4 Streams and Sketches

Streams can be seen as read-once sequences of data. Algorithms on streams must work under what we call the *data stream axioms*, already stated in chapter 2:

1. Only one pass is allowed on the stream; each stream item can be observed only once.
2. The processing time per item must be low.
3. Memory use must be low as well, certainly sublinear in the length of the stream; this implies that only a few stream items can be explicitly stored.
4. The algorithm must be able to provide answers at any time.
5. Streams evolve over time, that is, they are nonstationary data sources.

The concern of this chapter is the design of algorithms that satisfy the first four axioms. We make *no probabilistic assumption* on the stream, on the process that generates the stream items, or on any statistical laws it may obey. Streams are generated *adversarially*, that is, following the patterns that are hardest for the algorithm. Thus, axiom 5 is irrelevant in this chapter. In contrast, in chapter 5 we will define a more benign model of stream generation where items are generated according to some stochastic process, and therefore obey statistical laws. This assumption lets us formalize axiom 5 and is implicit in most available stream ML and data mining algorithms.

Many solutions to streaming problems use the notion of a stream *sketch* or *summary*. A sketch is a data structure plus accompanying algorithms that read a stream and store sufficient information to be able to answer one or more predefined queries about the stream. We will view sketches as building blocks for higher-level learning and mining algorithms on streams. In this light, the requirement to use little memory is particularly pressing, because the mining algorithm will often create not a single sketch, but many, for keeping track of many different statistics on the data simultaneously.

4.1 Setting: Approximation Algorithms

We first fix some notation used throughout the rest of the book. We use $\mathrm{E}[X]$ and $\mathrm{Var}(X)$ to denote the expected value and the variance, respectively, of random variable X. A $\log$ without a subscript denotes a logarithm in base 2, and $\ln$ denotes a natural logarithm. A function f is $O(g)$ if there is a constant $c > 0$ such that for every x we have $f(x) \leq c \cdot g(x)$. The cardinality of a set A

is denoted $|A|$. In pseudocode, $\leftarrow$ denotes variable assignment, and $\triangleright$ starts a comment line.

In streaming, an *item* is simply an element of a universe of items I without any relevant inner structure. Set I is potentially very large, which means that it is unfeasible to use memory proportional to $|I|$. We will sometimes assume that storing an element of I takes one unit of memory; this is necessary when I is infinite—for instance, the set of real numbers. In other cases we will go into more detail and count the number of bits required to store an item, which is at least $\log |I|$.

We define a *sketching* algorithm by giving three operations: an *Init*$(\dots)$ operation that initializes the data structure, possibly with some parameters such as the desired approximation or the amount of memory to be used; an *Update*(*item*) operation that will be applied to every item on the stream; and a *Query*$(\dots)$ operation that returns the current value of a function of interest on the stream read so far (and may or may not have parameters). In general, a sketch may implement several types of queries.

Many functions that can be computed easily on offline data are impossible to compute *exactly* on streams in sublinear memory. For example, computing the number of distinct items seen requires linear memory in the length of the stream, in the worst case and large enough item universes. On the other hand, for many functions there are algorithms that provide *approximate* solutions and obtain large gains in memory and time with only a small loss in accuracy, which is often acceptable in applications. We next introduce some formalism for discussing approximate computation.

Let f and g be two real-valued functions. We think of g as an approximation of f. For an *accuracy* value ϵ, we say that g is

- an *absolute* or *additive* ϵ-approximation of f if $|f(x) - g(x)| \leq \epsilon$ for every input x;
- a *relative* or *multiplicative* ϵ-approximation of f if $|f(x) - g(x)| \leq \epsilon |f(x)|$ for every input x.

Many approximate algorithms on streams are also *randomized*, meaning that they have access to a source of random bits or numbers to make their choices. Their output on a fixed input is not uniquely defined, as it may be different on different runs. It should therefore be treated as a *random variable* g that hopefully is close to the desired f in most places. The definition of (absolute or relative) (ϵ, δ)-approximation is like that of ϵ-approximation as above, but

requiring only that the approximation above holds with probability at least $1 - \delta$ over the runs of the algorithm, rather than in every single run. For example, to say that the output g of an algorithm is an absolute $(0.1, 0.01)$-approximation of a desired function f, we require that, for every x, $g(x)$ is within ± 0.1 of $f(x)$ at least 99% of the times we run the algorithm. The value $1 - \delta$ is usually called *confidence*.

This framework is similar to the PAC learning model [146] used to formalize prediction learning problems, where PAC stands for "probably approximately correct." It should be noted that, usually, the approximation guarantees only hold when the updates and queries made do not depend on the answers returned by previous queries. That is, we cannot apply these guarantees for *adaptive queries*, which are common in exploratory data analysis.

In statistics, the traditional way of approximately describing a random variable X is by giving its expected value $\mathrm{E}[X]$ and its variance $\mathrm{Var}(X)$. Suppose we have a randomized algorithm g that approximates a desired function f, in the sense that $\mathrm{E}[g(x)] = f(x)$ with some variance $\mathrm{Var}(g) = \sigma^2$. In that case, we can design, on the basis of g, another algorithm h that approximates f in the seemingly stricter sense of (ϵ, δ)-approximation. The new algorithm h takes any ϵ and δ as parameters, in addition to x, and uses $O((\sigma^2/\epsilon^2)\ln(1/\delta))$ times the memory and running time of f. As we might expect, it uses more resources as more accuracy and more confidence are required. The details of the conversion are given in section 4.9.2, but we describe next the main ingredient, as it is ubiquitous in stream algorithmics and stream mining.

4.2 Concentration Inequalities

While processing a stream, algorithms often need to make decisions based on the information they have seen, such as deciding the value to output or choosing among some finite number of options. If a decision has to be made around accumulated statistics, then a useful tool would be a method to establish the likelihood of an undesired event occurring. In other words, one could ask the question: If I make a decision at this point in time about the model I am constructing, what is the chance that I will be wrong in the long run? Such decision-making leans heavily on concentration inequalities.

Inequalities stating that random variables are concentrated around the mean with high probability are known as *concentration inequalities* or *large-deviation bounds*. They can also be regarded as providing *confidence bounds*

for the variables. An example is the law of large numbers of classical probability theory, or the three-sigma rule, which states that for a normal distribution nearly all (99.73%) of the probability mass lies within three standard deviations of the mean.

The most basic concentration inequality is *Markov's inequality*, which states that for any nonnegative random variable X, and $t > 0$,

$$\Pr[X \geq t] \leq \mathrm{E}[x]/t.$$

Chebyshev's inequality is an application of Markov's inequality and states that for any nonnegative random variable X, and $t > 0$,

$$\Pr[X - \mathrm{E}[X] \geq t] \leq \mathrm{Var}[X]/t^2.$$

Often, we want to estimate $\mathrm{E}[X]$ by averaging several copies of X. Averages, a special kind of random variable, converge to their expectation much faster than stated by Chebyshev's inequality, as the number of copies grows. For any n, define $X = \left(\sum_{i=1}^{n} X_i\right)/n$ where $X_1, \ldots, X_n$ are independent variables taking values in $[0, 1]$ with $\mathrm{E}[X] = p$. Then:

1. **Chernoff's bound:** For all $\epsilon \in (0, 1)$,

$$\Pr[\,|X - p| > \epsilon\, p\,] \leq 2\exp\left(-\epsilon^2\, pn/3\right).$$

2. **Hoeffding's bound:** For all $\epsilon > 0$,

$$\Pr[\,|X - p| > \epsilon\,] \leq 2\exp\left(-2\epsilon^2 n\right).$$

3. **The normal approximation:** If in a draw of $X_1, \ldots, X_n$ we obtain $X = \hat{p}$, then $|p - \hat{p}| > \epsilon$ happens with probability about $2(1 - \Phi(\epsilon\sqrt{n/(\hat{p}(1-\hat{p}))}\,))$, where $\Phi(z)$ is the probability that a normally distributed variable with average 0 and standard deviation 1 is less than z. The function $\Phi(z)$ equals $\frac{1}{2}(1 + \mathrm{erf}(z/\sqrt{2}))$, where erf is the error function available in most programming languages.

We can apply the bounds in the reverse direction and derive a formula relating the deviation from the mean and its probability. For example, if $\epsilon = \sqrt{\log(2/\delta)/2n}$, Hoeffding's bound gives $\Pr[\,|X - p| > \epsilon\,] \leq \delta$. This is useful for obtaining additive (ϵ, δ)-approximations. Chernoff's bound gives multiplicative approximations instead, but has the disadvantage that the bound mentions $p = \mathrm{E}[X]$, the unknown value that we want to estimate; also, it applies only to $\epsilon < 1$.

The normal approximation is based on the fact that, as n tends to infinity, the variable nX tends to a normal with mean pn and standard deviation at most $\sqrt{p(1-p)n}$; notice that 0/1-valued or Bernoulli variables are the worst case and lead to this variance exactly. It is not completely rigorous, both because n is finite and because it permutes the roles of p and $\hat{p}$. However, folklore says that the approximation is correct enough for all practical purposes when $\hat{p}n$ and $(1-\hat{p})n$ are larger than about 30, and tighter than Hoeffding's and Chernoff's bounds, which require larger n to reach the same uncertainty ϵ. It is thus preferable in most practical settings, except when we are interested in proving rigorous theorems about the performance of algorithms.

Many more inequalities are known. For example, Bernstein's inequality generalizes both Chernoff's and Hoeffding's (up to constant factors) using the variance of the variables X_i. McDiarmid's inequality applies to functions of the X_is other than the sum, as long as no single X_i determines the function too much. Other inequalities relax the condition that the X_is are fully independent. There are also approximate bounds other than the normal one, such as the Agresti-Coull bound, which is better for values of p close to 0 or to 1 and small n.

In the following sections, we will consider these problems in detail:

- Sampling
- Counting items and distinct items
- Frequency problems
- Counting within a sliding window
- Merging sketches computed distributedly

For readability, some technical details and references are deferred to section 4.9.

4.3 Sampling

Sampling is an obvious approach to dealing with streams: for each item in the stream, process it with probability α, and ignore it with probability $1-\alpha$.

For many tasks, including many in learning and data mining contexts, we can obtain quite good approximations while reducing computational cost by computing the function on the fraction α of sampled elements. But be warned that sampling is not a universal solution, as it does perform poorly in some

tasks. For example, for the problem of estimating the number of distinct elements, rare elements may be missed entirely and their numbers are typically underestimated.

Often, we want to reduce the number of items to be processed, yet maintain a representative sample of the stream of a given size k. *Reservoir sampling*, due to Vitter [240], is a simple and efficient technique for achieving this. It is presented in figure 4.1.

RESERVOIR SAMPLING

```
1   Init(k):
2       create a reservoir (array[0...k − 1]) of up to k items
3       fill the reservoir with the first k items in the stream
4       t ← k          ▷ t counts the number of items seen so far
5   Update(x):
6       select a random number r between 0 and t − 1
7       if r < k
8           replace the rth item in the reservoir with item x
9       t ← t + 1
10  Query():
11      return the current reservoir
```

Figure 4.1
The RESERVOIR SAMPLING sketch.

Reservoir sampling has the property that no matter how many stream elements t have been read so far, each of them is currently in the reservoir with the same probability, k/t. That is, it is equally biased toward early elements and late elements. To see that this is true, use induction on t. It is clearly true after the first t steps, for $t \leq k$. At a later time $t > k$, the probability that the tth item in the stream is in the reservoir is k/t because it is added with that probability. For item number i (with $i < t$) to be in the reservoir, (1) it must happen that it was in the reservoir at time $t - 1$, which inductively happens with probability $k/(t-1)$, and (2) it must *not* happen that the tth item is selected (probability k/t) *and* that it is placed in the position of the reservoir where the ith item is (probability $1/k$). All this occurs, as desired, with probability

$$\frac{k}{t-1} \cdot \left(1 - \frac{k}{t} \cdot \frac{1}{k}\right) = \frac{k}{t-1} \cdot \frac{t-1}{t} = \frac{k}{t} .$$

A variant called *skip counting* is better when the time for processing one item exceeds the arrival time between items. The idea is to randomly choose at time t the index $t' > t$ of the next item that will be included in the sample, and skip all records before it with no computation—in particular, without the relatively costly random number generation. Rules for randomly choosing t' given t and k can be derived so that the output of this method is probabilistically identical to that of reservoir sampling [240].

4.4 Counting Total Items

Counting the total number of items seen so far is perhaps the most basic question one can imagine on a stream. The trivial solution keeps a standard integer counter, and has the update operation just increment the counter. Typical programming languages implement integers with a fixed number of bits, such as 16, 32, or 64, which is supposed to be at least $\log t$ to count up to t.

A more economical solution is *Morris's approximate counter* [183]. It is a randomized, approximate counter that counts up to t using $\log\log t$ bits, instead of $\log t$. This means that it can count up to 10^9 with $\log\log 10^9 = 5$ bits, and to any practical number of items in 6 bits. The tradeoff is, of course, that the counting is only approximate.

The initial idea is to use uniform sampling as we have described it before: fix some probability p that is a power of 2, say 2^{-k}. Use an integer counter c as before, but upon receiving an item, increment the counter with probability p, and do nothing with probability $1 - p$. Now, the expected value of c after t items is pt, so c/p should be a good approximation of t. The sketch uses $\log(pt) = \log t - k$ bits instead of $\log t$ bits; for example, with $p = 1/8$, 3 bits are saved. The returned value will be inaccurate, though, because of the random fluctuations of c around pt.

Morris's simple but clever idea is as follows: instead of fixing k a priori and counting up to 2^k, use c in the place of k. The algorithm is shown in figure 4.2; observe that it does not mention k.

More precisely, it can be shown [103] that after reading t stream elements the expected value of 2^c is $t + 1$ and its variance is $t(t-1)/2$. That is, it returns the right value in expectation, although with a large variance. Only the value of c needs to be stored, which uses $\log c \simeq \log\log t$ bits, with an additional copy of c alive only during the *Update* procedure.

MORRIS'S COUNTER

```
1  Init():
2      c ← 0
3  Update(item):
4      increment c with probability 2^−c
5      ▷ and do nothing with probability 1 − 2^−c
6  Query():
7      return 2^c − 1
```

Figure 4.2
Morris's approximate counting algorithm.

We can reduce the variance by running several copies of the algorithm and taking averages. But in this case, there is an ad-hoc more efficient method. Intuitively, c is incremented about once every time that t doubles—that is why we have $t \simeq 2^c$; the sketch is probabilistically placing t in intervals of the form $(2^c, 2^{c+1})$. If we replace the 2s in the algorithm with a smaller base $b < 2$, intuitively we are increasing the resolution to finer intervals of the form (b^c, b^{c+1}). It can be shown [103] that the expected value of b^c is $(b-1)t+1$ and its variance $(b-1)t(t-1)/2$. The number of bits required is roughly $\log c = \log\log t - \log\log b$. For example, taking $b = 1.08$, we can count up to $t \simeq 10^9$ with 1 byte of memory and standard deviation $0.2t$.

As an example of application, Van Durme and Lall [94] addressed the problem of estimating the frequencies of k-grams in large textual corpora by using a variant of Morris's counter. The number of occurring k-grams may be huge even for moderate values of k and they have to be kept in RAM for speed, so it is important to use as few bits as possible for each counter.

4.5 Counting Distinct Elements

A more challenging problem is counting the number of *distinct* elements observed in the stream so far (also somewhat incorrectly called *unique* elements, perhaps by analogy with the Unix `uniq` command). The difficulty is that an item may appear only once and another item millions of times, but both add the same, 1, to the distinct element count.

As motivation, consider the problem of counting the number of distinct flows (source-destination IP address pairs) seen by an Internet link. The number of potential items is huge: an IPv6 address is 64 bits, so there are $2^{128} \simeq 10^{38}$ potential items, or address pairs. The direct solution is to keep some associative table, such as a hash table, and record the pairs that have actually been seen. This may quickly become too large for busy links. Fortunately, we can often trade off approximation quality for memory usage.

4.5.1 Linear Counting

Several methods for distinct element counting are variations of the classical idea of *Bloom filter*. As an example, we discuss LINEAR COUNTING, due to Whang et al. [246], which is presented in figure 4.3.

LINEAR COUNTING

Init(D, ρ):
  $\triangleright$ D is an upper bound on the number of distinct elements
  $\triangleright$ $\rho > 0$ is a load factor
  $s \leftarrow D/\rho$
  choose a hash function h from items to $\{0, \ldots, s-1\}$
  build a bit vector B of size s
Update(x):
  $B[h(x)] \leftarrow 1$
Query():
  let w be the fraction of 0s in B
  return $s \cdot \ln(1/w)$

Figure 4.3
The LINEAR COUNTING sketch.

Parameter D in the initialization is an upper bound on the number of distinct elements d in the stream, that is, we assume that $d \leq D$. A *hash function* between two sets, intuitively, maps every element x in the first set to a "random" element in the second set. The technical discussion on what exactly this means, and how to build "good enough" hash functions that can be stored in few bits is left for section 4.9.1. Let us say only that hash functions that are random enough for our purposes and map $\{0, \ldots, m-1\}$ to itself can be obtained from $O(\log m)$ truly random bits, and stored with $O(\log m)$ bits.

Each item present in the stream sets to 1 an entry of table B, the same one regardless of how many times it occurs. Intuitively, a good hash function ensures that the distinct items appearing in the stream are uniformly allocated to the entries of B. Then, the expected value returned by *Query* is the number of distinct values seen in the stream plus a tiny bias term, whose standard deviation is $\sqrt{D\rho(e^{\rho} - \rho - 1)}$. For a reasonably large number of distinct elements ($d \gg \sqrt{D}$) this is much smaller than d itself, which means that LINEAR COUNTING will be pretty accurate in relative terms.

4.5.2 Cohen's Logarithmic Counter

Cohen's counter [74] builds on this idea but moves to logarithmic space using the following observation. The average gap between two consecutive 1s in B is determined by the number of 1 bits in the stream, and is approximately s/d. This is true in particular for the first gap, or equivalently for the position of the first 1. Conveniently, it only takes $\log s$ bits to store this first position, because it is the minimum of the observed values of the hash function. The sketch is shown in figure 4.4. Cohen's counter only uses $\log D$ bits to store p.

COHEN'S COUNTER

```
1  Init(D):
2      ▷ D is an upper bound on the number of distinct elements
3      p ← D
4      choose a hash function h from items to {0, ..., D − 1}
5  Update(x):
6      p ← min( p, h(x) )
7  Query():
8      return D/p
```

Figure 4.4
Cohen's counting algorithm.

It can be shown that the expected value of p is D/d and that the *Query* procedure returns d in expectation. Its variance is large, roughly $(D/d)^2$. It can be reduced by running several copies in parallel and averaging, but this requires building several independent and "good" hash functions, which is problematic. An alternative method using a single hash function will be presented as part of the next sketch.

4.5.3 The Flajolet-Martin Counter and HyperLogLog

A series of algorithms starting with the Flajolet-Martin counter [105] and culminating in HyperLogLog [104], achieve relative approximation ϵ with even fewer bits, $O((\log\log D)/\epsilon^2)$, plus whatever is required to store a hash function. The intuition of this method is the following: let D be 2^k. Select without repetition numbers in the range $[0, 2^k - 1]$ uniformly at random; this is what we obtain after applying a good hash function to stream items. All these numbers can be written with k bits in binary; half of these numbers have 0 as the first bit, one-quarter have two leading 0s, one-eighth start with 000, and so on. In general, a leading pattern 0^i1 appears with probability $2^{-(i+1)}$, so if 0^i1 is the longest such pattern in the hash values drawn so far, then we may expect d to be around 2^{i+1}. Durand and Flajolet [93], in their improvements of Flajolet-Martin, realized that one only needs to remember k, with $\log k = \log\log D$ bits, to keep track of the longest pattern, or largest i. The basic Flajolet-Martin probabilistic counter with this improvement is given in pseudocode in figure 4.5.

PROBABILISTIC COUNTER

```
1  Init(D):
2      ▷ D is an upper bound on the number of distinct elements
3      L ← log D
4      choose a hash function h from items to {0, ..., L − 1}
5      p ← L
6  Update(x):
7      let i be such that h(x) in binary starts with 0^i 1
8      p ← min(p, i)
9  Query():
10     return 2^p / 0.77351
```

Figure 4.5
The basic Flajolet-Martin probabilistic counter.

The method needs a hash function h that transforms each item into an integer between 0 and $L - 1$, as randomly as possible. It was shown in [105] that, if d distinct items have been read, the expected value of p is $\log(0.77351\, d)$ and its standard deviation is $1.12127\ldots$ The sketch only requires memory to store p, whose value is at most $L = \log D$, and so requires $\log\log D$ bits.

The main drawback of this method is the relatively large error in the estimation, on the order of d. It can be reduced by averaging several copies, as discussed before. But this has two problems: one, it multiplies the computation time of the update operation accordingly. Two, obtaining several truly independent hash functions is hard.

Flajolet and Martin propose instead the technique they call *stochastic averaging*: Conceptually divide the stream into $m = 2^b$ substreams, each one feeding a different copy of the sketch. For each item x, use the first $b = \log m$ bits of $h(x)$ to decide the stream to which x is assigned, and the other $m - \log m$ of bits of $h(x)$ to feed the corresponding sketch. A single hash function can be used now, because items in different substreams see independent parts of the same function, and update time is essentially the same as for a single sketch. Memory use is still multiplied by m.

Different variants of the algorithm differ in how they combine the results from the m copies into a single estimation.

- LogLog [93] returns the geometric average of their estimations, which is 2 to the arithmetic average of their variables p, times a constant.
- SuperLogLog [93] removes the top 30% of the estimations before doing the above; this reduces the effect of outliers.
- HyperLogLog, by Flajolet et al. [104], returns instead the harmonic average of the m estimations, which also smooths out large outliers.

The variants above achieve standard deviations $1.3d/\sqrt{m}$, $1.05d/\sqrt{m}$, and $1.04d/\sqrt{m}$, respectively, and use memory $O(m \log\log D)$, plus $O(\log D)$ bits to store a hash function that can be shared among all instances of the counter. By using Chebyshev's inequality, we obtain an (ϵ, δ)-approximation provided that $m = O(1/(\delta\epsilon^2))$; note that it is not clear how to apply Hoeffding's bound because the estimation is not an average. Stated differently, taking $\epsilon = O(1/\sqrt{m})$, we achieve relative approximation ϵ for fixed confidence using $O((\log\log D)/\epsilon^2)$ bits of memory.

In more concrete terms, according to [104], HyperLogLog can approximate cardinalities up to 10^9 within, say, 2% using just 1.5 kilobytes of memory. A discussion of practical implementations of HyperLogLog for large applications can be found in [134].

4.5.4 An Application: Computing Distance Functions in Graphs

Let us mention a nonstreaming application of the Flajolet-Martin counter, to illustrate that streaming algorithms can sometimes be used to improve time or memory efficiency in nonstreaming problems.

Boldi et al. [47], improving on [192], proposed the HyperANF counter, which uses HyperLogLog to approximate the *neighborhood function* N of graphs. For a node v and distance d, $N(v, d)$ is defined as the number of nodes reachable from node v by paths of length at most d. The probabilistic counter allows for a memory-efficient, approximate calculation of $N(., d)$ given the values of $N(., d-1)$, and one sequential pass over the edges of the graph in the arbitrary order in which they are stored in disk, that is, with no expensive disk random accesses.

We give the intuition of how to achieve this. The first key observation is that a vertex is reachable from v by a path of length at most d if it is either reachable from v by a path of length at most $d-1$, or reachable by a path of length at most $d-1$ from a vertex u such that (v, u) is an edge. The second key observation is that HyperLogLog counters are mergeable, a concept we will explore in section 4.8. It means that, given two HyperLogLog counters H_A and H_B that result from processing two streams A and B, there is a way to obtain a new HyperLogLog counter $\mathit{merge}(H_A, H_B)$ identical to the one that would be obtained by processing the concatenation of C of A and B. That is, it contains the number of distinct items in C. Then, suppose inductively that we have computed for each vertex v a HyperLogLog counter $H_{d-1}(v)$ that (approximately) counts the number of vertices at distance at most $d-1$ from v, or in other words, approximates $N(v, d-1)$. Then compute $H_d(v)$ as follows:

1. $H_d(v) \leftarrow H_{d-1}(v)$.
2. For each edge (v, u) (in arbitrary order) do

$$H_d(v) \leftarrow \mathit{merge}(H_d(v), H_{d-1}(u)).$$

Using the first key observation above, it is clear that $H_d(v)$ equals $N(v, d)$ if the counts are exact; errors due to approximations might in principle accumulate and amplify as we increase d, but experiments in [47] do not show this effect. We can discard the H_{d-1} counters once the H_d are computed, and reuse the space they used. Thus we can approximate neighborhood functions up to distance d with d sequential scans of the edge database and memory $O(n \log \log n)$.

Using this idea and a good number of clever optimizations, Backstrom et al. [21] were able to address the estimation of distances in the Facebook graph, the graph where nodes are Facebook users and edges their direct friendship relationships. The graph had 721 million users and 69 billion links. Remarkably, their computation of the average distance between users ran in a few hours on a single machine, and resulted in average distance 4.74, corresponding to 3.74 intermediaries or "degrees of separation." Informally, this confirms the celebrated experiment by S. Milgram in the 1960s [177], with physical letters and the postal service, hinting that most people in the world were at the time no more than six degrees of separation apart from each other.

A unified view of sketches for distance-related queries in massive graphs is presented in [75].

4.5.5 Discussion: Log vs. Linear

More details on algorithms for finding distinct elements can be found in [176], which also provides empirical evaluations. It focuses on the question of when it is worth using the relatively sophisticated sublinear sketches rather than simple linear counting. The conclusion is that linear sketches are preferable when extremely high accuracy, say below $\epsilon = 10^{-4}$, is required. Logarithmic sketches typically require $O(1/\epsilon^2)$ space for multiplicative error ϵ, which may indeed be large for high accuracies.

In data mining and ML applications, relatively large approximation errors are often acceptable, say within $\epsilon = 10^{-1}$, because the model mined is the combination of many statistics, none of which is too crucial by itself, and because anyway data is viewed as a random sample of reality. Thus, logarithmic space sketches may be an attractive option.

Also worthy of mention is that [144] gives a more involved, asymptotically optimal algorithm for counting distinct elements, using $O(\epsilon^{-2} + \log d)$ bits, that is, removing the $\log \log d$ factor in HyperLogLog. HyperLogLog still seems to be faster and smaller for practical purposes.

4.6 Frequency Problems

In many cases, we do not simply want to count all items in a stream, but separately count different types of items in the stream. Estimating the frequencies of all items is impossible to do in general in sublinear memory, but often it suffices to have approximate counts for the most frequent items.

For an initial segment of length t of a stream, the *absolute frequency* of an item x is the number of times it has appeared in the segment, and its *relative frequency* is the absolute frequency divided by t. We will mostly consider the following formulation of the problem:

The heavy hitter problem. Given a threshold ϵ and access to a stream, after reading any number of items from the stream, produce the set of *heavy hitters*, all items whose relative frequency exceeds ϵ.

This problem is closely related to that of computing *iceberg queries* [97] in the database area. A basic observation is that there can be at most $1/\epsilon$ items having relative frequency above ϵ; otherwise relative frequencies add to more than 1, which is impossible.

A first approximation to the heavy hitter problem is to use sampling; for example, use reservoir sampling to draw a uniform sample from the stream, and at all times declare that the frequent elements in the sample are the frequent elements in the stream. This is a valid algorithm, but the sample size required to achieve any fixed confidence is proportional to $1/\epsilon^2$. In this section we will present three algorithms for the heavy hitter problem, each with its own strong points. The first is deterministic and based on counters; the other two are based on hashing.

The surveys [80, 164] discuss and evaluate the algorithms we are going to present, and several others.

In this section we measure memory in words, assuming that an item, integer, pointer, and such simple variables fit in one memory word.

4.6.1 The SPACESAVING Sketch

Counter-based algorithms are historically evolutions of a method by Boyer and Moore [50] to find a majority element (one with frequency at least 1/2) when it exists. An improvement of this method was independently discovered by Misra and Gries [179], Demaine et al. [86], and Karp et al. [145]. Usually called FREQUENT now, this method finds a list of elements guaranteed to contain all ϵ–heavy hitters with memory $O(1/\epsilon)$ only.

A drawback of FREQUENT is that it provides the heavy hitters but no reliable estimates of their frequencies, which are often important to have. Other counter-based sketches that do provide approximations of the frequencies include Lossy Counting and Sticky Sampling, both due Manku and Motwani [165, 166], and Space Saving, due to Metwally et al. [175]. We describe Space Saving, reported in [80, 161, 164] to have the best performance among

several heavy hitter algorithms. Additionally, it is simple and has intuitive rigorous guarantees on the quality of approximation.

Figure 4.6 shows the pseudocode of SPACESAVING. It maintains in memory a set of at most k pairs (item, count), initialized with the first k distinct elements and their counts. When a new stream item arrives, if the item already has an entry in the set, its count is incremented. If not, this item replaces the item with the lowest count, which is then incremented by 1. The nonintuitive part is that this item inherits the counts of the element it evicts, even if this is its first occurrence. However, as the following claims show, the error is never too large, and can neither make a rare element look too frequent nor a true heavy hitter look too infrequent.

SPACESAVING

Init(k):
    create an empty set S of pairs (item,count) able to hold up to k pairs
Update(x):
    if item x is in S
        increase its count by 1
    else if S contains less than k entries
        add $(x, 1)$ to S
    else
        let $(y, count)$ be an entry in S with lowest count
        replace the entry $(y, count)$ with $(x, count + 1)$ in S
Query():
    return the list of pairs $(x, count(x))$ currently in S

Figure 4.6
The SPACESAVING sketch.

Let $f_{t,x}$ be the absolute frequency of item x among the first t stream elements, and if x is in the sketch at time t, let $count_t(x)$ be its associated count; we omit the subindex t when it is clear from the context. The following are true for every item x and at all times t:

1. The value of the smallest counter min is at most t/k.
2. If $x \notin S$, then $f_x \leq min$.
3. If $x \in S$, then $f_x \leq count(x) \leq f_x + min$.

(1) and (2) imply that every x with frequency above t/k is in S, that is, the sketch contains all $(1/k)$–heavy hitters. (3) says that the counts for elements in S do not underestimate the true frequencies and do not overestimate them by more than t/k. The value t/k can be seen as the minimum resolution: counts are good approximations only for frequencies substantially larger than t/k, and an item with frequency 1 may have just made it into the sketch and have associated count up to t/k.

The *Stream-Summary* data structure proposed in [175] implements the sketch, ensuring amortized constant time per update. For every count value c present in the sketch, a bucket is created that contains c and a pointer to a doubly linked list of all the items having count c. Buckets are kept in a doubly linked list, sorted by their values c. The nodes in the linked lists of items all contain a pointer to their parent bucket, and there is a hash table that, given an item, provides a pointer to the node that contains it. The details are left as an exercise.

All in all, taking $k = \lceil 1/\epsilon \rceil$, SPACESAVING finds all ϵ–heavy hitters, with an approximation of their frequencies that is correct up to ϵ, using $O(1/\epsilon)$ memory words and constant amortized time.

SPACESAVING, FREQUENT, Lossy Counting, and Sticky Sampling are all deterministic algorithms; they do not use hash functions or randomization, except perhaps inside the hash table used for searching the set. It is easy to adapt SPACESAVING to updates of the form (x, c), where x is an item, c any *positive* weight, and the pair stands for the assignment $f_x \leftarrow f_x + c$, with no significant impact on memory or update time. But it cannot simulate item deletions or, equivalently, updates with *negative* weights.

4.6.2 The CM-Sketch Algorithm

The *Count-Min sketch* or *CM-sketch* is due to Cormode and Muthukrishnan [81] and can be used to solve several problems related to item frequencies, among them the heavy hitter problem. Unlike FREQUENT and SPACESAVING, it is probabilistic and uses random hash functions extensively. More importantly for some applications, it can deal with updates having positive and negative weights, in particular, with item additions and subtractions.

We first describe what the sketch achieves, then its implementation. We consider streams with elements of the form (x, c) where x is an item, c any positive or negative weight update to f_x. We use F to denote the total weight $\sum_x f_x$ at a given time.

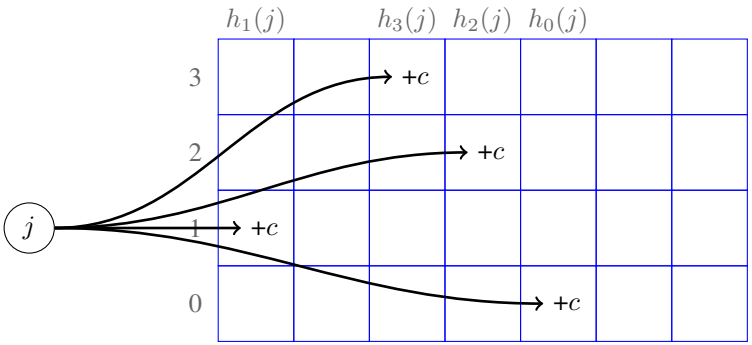


Figure 4.7
Example CM-sketch structure of width 7 and depth 4, corresponding to $\epsilon = 0.4$ and $\delta = 0.02$.

The CM-sketch creation function takes two parameters, the *width* w and the *depth* d. Its update function receives a pair (x, c) as above. Its *Query* function takes as parameter an item x, and returns a value $\hat{f}_x$ that satisfies, with probability $1 - \delta$, that

$$f_x \leq \hat{f}_x \leq f_x + \epsilon F \tag{4.1}$$

provided $w \geq \lceil \mathrm{e}/\epsilon \rceil$ and $d \geq \lceil \ln(1/\delta) \rceil$. This holds at any time, no matter how many items have been added and then deleted before.

We assume from now on that $F > 0$ for the inequality above to make sense, that is, the negative weights are smaller than the positive weight. Like in SPACESAVING, this means that $\hat{f}_x$ is a reasonable approximation of f_x for heavy items, that is, items whose weight is a noticeable fraction of F. Additionally, the memory used by the algorithm is approximately wd counters or accumulators. In summary, values f_x are approximated within an additive factor of ϵ with probability $1 - \delta$ using $O(\ln(1/\delta)/\epsilon)$ memory words.

Let us now describe the implementation of the sketch. A CM-sketch with parameters w, d consists of a two-dimensional array of counters with d rows and w columns, together with d independent hash functions $h_0, \ldots, h_{d-1}$ mapping items to the interval $[0 \ldots w - 1]$. Figure 4.7 shows an example of the CM-sketch structure for $d = 4$ and $w = 7$, and figure 4.8 shows the pseudocode of the algorithm.

It can be seen that both *Update* and *Query* operations perform $O(d)$ operations. We do not formally prove the approximation bounds in inequality (4.1), but we provide some intuition of why they hold. Observe first that for a row i and column j, $A[i, j]$ contains the sum of the weights in the stream of all

CM-SKETCH

Init(w, d):
    create a $d \times w$ array of counters A, all initialized to 0
    choose d hash functions $h_0, \ldots, h_{d-1}$ from items to $\{0, \ldots, w-1\}$
Update(x, c):
    for $i \leftarrow 0 \ldots d-1$
        $A[i, h_i(x)] \leftarrow A[i, h_i(x)] + c$
Query(x):
    return $\min\{A[i, h_i(x)] : i = 0 \ldots d-1\}$

Figure 4.8
The CM-SKETCH algorithm.

items x such that $h_i(x) = j$. This alone ensures that $f_x \leq A[i, h_i(x)]$ and therefore that $f_x \leq$ *Query*(x). But $A[i, h_i(x)]$ may overestimate f_x if there are other items y that happen to be mapped to the same cell of that row, that is $h_i(x) = h_i(y)$. The total weight to be distributed among the w cells in the row is F, so the expected weight falling on any particular cell is F/w. One can show that it is unlikely that any cell receives much more than this expected value. Intuitively, this may happen if a particularly heavy item is mapped in the cell, but the number of heavy items is limited, so this happens rarely; or else if too many light items are mapped to the cell, but this is also unlikely because the hash function places items more or less uniformly among all w cells in the row. Markov's inequality is central in the proof.

Returning to the heavy hitter problem, the CM-sketch provides a reasonable approximation of the frequency of elements that are heavy hitters, but there is no obvious way to locate them from the data structure. A brute-force search that calls *Query*(x) for all possible x is unfeasible for large item universes. The solution uses an efficient implementation of so-called *Range-Sum queries*, which are of independent interest in many situations.

A Range-Sum query is defined on sets of items that can be identified with integers in $\{1, \ldots, |I|\}$. The range-sum query with parameters $[x, y]$ returns $f_x + f_{x+1} + \cdots + f_{y-1} + f_y$. As an example, consider a router handling packets destined to IP addresses. A range-sum query for the router could be "How many packets directed to IPv4 addresses in the range 172.16.aaa.bbb have you

seen?" We expect to solve this query faster than iterating over the 2^{16} addresses in the range.

Range-sum queries can be implemented using CM-sketches as follows: Let $|I| = 2^n$ be the number of potential items. For every power of two 2^i, $0 \leq i \leq n$, create a CM-sketch C_i that range-sums items in intervals of size 2^i. More precisely, *Query*(j) to C_i will return the total count of items in the interval $[j2^i \ldots (j+1)2^i - 1]$. Alternatively, we can imagine that at level i each item x belongs to the "superitem" $x \bmod 2^{n-i}$. In particular, C_0 stores individual item counts and C_n the sum of all counts. Then we can answer the range-sum query (x, y) to C_i, for x and y in the same interval of size 2^i, with the following recursion: If x and y fall in the same interval of size 2^{i-1}, we query (x, y) to C_{i-1} and return its answer. If, on the other hand, there is some j such that $x < j2^{i-1} \leq y$ (that is, x and y do not belong to the same interval of size 2^{i-1}), we ask the range-sum queries $(x, j2^{i-1} - 1)$ and $(j2^{i-1}, y)$ to C_{i-1}, and then return the sum of the answers. Sketch C_0 is the base of the recursion.

Now, the heavy hitter problem with parameter k can be solved by exploring the set of intervals $[j2^i, (j+1)2^i - 1]$ by depth. If the weight of the interval is below F/k, no leaf below it can be a heavy hitter, so the empty list is returned. Otherwise, it is divided into two equal-size intervals that are explored separately, and the lists returned are concatenated. When a leaf is reached, it is returned if and only if it is a heavy hitter, that is, its weight is F/k or more. Also, at each level of the tree there can be at most k nodes of weight F/k or more, so at most $k \log |I|$ nodes of the tree are explored. All in all, heavy hitters are found using $O((\log |I|) \ln(1/\delta)/\epsilon)$ memory words.

As already mentioned, the CM-sketch can be used for solving several other problems on streams efficiently [81], including quantile computations, item frequency histograms, correlation, and the inner product of two streams. For the case of quantile computation, however, the specialized sketch FrugalStreaming [163] is considered the state of the art. We will describe another quantile summary in section 6.4.3 when describing the management of numeric attributes in classification problems.

4.6.3 CountSketch

The CountSketch algorithm was proposed by Charikar et al. [64]. Like CM-sketch, it supports updates with both positive and negative weights. It

uses memory $O(1/\epsilon^2)$, but it finds the so-called F_2–heavy hitters, which is an interesting superset of the regular heavy hitters, or F_1–heavy hitters.

The F_2 norm of a stream is defined as $\sum_{x \in I} f_x^2$. Item x is an F_2–heavy hitter with threshold ϵ if $f_x \geq \epsilon\sqrt{F_2}$. Some easy algebra shows that every F_1–heavy hitter is also an F_2–heavy hitter.

In its simplest form, COUNTSKETCH can be described as follows. It chooses two hash functions. Function h maps items to a set of w buckets $\{0, \ldots, w-1\}$. Function σ maps items to $\{-1, +1\}$. Both are assumed to be random enough. In particular, an important property of σ is that for different x and y, $\sigma(x)$ and $\sigma(y)$ are independent, which means that $\Pr[\sigma(x) = \sigma(y)] = 1/2$, which means that $\mathrm{E}[\sigma(x) \cdot \sigma(y)] = 0$.

The algorithm keeps an array A of w accumulators, initialized to 0. For every pair (x, c) in the stream, it adds to $A[h(x)]$ the value $\sigma(x) \cdot c$. Note that updates for a given x are either all added or all subtracted, depending on $\sigma(x)$. When asked to provide an estimation of f_x, the algorithm returns $\mathit{Query}(x) = \sigma(x) \cdot A[h(x)]$. This estimation would be exact if no other item y was mapped to the same cell of A as x, but this is rarely the case. Still, one can prove that the expectation over random choices of the hash functions is the correct one. Indeed:

$$\begin{aligned}\mathrm{E}[\mathit{Query}(x)] &= \mathrm{E}[\,\sigma(x) \cdot \sum_{y:h(y)=h(x)} \sigma(y) f_y\,] \\ &= \mathrm{E}[\,\sigma(x)^2 f_x] + \sum_{y:h(y)=h(x), y \neq x} \sigma(x)\sigma(y) f_y\,] = f_x\end{aligned}$$

because, as mentioned, $E[\sigma(x)\sigma(y)] = 0$ for $x \neq y$, and $\sigma(x)^2 = 1$. Similar algebra shows that $\mathrm{Var}(\mathit{Query}(x)) \leq F_2/w$.

Now we can use a strategy like the one described in section 4.9.2 to improve first accuracy, then confidence: take the average of $w = 12/(5\epsilon^2)$ independent copies; by Chebyshev's inequality, this estimates each frequency with additive error $\epsilon/2 \cdot \sqrt{F_2}$ and probability $5/6$. Then do this $d = O(\ln(1/\delta))$ times independently and return the median of the results; Hoeffding's inequality shows (see exercise 4.9) that this lifts the confidence to $1 - \delta$. Therefore, the set of x's such that $\mathit{Query}(x) > \epsilon/2 \cdot \sqrt{F_2}$ contains each F_2–heavy hitter with high probability. Note that, unlike in CM-SKETCH and SPACESAVING, there may be false negatives besides false positives, because the median rather than the min is used. Memory use is $O(\ln(1/\delta)/\epsilon^2)$ words.

At this point, we can view the d copies of the basic algorithm as a single algorithm that resembles CM-SKETCH: it chooses hash functions

$h_0, \ldots, h_{d-1}$, $\sigma_0, \ldots, \sigma_{d-1}$, and maintains an array A with d rows and w columns. The pseudocode is shown in figure 4.9. To recover the heavy hitters efficiently, COUNTSKETCH also maintains a heap of the most frequent elements seen so far on the basis of the estimated weights.

COUNTSKETCH

```
1  Init(w, d):
2      create a d × w array of counters A, all initialized to 0
3      choose d hash functions h_0, ..., h_{d-1} from items to {0, ..., w − 1}
4      choose d hash functions σ_0, ..., σ_{d-1} from items to {−1, +1}
5      create an empty heap
6  Update(x, c):
7      for i ← 0 ... d − 1
8          A[i, h_i(x)] ← A[i, h_i(x)] + σ(i) · c
9          if x is in the heap, increment its count there by c
10         else if Query(y) < Query(x) where y has smallest weight in the heap
11             remove y from the heap and add (x, c) to the heap
12 Query(x):
13     return median{A[i, h_i(x)] : i = 0 ... d − 1}
```

Figure 4.9
The COUNTSKETCH algorithm.

Improvements of COUNTSKETCH are presented in [51, 249].

As a brief summary of the heavy hitter methods, and following [164], if an application is strictly limited to discovering frequent items, counter-based techniques such as Lossy Counting or Space Saving are probably preferable in time, memory, and accuracy, and are easier to implement. For example, COUNTSKETCH and CM-SKETCH were found to be less stable and perform worse for some parameter ranges. On the other hand, hash-based methods are the only alternative to support negative item weights, and provide useful information besides simply the heavy hitters.

4.6.4 Moment Computation

Frequency moments are a generalization of item frequency that give different information on the distribution of items in the stream. The pth moment of a

stream where each item x has absolute frequency f_x is defined as

$$F_p = \sum_x f_x^p.$$

Observe that F_1 is the length of the stream and that F_0 is the number of distinct items it contains (if we take 0^0 to be 0, and $f^0 = 1$ for any other f). F_2 is of particular interest, as it can be used to compute the variance of the items in the stream. The limit case is F_∞, defined as

$$F_\infty = \lim_{p\to\infty} (F_p)^{1/p} = \max_x f_x$$

or, in words, the frequency of the most frequent element.

Alon et al. [14] started the study of streaming algorithms for computing approximations of frequency moments. Their sketch for $p \geq 2$ (known as the Alon-Matias-Szegedy or AMS sketch) uses a technique similar to the one in CM-SKETCH: take hash functions that map each item to $\{-1, +1\}$, use them to build an unbiased estimator of the moment with high variance, then average and take medians of several copies to produce an arbitrarily accurate estimation.

Their work already indicated that there was an abrupt difference in the memory requirements for computing F_p for $p \leq 2$ and for $p > 2$. After a long series of papers it was shown in [141] that, considering relative approximations,

- for $p \leq 2$, F_p can be approximated in a data stream using $O(\epsilon^{-2}(\log t + \log m)\ln(1/\delta))$ bits of memory, and
- for $p > 2$, F_p can be approximated in a data stream using $O(\epsilon^{-2}\, t^{1-2/p} \ln(1/\delta))$ bits,

up to logarithmic factors, where t is the length of the stream and $m = |I|$ is the size of the set of items. Furthermore, memory of that order is in fact *necessary* for every p other than 0 and 1. Later work has essentially removed the logarithmic gaps as well. In particular, for F_∞ or the maximum frequency, linear memory in the length of the stream is required. Observe that for the special cases $p = 0$ and $p = 1$ we have given algorithms using memory $O(\log\log t)$ rather than $O(\log t)$ in previous sections.

4.7 Exponential Histograms for Sliding Windows

Often, more recent items in the stream should be considered more important for analysis. We will model this idea in several ways in chapter 5, but here we

consider the simplest model, the *sliding window*: At all times, we only consider relevant the last W items received, where W is the window size; all previous items are irrelevant. Alternatively, we fix a time duration T (such as one week, one hour, or one minute) and consider only the items that have arrived within time T from now. We will consider mostly the first model, but adaptation to the second is easy.

Consider the problem of maintaining the sum of the last W values in a stream of real numbers. The natural solution is to use a circular buffer of size W; every arriving element is added to the sum; it evicts the oldest element from the buffer, which is subtracted from the sum. The scheme has constant update time, but uses memory W. In fact, one can show that any streaming algorithm that *exactly* solves this problem must use memory at least W.

We will see next a method for maintaining an *approximate* value of the sum storing only $O(\log W)$ values. The *Exponential Histogram* sketch by Datar et al. [83] maintains approximations of aggregate functions on sliding windows of real numbers. We describe in detail only the case in which the aggregate function is the sum and the items are bits.

The sketch partitions the length-W window into a sequence of subwindows, and represents each subwindow by a *bucket*. Intuitively, buckets encode the desired window of size W where older 1s are represented with less and less resolution. Each bucket has a *capacity*, which is a power of 2, and represents a subwindow containing as many 1s as its capacity. It contains an integer value, the *timestamp*, which is the timestamp of the most recent 1 in the subwindow it represents. For a parameter $k > 1$, there are k buckets of capacity 1, delimited by the most recent k 1s, k buckets of capacity 2, k buckets of capacity 4, and so on, up to k buckets of capacity 2^m, where m is the smallest value ensuring that there are enough buckets to hold W items, that is,

$$k \sum_{i=0}^{m-1} 2^i < W \leq k \sum_{i=0}^{m} 2^i.$$

Observe that $m \simeq \log(W/k)$, and so there are $km \simeq k \log(W/k)$ buckets.

The pseudocode of the method is given in figure 4.11 and an example of a sliding window split into buckets is given in figure 4.10. The example shows the partition of the last 29 bits of a stream into 6 buckets at some time t, with $k = 2$. Each bucket contains as many 1s as its capacity. The most recent bit is on the right, with timestamp t. The least recent bit is on the left, with timestamp $t - 28$. The timestamp of each bucket is the timestamp of its most recent 1. If the desired sliding window size W is 25, the true count of 1s within

the most recent W bits is 11, as there are 11 1s with timestamps between $t-24$ and t. The sketch will report the sum of the window to be 12: the total number of 1s in the buckets (14), minus half the capacity of the oldest bucket (2). Therefore the relative error is $(12-11)/11 \simeq 9\%$.

Bucket:	1011100	10100101	100010	11	10	1000
Capacity:	4	4	2	2	1	1
Timestamp:	$t-24$	$t-14$	$t-9$	$t-6$	$t-5$	$t-3$

Figure 4.10
Partitioning a stream of 29 bits into buckets, with $k=2$. Most recent bits are to the right.

EXPONENTIAL HISTOGRAMS

```
1  Init(k, W):
2      t ← 0
3      create a list of empty buckets
4  Update(b):
5      t ← t + 1
6      if b = 1    ▷ do nothing with 0s
7          let t be the current time
8          create a bucket of capacity 1 and timestamp t
9          i ← 0
10         while there are more than k buckets of capacity 2^i
11             merge the two oldest buckets of capacity 2^i into a
12                 bucket of capacity 2^(i+1): the timestamp of the new bucket
13                 is the largest (most recent) of their two timestamps
14             i ← i + 1
15         remove all buckets whose timestamp is ≤ t − W
16 Query():
17     return the sum of the capacities of all existing buckets
18         minus half the capacity of the oldest bucket
```

Figure 4.11
The EXPONENTIAL HISTOGRAMS sketch.

The error in the approximation is introduced only by the oldest bucket. It must contain at least one 1 with timestamp greater than $t-W$, otherwise it would have been dropped, but possibly also up to 2^m-1 1s whose timestamp

is at most $t - W$ and that should not be counted by an exact method. The relative error is therefore at most half the capacity of the oldest bucket over the capacity of all buckets. Since, after some transient, there are either $k - 1$ or k buckets of each size except the largest one, the relative error is at most $(2^m/2)/(2^m + (k-1)(2^{m-1} + \cdots + 1)) \simeq 1/2k$.

If the desired approximation rate is ϵ, it suffices to set $k = 1/2\epsilon$. The number of buckets is then $(\log W)/2\epsilon$. Each bucket contains a timestamp, which is in principle an unbounded number. However, we can reduce the timestamps to the interval $0 \ldots W - 1$ by counting mod W, so a timestamp has $\log W$ bits, giving a total of $O((\log W)^2/\epsilon)$ bits of memory. Update time is $O(\log W)$ in the worst case if the cascade of bucket updates affects all buckets, but this happens rarely; amortized time is $O(1)$. A variant of exponential histograms called *deterministic waves* achieves worst-case $O(1)$ time [122].

The sketch can be adapted to work for nonnegative numbers in a range $[0, B]$ multiplying memory by $\log B$, and also to maintain approximations of other aggregates such as variance, maximum, minimum, number of distinct elements, and histograms [83].

4.8 Distributed Sketching: Mergeability

Most of the sketches discussed so far have the property called *mergeability*, which makes them appropriate for distributed computation. We say that a sketch type is *mergeable* if two sketches built from data streams D_1 and D_2 can be combined efficiently into another sketch of the same type that can correctly answer queries about interleavings of D_1 and D_2. Note that in problems that only depend on item frequencies, the answer is the same for all interleavings of the streams. For other problems, the definition of mergeability can be more complex.

As a simple example, consider a "sketch" consisting of an integer that counts the number of items in the stream. Now we build two such sketches from two streams. If we add their values, the result is—correctly—the number of items in any stream obtained by merging the original two.

Less trivially, taking the OR is a correct merging operation for Linear Counters. For Cohen's and Flajolet-Martin counters and variants, take the minimum of their stored values, componentwise if we use several copies to reduce the variance. For the CM-sketch, add the two arrays of counters componentwise. In these cases, the two sketches to be merged must use the same hash

functions. The cases of SPACESAVING and EXPONENTIAL HISTOGRAMS are left for the exercises.

Mergeability allows for distributed processing as follows: suppose we have k different computing nodes, each receiving a different stream—or equivalently some fraction of a large, distributed stream. A central node that wants to build a sketch for this large implicit stream can periodically request the k nodes to send their sketches, and merge them into its own global sketch.

For Morris's counter, Cohen (section 7 of [75]) presents a merging method; it is not as trivial as the ones above, particularly the correctness analysis.

4.9 Some Technical Discussions and Additional Material

4.9.1 Hash Functions

For several sketches we have assumed the existence of hash functions that randomly map a set of items I to a set B. Strictly speaking, a *fully independent* hash function h should be such that the value of h on some value x cannot be guessed with any advantage given the values of h on the rest of the elements of I; formally, for every i,

$$\Pr[h(x_i)|h(x_1),\ldots,h(x_{i-1}),h(x_{i+1}),\ldots,h(x_{|I|})] = \Pr[h(x_i)] = 1/|I|.$$

But it is incorrect to think of $h(x_i)$ as generating a new random value each time it is called—it must always return the same value on the same input. It is thus random, but reproducibly random. To create such an h we can randomly guess and store the value of $h(x_i)$ for each $x_i \in I$, but storing h will use memory proportional to $|I|\log|B|$, defeating the goal of using memory far below $|I|$.

Fortunately, in some cases we can show that weaker notions of "random hash function" suffice. Rather than using a fixed hash function, whenever the algorithm needs to use a new hash function, it will use a few random bits to choose one at random from a *family* of hash functions H. We say that family H is *pairwise independent* if for any two distinct values x and y in I and α, β in B we have

$$\Pr[h(x) = \alpha|h(y) = \beta] = \Pr[h(x) = \alpha] = 1/|I|,$$

where the probability is taken over a random choice of $h \in H$. This means that if we do not know the h chosen within H, and know only one value of h, we cannot guess any other value of h.

Here is a simple construction of a family of pairwise independent hash functions when $|I|$ is some prime p. View I as the interval $[0 \ldots p-1]$. A function h in the family is defined by two values a and b in $[0 \ldots p-1]$ and computes $h(x) = (ax + b) \bmod p$. Note that one such function is described by a and b, so with $2 \log p$ bits. This family is pairwise independent because given x, y and the value $h(y)$, for each possible value $h(x)$ there is a unique solution (a, b) to the system of equations $\{h(x) = ax + b, h(y) = ay + b\}$, so all values of $h(x)$ are equally likely. Note however that if we know x, y, $h(x)$, and $h(y)$, we can solve uniquely for a, b so we know the value $h(z)$ for every other input z. If $|I|$ is a prime power p^b, we can generalize this construction by operating in the finite field of size p^b; be warned that this is *not* the same as doing arithmetic modulo p^b. We can also choose a prime slightly larger than $|I|$ and get functions that deviate only marginally from pairwise independence.

Pairwise independent functions suffice for the CM-sketch. Algorithms for moments require a generalization called 4-wise independence. For other algorithms, such as HyperLogLog, k-wise independence cannot be formally shown to suffice, to the best of our knowledge. However, although no formal guarantee exists, well-known hash functions such as MD5, SHA1, SHA256, and Murmur3 seem to pose no problem in practice [104, 134]. Theoretical arguments have been proposed in [71] for why simple hash functions may do well on data structure problems such as sketching.

4.9.2 Creating (ϵ, δ)-Approximation Algorithms

Let f be a function to be approximated, and g a randomized algorithm such that $\mathrm{E}[g(x)] = f(x)$ for all x, and $\mathrm{Var}(g(x)) = \sigma^2$. Run k independent copies of g, say $g_1, \ldots, g_k$, and combine their outputs by averaging them, and let h denote the result. We have $\mathrm{E}[h(x)] = f(x)$ and, by simple properties of the variance, $\mathrm{Var}(|f(x) - h(x)|) \leq \mathrm{Var}(f(x) - h(x)) = \mathrm{Var}(h(x)) = \sigma^2/k$. Then by Chebyshev's inequality we have $|f(x) - h(x)| > \epsilon$ with probability at most $\sigma^2/(k\epsilon^2)$, for every ϵ. This means that h is an (ϵ, δ)-approximation of f if we choose $k = \sigma^2/(\epsilon^2\delta)$. The memory and update time of h are k times those of g.

A dependence of the form σ^2/ϵ^2 to achieve accuracy ϵ is necessary in general, but the dependence of the form $1/\delta$ can be improved as follows: use the method above to achieve a fixed confidence, say, $(\epsilon, 1/6)$-approximation. Then run ℓ copies of this fixed confidence algorithm and take the *median* of the results. Using Hoeffding's inequality we can show that this new algorithm is

an (ϵ, δ)-approximation if we take $\ell = O(\ln(1/\delta))$; see exercise 4.9. The final algorithm therefore uses $k \cdot \ell = O((\sigma^2/\epsilon^2)\ln(1/\delta))$ times the memory and running time of the original algorithm g.

This transformation is already present in the pioneer paper on moment estimation in streams by Alon et al. [14].

4.9.3 Other Sketching Techniques

The sketches presented in this chapter were chosen for their potential use in stream mining and learning on data streams. Many other algorithmic problems on data streams have been proposed, for example solving combinatorial problems, geometric problems, and graph problems. Also, streaming variants of deep algorithmic techniques such as wavelets, low-dimensionality embeddings, and compressed sensing have been omitted in our presentation. Recent comprehensive references include [79, 172].

Algorithms for linear-algebraic problems deserve special mention, given how prominent they are becoming in ML. Advances in online Singular Value Decomposition ([221] and the more recent [159]) and Principal Component Analysis [49] are likely to become highly influential in streaming. The book [248] is an in-depth presentation of sketches for linear algebra; their practical use in stream mining and learning tasks is a very promising, but largely unexplored, direction.

4.10 Exercises

Exercise 4.1 We want to estimate the fraction f of items in a stream that satisfy some boolean property P. We do not have access to the stream itself, but only to a reservoir sampling of it, of capacity k. Let g be the fraction of items in the reservoir that satisfy P. Give the accuracy ϵ of the approximation g to f as a function of k if the desired confidence is 95%. (*Hint*: use Hoeffding's bound.)

Exercise 4.2 Suppose that in Morris's counter we change the threshold 2^{-c} to the exponentially smaller threshold 2^{-2^c}. (1) Argue informally that we obtain an algorithm that can count up to t using memory $O(\log\log\log t)$. (2) Explain why this algorithm is not interesting in practice.

Exercise 4.3 Give pseudocode for (or better, implement) the SPACESAVING algorithm using the Stream Summary data structure outlined in the text.

Exercise 4.4 Complete the proof by induction of the approximation guarantees (1)–(3) given for SPACESAVING.

Exercise 4.5 Give an algorithm for merging two SPACESAVING sketches of the same size k into a sketch of size k', where $k \leq k' \leq 2k$. Its runtime should be proportional to k, not to the sum of the counts contained in either of the sketches.

Exercise 4.6 Give pseudocode for (or better, implement) the range-sum query algorithm and the heavy hitter algorithm based on CM-SKETCH.

Exercise 4.7 Suppose that we change the Exponential Histograms (for the worse) as follows: 0 bits are now not ignored, but added to the bucket. A bucket, besides a timestamp, has a counter m of the 1s it contains (the number of 0s being its capacity minus m). Explain how buckets should be merged, and argue that querying the sketch returns an *additive* ϵ-approximation of the fraction of 1s in the sliding window of size W, instead of a multiplicative one.

Exercise 4.8 Give a procedure to merge two Exponential Histograms for the sum of two bit streams. The two input sketches and the output sketch must all have the same window length W and the same parameter k. You can start by assuming that the sketches are synchronized, that is, they represent windows with the same set of timestamps. An extension is to do without this assumption.

Exercise 4.9 Complete the proof that the construction of an (ϵ, δ)-approximation algorithm from any approximation algorithm is correct.

First show the following: suppose that a real-valued random variable X satisfies

$$\Pr[a \leq X \leq b] \geq 5/6.$$

Then the median Z of ℓ independent copies of X, say $X_1, \ldots, X_\ell$, satisfies

$$\Pr[a \leq Z \leq b] \geq 1 - 2\exp(-2\ell/9).$$

To show this, follow these steps:

- For each $i \leq \ell$, define an indicator variable Z_i as 1 if the event "$a \leq X_i$" occurs, and 0 otherwise.
- State the relation between $\sum_{i=1}^{\ell} Z_i$ and the event "$Z < a$."
- Use Hoeffding's inequality to bound the probability that $Z < a$.
- Proceed similarly to bound the probability that $Z > b$, and conclude the proof.

Now take $k = 6\sigma^2/\epsilon^2$ and create an algorithm h that averages k copies of the given approximation algorithm g for f. As shown in the text, h is an $(\epsilon, 1/6)$-approximation algorithm for f. Show that the algorithm that returns the median of ℓ independent copies of h is an (ϵ, δ)-approximation algorithm of f for $\ell = 9/2 \cdot \ln(2/\delta)$. This gives $k\ell = 27\sigma^2/\epsilon^2 \cdot \ln(2/\delta)$.

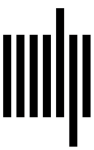
Machine Learning for Data Streams

with Practical Examples in MOA

By: Albert Bifet, Ricard Gavaldà, Geoffrey Holmes, Bernhard Pfahringer

Citation:

Machine Learning for Data Streams: with Practical Examples in MOA
By: Albert Bifet, Ricard Gavaldà, Geoffrey Holmes, Bernhard Pfahringer
DOI: 10.7551/mitpress/10654.001.0001
ISBN (electronic): 9780262346047
Publisher: The MIT Press
Published: 2023



The MIT Press

This book was set in Times Roman and Mathtime Pro 2 by the authors.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data is available

ISBN: 978-0-262-03779-2

10 9 8 7 6 5 4 3 2 1