Lecture 11: Count-Min Sketch and HyperLogLog Counting

Sam McCauley

Written April 20, 2020; Last Edited April 20, 2020

1 Goal for Today

Computers are increasingly often tasked with dealing with *extremely* large amounts of data. This is particularly true on the internet: every user, every action, every message is a piece of data that needs to be considered and processed very quickly.

In the last class, we learned about Bloom filters, which allow us to compress data. But you may have noticed that there were serious limits to this compression: generally, each data item is compressed to the size of a byte. At absolute minimum, a Bloom filter requires a bit or two of space for every stored item. That isn't going to work when the number of items we want to deal with number in the trillions or even quadrillions.[1]

What we want is a much more aggressive form of compression. In fact, we want it to be so aggressive that we can have a guaranteed size of our data structure that's only *logarithmic* in the size of our data—or even better than that.

How could such a thing be possible? And, if our data structure is that small, won't we be losing almost all of the information we're looking for? Surprisingly, a simple and carefully built data structure can give us extremely valuable information about the data, even under such draconian space requirements.

2 What is a Stream?

A *stream* is a very long sequence of data which is given to the data structure one item at a time. Once an item is given to the data structure, the data structure updates its representation and the item is never seen again. You can imagine this as sitting next to a large stream. You are trying to sample small bits of the stream for analysis. But once something in the stream goes past you, that's it—if you didn't sample it, you won't see it again.

Formalism A stream is a list of items $x_1, \ldots x_N$ (a stream may contain duplicates, where $x_i = x_j$ but $i \neq j$). As in filters, we assume that items are of the same type for the sake of hashing: for some universe U, $x_i \in U$ for all i, and all queries q must also satisfy $q \in U$. Each item in the stream is presented to the data structure one at a time, in order. We will assume that an `insert` function is called on x_1, then x_2, and so on until `insert` is called on x_N.

The data structure is required to have small space, usually $O(\log N)$ or smaller (sometimes $O(1)$). Today we will be looking at an example of each: the Count-Min Sketch requires $O(\log N)$ bits of space for constant error, whereas the HyperLogLog data structure requires $O(\log \log N)$ bits.

After all items have been inserted a single time into the data structure, we can make queries to learn information about the stream. We will focus on two queries today:

- For a query q, how many times did q appear in the stream? In set notation, this asks us to determine $|\{i \in \{1, \ldots, N\} \mid x_i = q\}|$.

[1] The Brazil Internet Exchange alone processes over 7 trillion bits, on average, every second.

- How many distinct items appeared in the stream overall? In set notation, this asks us to determine $|\{x_i \mid i \in \{1, \ldots, N\}\}|$.

Discussion Streaming algorithms have been a major area of research throughout the 2000s and 2010s. There exist entire courses on the topic.[2] Streaming algorithms are are an excellent example of how randomized algorithms can make otherwise-impossible problems tractable.

Why are streaming algorithms so useful? One reason is that data streams are, in fact, fairly common nowadays: packets being transmitted through a node in a network (say, a router on the internet) essentially form a data stream. Any entity interested in monitoring that traffic needs to solve a problem on a large data stream.

However, a further motivation comes (again!) from cache efficiency. As we discussed in class, the most cache-efficient way to go through data is to scan it.[3] So if you have a massive amount of data stored somewhere that you want to analyze, the cheapest way to approach it is to scan through the data once, keeping track of statistics in a small-space data structure as you go. This is, again, a problem on a large data stream.[4]

3 Count Min Sketch

Our first data structure keeps an impressively accurate count of *every* item in a data stream while using extremely small space.

The Count-Min Sketch is one of the best-known streaming data structures. It looks somewhat like a Bloom filter, but with substantially different parameters. We will use two error terms ε and δ in describing the Count Min Sketch, as well as the length of the stream N. We assume every hash h_i outputs a number in $\{0, \ldots, \lceil e/\varepsilon \rceil - 1\}$.

Algorithm 1 Insert x_i

```
for j = 0 to ⌈1/δ⌉ − 1 do
    T[j][h_j(x_i)] = T[j][h_j(x_i)] + 1
end for
```

Algorithm 2 Query q

```
min = 0
for j = 0 to ⌈1/δ⌉ − 1 do
    if min < T[j][h_j(q)] then
        min ← T[j][h_j(q)]
    end if
end for
return min
```

Structure The Count Min Sketch (CMS) consists of a two-dimensional table T with $\lceil \ln 1/\delta \rceil$ rows. Each row consists of $\lceil e/\varepsilon \rceil$ entries. Each entry must be of size at least $\log N$ bits.[5]

A Count Min Sketch has $\lceil \ln 1/\delta \rceil$ hash functions, one for each row of the table. In this way, the CMS essentially consists of $\lceil \ln 1/\delta \rceil$ independent hash tables stored in one location.

To initialize the structure, we set all entries in T to 0.

Insert To insert an item x_i, we iterate through each row of the CMS. For each row, we hash x_i, and increment the counter stored at that hash location.

Query To query q, we iterate through each row of the CMS. For each row, we hash q, and keep track of the value stored at that location. We return the minimum such value found. This value is an estimate of how many times q occurred during the stream: that is to say, it estimates $|\{i \mid x_i = q\}|$.

[2] For example, this 2007 course at MIT: `http://stellar.mit.edu/S/course/6/fa07/6.895/materials.html`.

[3] Scans also tend to avoid branch mispredictions and other practical inefficiencies.

[4] You may notice that if our goal cache-efficiency, there's an opportunity for a tradeoff: I can scan the data multiple times, resulting in an increase in I/O cost, but with a second chance to see important data items. Can seeing a stream several times lead to more efficient methods? In short, the answer is frequently yes. This is called the "multi pass" streaming model. While this model is fairly popular we will not be looking at it in this class.

[5] In practice, one would just choose 16, 32, or 64 bit entries depending on the size of the stream.

3.1 Bounds

A Count Min Sketch gives the following guarantee. Let $q \in U$ be any query, let o_q be the value given by the query algorithm on q, and let $\widehat{o_q}$ be the true number of occurences of q in the stream (i.e. $\widehat{o_q} = |\{i \mid x_i = q\}|$). Then we have the following two guarantees:

1. $\widehat{o_q} \leq o_q$, and
2. with probability at least $1 - \delta$, $o_q \leq \widehat{o_q} + \varepsilon N$.

Multiplying the number of rows, number of columns, and size of each entry, we can see that the Count Min Sketch requires $\lceil e/\varepsilon \rceil \lceil \ln 1/\delta \rceil \lceil \log N \rceil$ bits.

4 HyperLogLog

The HyperLogLog data structure is much smaller than Count Min Sketch. It does not attempt to retain the number of occurrences for each individual count. Rather, it attempts to estimate the number of *distinct* items in the stream. That is to say, it estimates $|\{x_i \mid i \in \{1, \ldots N\}\}|$.

Structure The HyperLogLog data structure consists of an array M of counters, where M is of length m (we assume m is a power of 2). Each counter should have at least $\log \log n$ bits; in practice 8-bit counters are sufficient for *any* application.[6] The HyperLogLog data structure also requires a (single) hash function, which we will call h.

Algorithm 3 Insert x_i

$j \leftarrow h(x_i) \log_2 m$
$r \leftarrow h(x_i) \texttt{>>} \log_2 m$
$z \leftarrow$ # trailing 0s of r
if $M[j] < z + 1$ **then**
  $M[j] \leftarrow z + 1$
end if

Algorithm 4 Query

$Z = 0$
for $j = 0$ to $m - 1$ **do**
  $Z \leftarrow Z + (1/2)^{M[j]}$
end for
return bm^2/Z

Insert Inserting x_i begins by hashing x_i. Processing this hash proceeds in two steps. First, we obtain an index into the table. Then, we perform a calculation on the hash.

The first step is to obtain an index j using the rightmost $\log_2 m$ bits of $h(x_i)$.[7] These bits are then shifted off to obtain a remainder $r = h(x_i)$ >> $\log_2 m$.

Then, we count the number of trailing zeroes in the binary representation of r and store that number in an integer z. (Mathematically, we find the largest z such that $r/2^z$ is an integer.)

We look up $M[j]$. If $M[j] > z+1$, we set $M[j] = z + 1$. In other words, at every point in time $M[j]$ stores the largest number of zeroes (plus one!) of the remainder of any item that hashes to j.

Query The query attempts to estimate how many distinct items were seen in the stream based on M. At a high level, the following process constitutes taking a biased harmonic mean[8] of the entries of M.

[6] In particular, 8-bit counters are sufficient even if your stream consists of all particles in the observable universe—much larger than any stream you would see on Earth.

[7] Because m is a power of 2, $\log_2 m$ is always an integer.

[8] That is to say, a special kind of average that happens to work particularly well in this setting.

m	b
16	.673
32	.697
64	.709
≥ 128	$.7213/(1+1.079/m)$

Table 1

To begin, we set a double $Z = 0$. Then for each j from 0 to $m-1$, we add $(1/2)^{M[j]}$ to Z.

We calculate a bias constant b, which depends on m. Good values of b can be found in Table 1.

Finally, we return bm^2/Z.

Parameters and caveats Using 8 hash bits per element is sufficient for a stream of any size. In fact, as streams get large, the main parameter that needs to change is the size of the hash function output. The hash function output should be large enough that any two elements are unlikely to collide (after all, if x and y have $h(x) = h(y)$, the HyperLogLog will treat them the same and will not count them as distinct elements). 32 bits was suggested in the original paper, and is fine for moderate-sized streams (in fact, for the assignment, I'd suggest you just use one 32-bit output from Murmurhash). However, streams with billions (or even hundreds of millions) of distinct elements—specifically, streams with close to 2^{32} elements or more—will incur a decrease in quality with 32 bit hashes, and a 64 bit hash should be used instead.

The value of m is relatively small in the assignment, but is often moderately large in practice. Counters are cheap, so one often sees $m = 1024$ or $m = 2048$ to give improved estimates.

For streams where the number of distinct elements is very small (less than $5m/2$) or very large (close to 2^{32} with 32-bit hashes), one can still use the same data structure, but should calculate the approximate set size using an alternate method. We won't be dealing with these methods in class, but they are important for implementing this structure in practice.

Guarantees? You are not expected to know the guarantees of this algorithm in this class, but I include them here for completeness. Let D be the number of distinct elements in the stream. The *expected* output of HyperLogLog is D itself—great, but how often is it going to be close to that? To get an idea of how frequently this happens, the standard error[9] of D is $1.04/\sqrt{m}$. It appears that the outputs given by HyperLogLog are close to a Gaussian distribution. Therefore, the estimate should be within a $1.04/\sqrt{m}$ factor of D 65% of the time, and should be within a $2.08/\sqrt{m}$ factor of D 95% of the time.

[9]This is essentially the standard deviation multiplied by the correct estimate D.