implement Estimate(x): we return $\min_j A[j, h_j(x)]$.

The idea is that each column $j = 1, \ldots, d$, can be thought of as a separate array or hash table with hash function h_j. When item x arrives, we simply increment the counter $A[j, h_j(x)]$. Collisions mean that we will overestimate the number of arrivals, but it is unlikely that all the columns will have many collisions. So taking the minimum will be a good estimate.

Fact 2. *Let f_x be the true number of times x arrived and let $\hat{f}_x$ =Estimate(x). Then $\hat{f}_x \geq f_x$.*

Proof. Each time x arrives, we increment the counter $A[j, h_j(x)]$ for $j = 1, \ldots, d$. So each of them is at least as large as f_x. Estimate(x) returns $\hat{f}_x = \min_j A[j, h_j(x)] \geq f_x$. □

Claim 3. *Choosing $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \lceil \ln \frac{1}{\delta} \rceil$, for any x, we have $\Pr[\hat{f}_x > f_x + \epsilon n] \leq \delta$.*

Proof. Fix x. Define the indicator variable $X_{j,x,y} = 1$ if $h_j(x) = h_j(y)$. Then $\mathbb{E}[X_{j,x,y}] = \Pr[h_j(x) = h_j(y)] = \frac{1}{w}$.

Let $X_{j,x} = \sum_{y \neq x} X_{j,x,y} f_y$. This is the total amount of collisions with x in column j. Note that $\hat{f}_x = f_x + \min_j X_{j,x}$. In other words, $\hat{f}_x > f_x + \epsilon n$ if and only if, for all j, we have $X_{j,x} > \epsilon n$.

Now, $\mathbb{E}[X_{j,x}] = \frac{1}{w}\sum_{y \neq x} f_y \leq \frac{n}{w}$. By Markov's Inequality[2],

$$\begin{aligned}\Pr[X_{j,x} > \epsilon n] &\leq \frac{\mathbb{E}[X_{j,x}]}{\epsilon n} \\ &= \frac{n}{w\epsilon n} \\ &= \frac{1}{\epsilon w} \\ &\leq \frac{1}{e}.\end{aligned}$$

Now, each column is independent, so the probability that $X_{j,x} > \epsilon n$ for all $j = 1, \ldots, d$ is at most

$$\begin{aligned}\left(\frac{1}{e}\right)^d &\leq e^{-\ln(1/\delta)} \\ &= \delta.\end{aligned}$$

□

This gives a recipe for constructing a CountMin Sketch:

- Decide on an accuracy parameter ϵ.
- Decide an error-probability bound δ.
- Select $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \lceil \ln \frac{1}{\delta} \rceil$, as in Claim 3.

Note that Estimate(x) is only accurate up to $\pm\epsilon n$, where n is the total number of items. So the CountMin Sketch is best at estimating items that arrive many times, i.e. take up a noticeable fraction of all the items. It may not give very accurate counts for items that arrive only a handful of times.

On the positive side, the space usage of a CountMin Sketch is very low. In fact, w and d only depend on ϵ and δ, not on the total number of items arriving![3]

Example. Suppose we wish counts to be accurate to 1% of n, except with probability $e^{-10} \approx 0.00005$. Then we can use a two-dimensional array of dimensions $\lceil 100e \rceil \times 10$, for a total array size of 2720.

References

[1] Jeff Erickson. Algorithms. 2019. `http://jeffe.cs.illinois.edu/teaching/algorithms`.

[2] For a nonnegative variable X and any t, $\Pr[X \geq t] \leq \mathbb{E}[X]/t$.

[3] Presumably the array will hold 64-bit integers, or some integer data type that is large enough to count all the items without overflow.