Lecture #2: Heavy hitters and the count-min sketch

1 The Heavy Hitters Problem

1.1 Finding the Majority Element

Let's begin with a problem that many of you have seen before. It's a common question in technical interviews. You're given as input an array A of length n, with the promise that it has a *majority element* — a value that is repeated in strictly more than $n/2$ of the array's entries. Your task is to find the majority element.

In algorithm design, the usual "holy grail" is a linear-time algorithm. For this problem, you may already know (e.g. from CSE 373 or CSE 421) a subroutine that gives a linear-time solution — just compute the median of A. (Note that a majority element will be the median element.) So let's be more ambitious: can we compute the majority element with a single left-to-right pass through the array? If you haven't seen it before, here's the solution:

```
• Initialize counter := 0, current := NULL.
  [current stores the frontrunner for the majority element]

• For $i = 1$ to $n$:

    – If counter == 0:
      [In this case, there is no frontrunner.]
        ∗ current := A[i]
        ∗ counter++
    – else if A[i]==current:
      [In this case, our confidence in the current frontrunner goes up.]
        ∗ counter++
    – else
      [In this case, our confidence in the current frontrunner goes down.]
        ∗ counter- -

• Return current
```

For example, suppose the input is the array $\{2, 1, 1\}$. The first iteration of the algorithm makes "2" the current guess of the majority element, and sets the counter to 1. The next element decreases the counter back to 0 (since $1 \neq 2$). The final iteration resets the current guess to "1" (with counter value 1), which is indeed the majority element.

More generally, the algorithm correctly computes the majority element of any array that possesses one. We encourage you to formalize a proof of this statement (e.g., by induction on n). The intuition is that each entry of A that contains a non-majority-value can only "cancel out" one copy of the majority value. Since more than $n/2$ of the entries of A contain the majority value, there is guaranteed to be a copy of it left standing at the end of the algorithm.

1.2 The Heavy Hitters Problem

In the *heavy hitters* problem, the input is an array A of length n, and also a parameter k. You should think of n as very large (in the hundreds of millions, or billions), and k as modest (10, 100, or 1000). The goal is to compute the values that occur in the array at least n/k times.[1] Note that there can be at most k such values; and there might be none. The problem of computing the majority element corresponds to the heavy hitters problem with $k \approx 2 - \delta$ for a small value $\delta > 0$, and with the additional promise that a majority element exists.

The heavy hitters problem has lots of applications, as you can imagine. We'll be more specific later when we discuss a concrete solution, but here are some high-level examples:[2]

1. Computing popular products. For example, A could be all of the page views of products on `amazon.com` yesterday. The heavy hitters are then the most frequently viewed products.

2. Computing frequent search queries. For example, A could be all of the searches on Google yesterday. The heavy hitters are then searches made most often.

3. Identifying heavy TCP flows. Here, A is a list of data packets passing through a network switch, each annotated with a source-destination pair of IP addresses. The heavy hitters are then the flows that are sending the most traffic. This is useful for, among other things, identifying denial-of-service attacks.

4. Identifying volatile stocks. Here, A is a list of stock trades.

It's easy to think of more. Clearly, it would be nice to have a good algorithm for the heavy hitters problem at your disposal for data analysis.

The problem is easy to solve efficiently if A is readily available in main memory — just sort the array and do a linear scan over the result, outputting a value if and only if it occurs

[1] A similar problem is the "top-k problem," where the goal is to output the k values that occur with the highest frequencies. The algorithmic ideas introduced in this lecture are also relevant for the top-k problem.

[2] You wouldn't expect there to be a majority element in any of these applications, but you might expect a non-empty set of heavy hitters when k is 100, 1000, or 10000.

(consecutively) at least n/k times. After being spoiled by our slick solution for finding a majority element, we naturally want to do better. Can we solve the heavy hitters problem with a single pass over the array? This question isn't posed quite correctly, since it allows us to cheat: we could make a single pass over the array, make a local copy of it in our working memory, and then apply the sorting-based solution to our local copy. Thus what we mean is: can we solve the Heavy Hitters problem with a single pass over the array, using only a small amount of auxiliary space?[3]

1.3 An Impossibility Result

The following fact might surprise you.

Fact 1.1 *There is* no *algorithm that solves the Heavy Hitters problems in one pass while using a sublinear amount of auxiliary space.*

We next explain the intuition behind Fact 1.1. We encourage you to devise a formal proof, which follows the same lines as the intuition.

Set $k = n/2$, so that our responsibility is to output any values that occur at least twice in the input array A.[4] Suppose A has the form

$$\underbrace{|x_1|x_2|x_3|\cdots|x_{n-1}|}_{\text{set } S \text{ of distinct elements}}\,y|,$$

where $x_1, \ldots, x_{n-1}$ are an arbitrary set S of distinct elements (in $\{1, 2, \ldots, n^2\}$, say) and the final entry y may or may not be in S. By definition, we need to output y if and only if $y \in S$. That is, *answering membership queries reduces to solving the Heavy Hitters problem.* By the "membership problem," we mean the task of preprocessing a set S to answer queries of the form "is $y \in S$"? (A hash table is the most common solution to this problem.) It is intuitive that you cannot correctly answer all membership queries for a set S without storing S (thereby using linear, rather than constant, space) — if you throw some of S out, you might get a query asking about the part you threw out, and you won't know the answer. It's not too hard to make this idea precise using the Pigeonhole Principle.[5]

[3]Rather than thinking of the array A as an input fully specified in advance, we can alternatively think of the elements of A as a "data stream," which are fed to a "streaming algorithm" one element at a time. One-pass algorithms that use small auxiliary space translate to streaming algorithms that need only small working memory. One use case for streaming algorithms is when data arrives at such a fast rate that explicitly storing it is absurd. For example, this is often the reality in the motivating example of data packets traveling through a network switch. A second use case is when, even though data can be stored in its entirety and fully analyzed (perhaps as an overnight job), it's still useful to perform lightweight analysis on the arriving data in real time. The first two applications (popular transactions or search queries) are examples of this.

[4]A simple modification of this argument extends the impossibility result to all interesting values of k — can you figure it out?

[5]Somewhat more detail: if you always use sublinear space to store the set S, then you need to reuse exactly the same memory contents for two different sets S_1 and S_2. Your membership query answers will be the same in both cases, and in one of these cases some of your answers will be wrong.

1.4 The Approximate Heavy Hitters Problem

What should we make of Fact 1.1? Should we go home with our tail between our legs? Of course not — the applications that motivate the heavy hitters problem are not going away, and we still want to come up with non-trivial algorithms for them. In light of Fact 1.1, the best-case scenario would be to find a relaxation of the problem that remains relevant for the motivating applications and also admits a good solution.

In the *ϵ-approximate heavy hitters (ϵ-HH) problem*, the input is an array A of length n and user-defined parameters k and ϵ. The responsibility of an algorithm is to output a list of values such that:

1. Every value that occurs at least $\frac{n}{k}$ times in A is in the list.
2. Every value in the list occurs at least $\frac{n}{k} - \epsilon n$ times in A.

What prevents us from taking $\epsilon = 0$ and solving the exact version of the problem? We allow the space used by a solution to grow as $\frac{1}{\epsilon}$, so as $\epsilon \downarrow 0$ the space blows up (as is necessary, by Fact 1.1).

For example, suppose we take $\epsilon = \frac{1}{2k}$. Then, the algorithm outputs every value with frequency count at least $\frac{n}{k}$, and only values with frequency count at least $\frac{n}{2k}$. Thinking back to the motivating examples in Section 1.2, such an approximate solution is essentially as useful as an exact solution. Space usage $O(\frac{1}{\epsilon}) = O(k)$ is also totally palatable; after all, the output of the heavy hitters or ϵ-HH problem already might be as large as k elements.

2 The Count-Min Sketch

2.1 Discussion

This section presents an elegant small-space data structure, the *count-min sketch* [5], that can be used to solve the ϵ-HH problem. There are also several other good solutions to the problem, including some natural "counter-based" algorithms that extend the algorithm in Section 1.1 for computing a majority element [7, 6]. We focus on the count-min sketch for a number of reasons.

1. It has been implemented in real systems. For example, AT&T has used it in network switches to perform analyses on network traffic using limited memory [4].[6] At Google, a precursor of the count-min sketch (called the "count sketch" [3]) has been implemented on top of their MapReduce parallel processing infrastructure [8]. One of the original motivations for this primitive was log analysis (e.g., of source code check-ins), but presumably it is now used for lots of different analyses.
2. The data structure is based on hashing, and as such fits in well with the current course theme.

[6]There is a long tradition in the Internet of designing routers that are "fast and dumb," and many of them have far less memory than a typical smartphone.

3. The data structure introduces a new theme, present in many of the next few lectures, of "lossy compression." The goal here is to throw out as much of your data as possible while still being able to make accurate inferences about it. What you want to keep depends on the type of inference you want to support. For approximately preserving frequency counts, the count-min sketch shows that you can throw out almost all of your data!

We'll only discuss how to use the count-min sketch to solve the approximate heavy hitters problem, but it is also useful for other related tasks (see [5] for a start). Another reason for its current popularity is that its computations parallelize easily — as we discuss its implementation, you might want to think about this point.

2.2 A Role Model: The Bloom Filter

This section briefly reviews the bloom filter data structure, which is a role model for the count-min sketch. No worries if you haven't seen bloom filters before; our treatment of the count-min sketch below is self-contained. There are also review videos covering the details of bloom filters on the course Web site.

The raison d'être of a bloom filter is to solve the *membership problem.* The client can insert elements into the bloom filter and the data structure is responsible for remembering what's been inserted. The bloom filter doesn't do much, but what it does it does very well.

Hash tables also offer a good solution to the membership problem, so why bother with a bloom filter? The primary motivation is to save space — a bloom filter compresses the stored set more than a hash table. In fact, the compression is so extreme that a bloom filter cannot possibly answer all membership queries correctly. That's right, it's a *data structure that makes errors.* Its errors are "one-sided," with no false negatives (so if you inserted an element, the bloom filter will always confirm it) but with some false positives (so there are "phantom elements" that the data structure claims are present, even though they were never inserted). For instance, using 8 bits per stored element — well less than the space required for a pointer, for example — bloom filters can achieve a false positive probability less than 2%. More generally, bloom filters offer a smooth trade-off between the space used and the false positive probability. Both the insertion and lookup operations are super-fast ($O(1)$ time) in a bloom filter, and what little work there is can also be parallelized easily.

Bloom filters were invented in 1970 [1], back when space was at a premium for everything, even spellcheckers.[7] This century, bloom filters have gone viral in the computer networking community [2]. Saving space is still a big win in many networking applications, for example by making better use of the scarce main memory at a router or by reducing the amount of communication required to implement a network protocol.

Bloom filters serve as a role model for the count-min sketch in two senses. First, bloom filters offer a proof of concept that sacrificing a little correctness can yield significant space savings. Note this is exactly the trade-off we're after: Fact 1.1 states that exactly solving the

[7]The proposal was to insert all correctly spelled words into a bloom filter. A false positive is then a misspelled word that the spellchecker doesn't catch.

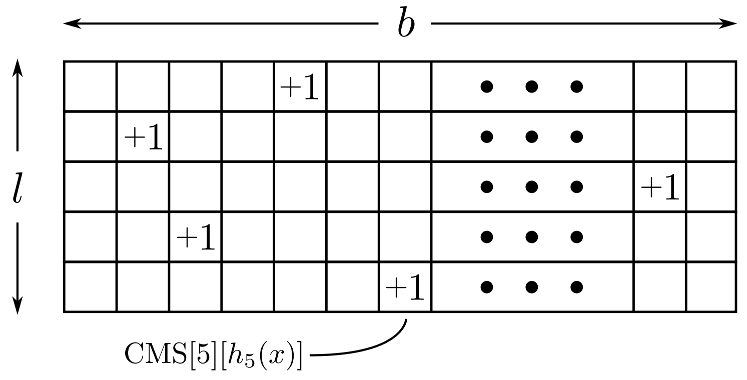

Figure 1: Running $\mathsf{Inc}(x)$ on the CMS data structure. Each row corresponds to a hash function h_i.

heavy hitters problem requires linear space, and we're hoping that by relaxing correctness — i.e., solving the ϵ-HH problem instead — we can use far less space. Second, at a technical level, if you remember how bloom filters are implemented, you'll recognize the count-min sketch implementation as a bird of the same feather.

2.3 Count-Min Sketch: Implementation

The count-min-sketch supports two operations: $\mathsf{Inc}(x)$ and $\mathsf{Count}(x)$.[8] The operation $\mathsf{Count}(x)$ is supposed to return the *frequency count* of x, meaning the number of times that $\mathsf{Inc}(x)$ has been invoked in the past.

The count-min sketch has two parameters, the number of buckets b and the number of hash functions ℓ. We'll figure out how to choose these parameters in Section 2.5, but for now you might want to think of b as in the thousands and of ℓ as 5.[9] The point of b is to compress the array A (since $b \ll n$). This compression leads to errors. The point of ℓ is to implement a few "independent trials," which allows us to reduce the error. What's important, and kind of amazing, is that these parameters are *independent* of the length n of the array that we are processing (recall n might be in the billions, or even larger).

The data structure is just a $\ell \times b$ 2-D array CMS of counters (initially all 0). See Figure 1. After choosing ℓ hash functions $h_1, \ldots, h_\ell$, each mapping the universe of objects

[8]The same data structure supports weighted increments, of the form $\mathsf{Inc}(x,\Delta)$ for $\Delta \geq 0$, in exactly the same way. With minor modifications, the data structure can even support deletions. We focus on the special case of incrementing by 1 for simplicity, and because it is sufficient for many of the motivating applications.

[9]Where do we get ℓ hash functions from? The same way we get a single hash function. If we're thinking of hash functions as being drawn at random from a universal family, we just make ℓ independent draws from the family. If we're thinking about deterministic but well-crafted hash functions, in practice it's usually sufficient to take two good hash functions h_1, h_2 and use ℓ linear combinations of them (e.g. h_1, $h_1 + h_2$, $h_1 + 2h_2$, $\ldots$, $h_1 + (\ell - 1)h_2$). Another common hack, which you'll implement on Mini-Project #1, is to derive each h_i from a single well-crafted hash function h by defining $h_i(x)$ as something like the hash (using h) of the string formed by x with "i" appended to it.

to $\{1, 2, \ldots, b\}$, the code for $\mathsf{Inc}(x)$ is simply:

- for $i = 1, 2, \ldots, \ell$:
    - increment $\text{CMS}[i][h_i(x)]$

Assuming that every hash function can be evaluated in constant time, the running time of the operation is clearly $O(\ell)$.

To motivate the implementation of $\mathsf{Count}(x)$, fix a row $i \in \{1, 2, \ldots, \ell\}$. Every time $\mathsf{Inc}(x)$ is called, the same counter $\text{CMS}[i][h_i(x)]$ in this row gets incremented. Since counters are never decremented, we certainly have

$$\text{CMS}[i][h_i(x)] \geq f_x, \tag{1}$$

where f_x denotes the frequency count of object x. If we're lucky, then equality holds in (1). In general, however, there will be *collisions*: objects $y \neq x$ with $h_i(y) = h_i(x)$. (Note with $b \ll n$, there will be lots of collisions.) Whenever $\mathsf{Inc}(y)$ is called for an object y that collides with x in row i, this will also increment the same counter $\text{CMS}[i][h_i(x)]$. So while $\text{CMS}[i][h_i(x)]$ cannot underestimate f_x, it generally overestimates f_x.

The ℓ rows of the count-min sketch give ℓ different estimates of f_x. How should we aggregate these estimates? Later in the course, we'll see scenarios where using the mean or the median is a good way to aggregate. Here, our estimates suffer only one-sided error — all of them can only be bigger than the number f_x we want to estimate, and so it's a no-brainer which estimate we should pay attention to. The *smallest* of the estimates is clearly the best estimate. Thus, the code for $\mathsf{Count}(x)$ is simply:

- return $\min_{i=1}^{\ell} \text{CMS}[i][h_i(x)]$

The running time is again $O(\ell)$. By (1), the data structure has one-sided error — it only returns overestimates of true frequency counts, never underestimates. The key question is obviously: *how large are typical overestimates?* The answer depends on how we set the parameters b and ℓ. As b gets bigger, we'll have fewer collisions and hence less error. As ℓ gets bigger, we'll take the minimum over more independent estimates, resulting in tighter estimates. Thus the question is whether or not modest values of b and ℓ are sufficient to guarantee that the overestimates are small. This is a quantitative question that can only be answered with mathematical analysis; we do this in the next section (and the answer is yes!).

Remark 2.1 (Comparison to Bloom Filters) The implementation details of the count-min sketch are very similar to those of a bloom filter. The latter structure only uses bits, rather than integer-valued counters. When an object is inserted into a bloom filter, ℓ hash functions indicate ℓ bits that should be set to 1 (whether or not they were previously 0 or 1). The count-min sketch, which is responsible for keeping counts rather than just tracking membership, instead increments ℓ counters. Looking up an object in a bloom filter just involves checking the ℓ bits corresponding to that object — if any of them are still 0, then the object has not been previously inserted. Thus Lookup in a bloom filter can be thought of as taking the minimum of ℓ bits, which exactly parallels the Count operation of a count-min-sketch. That the count-min sketch only overestimates frequency counts corresponds to the bloom filter's property that it only suffers from false positives.

2.4 Count-Min Sketch: Heuristic Error Analysis

The goal of this section is to analyze how much a count-min sketch overestimates frequency counts, as a function of the parameters b and ℓ. Once we understand the relationship between the error and these parameters, we can set the parameters to guarantee simultaneously small space and low error.

Fix an object x. Let's first think about a single row i of the count-min sketch; we'll worry about taking the minimum over rows later. After a bunch of $\mathsf{Inc}(x)$ operations have been executed, what's the final value of $\text{CMS}[i][h_i(x)]$, row i's estimate for the frequency count of x?

If we're lucky and no other objects collide with x in the ith row, then $\text{CMS}[i][h_i(x)]$ is just the true frequency count f_x of x. If we're unlucky and some object y collides with x in the ith row, then y contributes its own frequency count f_y to $\text{CMS}[i][h_i(x)]$. More generally, $\text{CMS}[i][h_i(x)]$ is the sum of the contributions to this counter by x and all other objects that collide with it:

$$\text{CMS}[i][h_i(x)] = f_x + \sum_{y \in S} f_y, \tag{2}$$

where $S = \{y \neq x : h_i(y) = h_i(x)\}$ denotes the objects that collide with x in the ith row. In (2), f_x and the f_y's are fixed constants (independent of the choice of h_i), while the set S will be different for different choices of the hash function h_i.

Recall that a good hash function spreads out a data set as well as if it were a random function. With b buckets and a good hash function h_i, we expect x to collide with a roughly $1/b$ fraction of the other elements $y \neq x$ under h_i. Thus we expect

$$\text{CMS}[i][h_i(x)] = f_x + \frac{1}{b}\sum_{y \neq x} f_y \leq f_x + \frac{n}{b}, \tag{3}$$

where in the inequality we use that the sum of the frequency counts is exactly the total number n of increments (each increment adds 1 to exactly one frequency count). See also Section 2.5 for a formal (non-heuristic) derivation of (3).

We should be pleased with (3). Recall the definition of the ϵ-approximate heavy hitters problem (Section 1.4): the goal is to identify objects with frequency count at least $\frac{n}{k}$, without being fooled by any objects with frequency count less than $\frac{n}{k} - \epsilon n$. This means we just need to estimate the frequency count of an object up to additive one-sided error ϵn. If we take the number of buckets b in the count-min sketch to be equal to $\frac{1}{\epsilon}$, then (3) says the expected overestimate of a given object is at most ϵn. Note that the value of b, and hence the number of counters used by the data structure, is completely independent of n! If you think of $\epsilon = .001$ and n as in the billions, then this is pretty great.

So why aren't we done? We'd like to say that, in addition to the expected overestimate of a frequency count being small, with very large probability the overestimate of a frequency count is small. (For a role model, recall that typical bloom filters guarantee a false positive probability of 1-2%.) This requires translating our bound on an expectation to a bound on a probability.

Next, we observe that (3) implies that the probability that a row's overestimate of x is more than $\frac{2n}{b}$ is less than 50%. (If not, the expected overestimate would be greater than $\frac{1}{2} \cdot \frac{2n}{b} = \frac{n}{b}$, contradicting (3).) This argument is a special case of "Markov's inequality;" see Section 2.5 for details.

A possibly confusing point in this heuristic analysis is: in the observation above, what is the probability over, exactly? I.e., where is the randomness? There are two morally equivalent interpretations of the analysis in this section. The first, which is carried out formally and in detail in Section 2.5, is to assume that the hash function h_i is chosen uniformly at random from a universal family of hash functions. The second is to assume that the hash function h_i is fixed and that the data is random. If h_i is a well-crafted hash function, then your particular data set will almost always behave like random data.[10]

Remember that everything we've done so far is just for a single row i of the hash table. The output of $\mathsf{Count}(x)$ exceeds f_x by more than ϵn only if *every* row's estimate is too big. Assuming that the hash functions h_i are independent,[11] we have

$$\mathbf{Pr}\left[\min_{i=1}^{\ell} \text{CMS}[i][h_i(x)] > f_x + \frac{2n}{b}\right] = \prod_{i=1}^{\ell} \mathbf{Pr}\left[\text{CMS}[i][h_i(x)] > f_x + \frac{2n}{b}\right] \leq \left(\frac{1}{2}\right)^{\ell}.$$

To get an overestimate threshold of ϵn, we can set $b = \frac{2}{\epsilon}$ (so e.g., 200 when $\epsilon = .01$). To drive the error probability — that is, the probability of an overestimate larger than this threshold — down to the user-specified value δ, we set

$$\left(\frac{1}{2}\right)^{\ell} = \delta$$

and solve to obtain $\ell = \log_2 \frac{1}{\delta}$. (This is between 6 and 7 when $\delta = .01$.) Thus the total number of counters required when $\delta = \epsilon = .01$ is barely over a thousand (no matter how long

[10] In an implementation that chooses h_i deterministically as a well-crafted hash function, the error analysis below does not actually hold for an arbitrary data set. (Recall that for every fixed hash function there is a pathological data set where everything collides.) So instead we say that the analysis is "heuristic" in this case, meaning that while not literally true, we nevertheless expect reality to conform to its predictions (because we expect the data to be non-pathological). Whenever you do a heuristic analysis to predict the performance of an implementation, you should always measure the implementation's performance to double-check that it's working as expected. (Of course, you should do this even when you've proved performance bounds rigorously — there can always be unmodeled effects (cache performance, etc.) that cause reality to diverge from your theoretical predictions for it.)

[11] Don't forget that probabilities factor only for independent events. There are again two interpretations of this step: in the first, we assume that each h_i is chosen independently and randomly from a universal family of hash functions; in the second, we assume that the h_i's are sufficiently well crafted that they almost always behave as if they were independent on real data.

the array is!). See Section 2.6 for a detailed recap of all of the count-min sketch's properties, and Section 2.5 for a rigorous and optimized version of the heuristic analysis in this section.

2.5 Count-Min Sketch: Rigorous Error Analysis

This section carries out a rigorous version of the heuristic error analysis in Section 2.4. Let f_x denote the true frequency count of x, and Z_i the (over)estimate CMS[i][$h_i(x)$]. Z_i is a random variable over the state space equal to the set of all possible hash functions h_i. (I.e., given an h_i, Z_i is fully determined.)

If we're lucky and no other objects collide with x in the ith row, then $Z_i = f_x$. If we're unlucky and some object y collides with x in the ith row, then y contributes its own frequency count f_y to Z_i. As in (2), we can write

$$Z_i = f_x + \sum_{y \in S} f_y, \tag{4}$$

where $S = \{y \neq x : h_i(y) = h_i(x)\}$ denotes the objects that collide with x in the ith row. In (4), f_x and the f_y's are fixed constants (independent of the choice of h_i), while the set S is random (i.e., different for different choices of h_i).

To continue the error analysis, we make the following assumption:

(*) For every pair x, y of distinct objects, $\mathbf{Pr}[h_i(y) = h_i(x)] \leq \frac{1}{b}$.

Assumption (*) basically says that, after conditioning on the bucket to which h_i assigns an object x, the bucket h_i assigns to some other object y is uniformly random. For example, the assumption would certainly be satisfied if h_i is a completely random function. It is also satisfied if h_i is chosen uniformly at random from a universal family — it is precisely the definition of such a family.

Before using assumption (*) to analyze (4), we recall *linearity of expectation*: for any real-valued random variables $X_1, \ldots, X_m$ defined on the same probability space,

$$\mathbf{E}\left[\sum_{j=1}^{m} X_j\right] = \sum_{j=1}^{m} \mathbf{E}[X_j]\,. \tag{5}$$

That is, the expectation of a sum is just the sum of the expectations, *even if the random variables are not independent.*[12] The statement is trivial to prove — just expand the expectations and reverse the order of summation — and insanely useful.

To put the pieces together, we first rewrite (4) as

$$Z_i = f_x + \sum_{y \neq x} f_y \mathbf{1}_y, \tag{6}$$

[12]Note the analogous statement for products is false if the X_j's are not independent. For example, suppose $X_1 \in \{0, 1\}$ is uniform while $X_2 = 1 - X_1$. Then $\mathbf{E}[X_1 \cdot X_2] = 0$ while $\mathbf{E}[X_1] \cdot \mathbf{E}[X_2] = \frac{1}{4}$.

where $\mathbf{1}_y$ is the indicator random variable that indicates whether or not y collides with x under h_i:

$$\mathbf{1}_y = \begin{cases} 1 & \text{if } h_i(y) = h_i(x) \\ 0 & \text{otherwise.} \end{cases}$$

Recalling that f_x and the f_y's are constants, we can apply linearity of expectation to (6) to obtain

$$\mathbf{E}[Z_i] = f_x + \sum_{y \neq x} f_y \cdot \mathbf{E}[\mathbf{1}_y]\,. \tag{7}$$

As indicator random variables, the $\mathbf{1}_y$'s have very simple expectations:

$$\mathbf{E}[\mathbf{1}_y] = 1 \cdot \underbrace{\mathbf{Pr}[\mathbf{1}_y = 1]}_{=\mathbf{Pr}[h_i(y)=h_i(x)]} + \underbrace{0 \cdot \mathbf{Pr}[\mathbf{1}_y = 0]}_{=0} = \mathbf{Pr}[h_i(y) = h_i(x)] \overset{(*)}{\leq} \frac{1}{b}. \tag{8}$$

Combining (7) and (8) gives

$$\mathbf{E}[Z_i] \leq f_x + \frac{1}{b} \sum_{y \neq x} f_y \leq f_x + \frac{n}{b}. \tag{9}$$

Next we translate this bound on an expectation to a bound on a probability. A simple and standard way to do this is via *Markov's inequality.*

Proposition 2.2 (Markov's Inequality) *If X is a nonnegative random variable and $c > 1$ is a constant, then*

$$\mathbf{Pr}[X > c \cdot \mathbf{E}[X]] \leq \frac{1}{c}.$$

The proof of Markov's inequality is simple. For example, suppose you have a nonnegative random variable X with expected value 10. How frequently could it take on a value greater than 100? (So $c = 10$.) In principle, it is possible that X has value exactly 100 10% of the time (if it has value 0 the rest of the time). But it can't have value strictly greater than 100 10% or more of the time — if it did, its expectation would be strictly greater than 10. An analogous argument applies to nonnegative random variables with any expectation and for any value of c.

Let's return to our error analysis, for a fixed object x and row i. Define

$$X = Z_i - f_x \geq 0$$

as the amount by which the ith row of the count-min sketch overestimates x's frequency count f_x. By (9), with $b = \frac{e}{\epsilon}$, the expected value of X is at most $\frac{\epsilon n}{e}$.[13] Since this overestimate is always nonnegative, we can apply Markov's inequality (Proposition 2.2) with $\mathbf{E}[X] = \frac{\epsilon n}{e}$ and $c = e$ to obtain

$$\mathbf{Pr}\big[X > e \cdot \tfrac{\epsilon n}{e}\big] \leq \frac{1}{e}$$

[13]Here $e = 2.718\ldots$, which gives slightly better constant factors than the choice of 2 in Section 2.4.

and hence

$$\mathbf{Pr}[Z_i > f_x + \epsilon n] \le \frac{1}{e}.$$

Assuming that the hash functions are chosen independently, we have

$$\mathbf{Pr}\left[\min_{i=1}^{\ell} Z_i > f_x + \epsilon n\right] = \prod_{i=1}^{\ell} \mathbf{Pr}[Z_i > f_x + \epsilon n] \le \frac{1}{e^{\ell}}. \tag{10}$$

To achieve a target error probability of δ, we just solve for ℓ in (10) and find that $\ell \ge \ln \frac{1}{\delta}$ rows are sufficient. For δ around 1%, $\ell = 5$ is good enough.

2.6 Count-Min Sketch: Final Report Card

- The space required is that for $\frac{e}{\epsilon} \ln \frac{1}{\delta}$ counters. Recall from Section 1.4 that for the ϵ-HH problem, $\epsilon = \frac{1}{2k}$ is a sensible choice. For $k = 100$ and $\delta = .01$ this is in the low thousands. For larger values of k, the number of counters needed scales linearly.[14] In any case, the number of counters is totally independent of n (which could be in billions)! This is the magic of the count-min sketch — you can throw out almost all of your data set and still maintain approximate frequency counts. Contrast this with bloom filters, and pretty much every other data structure that you've seen, where the space grows linearly with the number of processed elements.[15]

- Assuming the hash functions take constant time to evaluate, the Inc and Count operations run in $O(\ln \frac{1}{\delta})$ time.

- The count-min sketch guarantees 1-sided error: no matter how the hash functions $h_1, \ldots, h_\ell$ are chosen, for every object x with frequency count f_x, the count-min sketch returns an estimate Count(x) that is at least f_x.

- Assuming that each hash function $h_1, \ldots, h_\ell$ is chosen uniformly from a universal family, for every object x with frequency count f_x, the probability that the estimate Count(x) output by the count-min sketch is greater than $f_x + \epsilon n$ is at most δ. One would expect comparable performance for fixed well-crafted hash functions $h_1, \ldots, h_\ell$ on pretty much any data set that you might encounter.

[14]Note that the challenging case, and the case that often occurs in our motivating applications, is an array A that simultaneously has lots of different elements but also a few elements that occur many times. If there are few distinct elements, one can maintain the frequency counts exactly using one counter per distinct element. If no elements occur frequently, then there's nothing to do.

[15]How is this possible? Intuitively, with an error of ϵn allowed, only the elements with with large ($> \epsilon n$) frequency counts matter, and there can be at most $\frac{1}{\epsilon}$ such elements (why?). Thus it is plausible that space proportional to $\frac{1}{\epsilon}$ might be enough. Of course, there's still the issue of not knowing in advance which $\approx \frac{1}{\epsilon}$ elements are the important ones!

2.7 Solving the ϵ-Heavy Hitters Problem

The count-min sketch can be used to solve the ϵ-HH problem from Section 1.4. If the total number n of array elements is known in advance, this is easy: set $\epsilon = \frac{1}{2k}$, process the array elements using a count-min sketch in a single pass, and remember an element once its estimated frequency (according to the count-min sketch) is at least $\frac{n}{k}$.

When n is not known a priori, here is one way to solve the problem. Assume that $\epsilon = \frac{1}{2k}$ and so the number of counters is $O(k \ln \frac{1}{\delta})$. In a single left-to-right pass over the array A, maintain the number m of array entries processed thus far. We store potential heavy hitters in a heap data structure. When processing the next object x of the array, we invoke $\mathsf{Inc}(x)$ followed by $\mathsf{Count}(x)$. If $\mathsf{Count}(x) \geq \frac{m}{k}$, then we store x in the heap, using the key $\mathsf{Count}(x)$. (If x was already in the heap, we delete it before re-inserting it with its new key value.) This requires one insertion and at most one deletion from the heap. Also, whenever m grows to the point that some object x stored in the heap has a key less than m/k (checkable in $O(1)$ time via Find-Min), we delete x from the heap (via Extract-Min). After finishing the pass, we output all of the objects in the heap.

Assume for simplicity that the count-min sketch makes no large errors, with $\mathsf{Count}(x) \in [f_x, f_x + \epsilon n]$ for all x. Every object x with $f_x \geq \frac{n}{k}$ is in the heap at the end of the pass. (To see this, consider what happens the last time that x occurs.) The "no large errors" assumption implies an approximate converse: every object x in the heap has true frequency count at least $\frac{n}{k} - \epsilon n = \frac{n}{2k}$ (other objects would be deleted from the heap by the end of the pass). These are exactly the two properties we ask of a solution to the ϵ-HH problem. If the count-min sketch makes large errors on a few objects, then these objects might erroneously appear in the final output as well. Ignoring the objects with large errors, the heap contains at most $2k$ objects at all times (why?), so maintaining the heap requires an extra $O(\log k) = O(\log \frac{1}{\epsilon})$ amount of work per array entry.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[3] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[4] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 35–46, 2004.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[6] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, pages 398–412, 2005.

[7] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[8] R. Pike, S. Dorward an R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Dynamic Grids and Worldwide Computing*, 13(4):277–298, 2005.