CS 167 Paper Report

"An improved data stream summary" by Cormode/Muthukrishnan

Leo Martel (lmartel)

May 14, 2014

In their 2005 paper, Cormode and Muthukrishnan propose the Count-Min Sketch, a new data structure designed to efficiently summarize streaming data.

1 Streaming Data Algorithms

1.1 Background

The Count-Min Sketch is a streaming data algorithm, which is a class of algorithm with some special properties that lend themselves well to summarizing large and dynamic data sets. A streaming data algorithm:

(A) Uses sublinear space, at most $O(\lg^k n)$ (polylogarithmic);

(B) Approximates queries rather than giving exact answers;

and

(C) Needs only a constant number of passes–for our purposes, only a single pass–over the data set.

It has been proved that (A) implies (B), which makes intuitive sense; we don't have enough space to even record the entire input, so expecting exact answers is unrealistic. However, these approximations can still be useful, because the error bounds of most streaming data algorithms can be tuned by the user, though often at the cost of space and runtime. The estimation error allowed by streaming algorithms is typically presented in terms of two parameters, ϵ and δ, such that the error in answering a query is within a factor of ϵ with probability $1 - \delta$.

Typical applications of streaming data algorithms involve creating approximate "sketches" of large, rapidly changing datasets, taking advantage of (A) and (C). Motivating examples include:

- Analyzing truly massive data like Google's web graph, where $O(n)$ space is too expensive.

- Running analytics on a production database, where new entries stream in constantly. We need to be able to update our sketch with new information without having to start from scratch.
- Monitoring streaming data that cannot be revisited later, like Internet traffic.
- Extracting the important information from a high-bandwidth stream of individually low-value data points. For example, a security camera spends almost all of its time staring at nothing, but we still need to be able to detect and react to changes or threats.

A simple example of a streaming algorithm is the basic min-finding algorithm of comparing each element to the previous minimum, and updating the minimum upon finding a smaller element. This simple approach requires only a single pass, and finds the minimum element using only constant space regardless of the size of the data set. Every streaming algorithm makes sacrifices, however; in this case, we have sacrificed flexibility. If we later decide we want the maximum element, we have no way of finding it.

1.2 Bloom Filters

The Count-Min Sketch proposed by Cormode and Muthukrishnan is in many ways a generalization of another better-known stream sketching algorithm called the bloom filter. The bloom filter is used for approximating containment queries in a set. It is represented as a bit vector and k independent hash functions. We insert an element e by hashing it with each of our k hash functions and setting the bits found to 1, and check containment by checking whether all of an element's corresponding bits have been toggled.

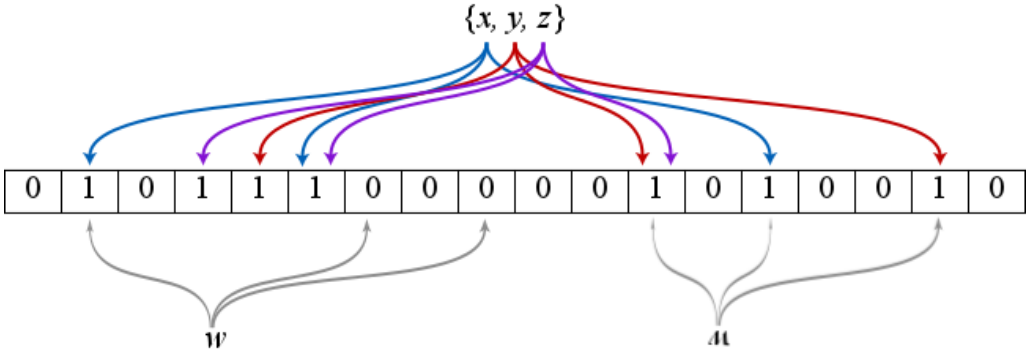

Figure 1: A series of bloom filter insertions (x, y, z) and containment queries (w, m)

Major advantages of the bloom filter include the **compact bit vector representation** and the **fast insertions** whose runtimes do not increease as we insert more objects into the structure.

As suggested by Figure 1, false positives are possible–$contains?(m)$ will evaluate to true even though we've only inserted x, y, z. False positives are caused by hash collisions; since all of the k bits associated

with m have already been flipped by hash collisions, we can no longer tell that m hasn't been inserted. On the other hand, false negatives are impossible; since hash functions are deterministic, we always check the k correct bits when doing a lookup. We can be completely confident that the $FALSE$ result of $contains?(w)$ in Figure 1 is accurate.

This "one-sided" error allows us to tune the parameters of the bloom filter to minimize false positives without worrying about incurring false negatives, making the bloom filter a practical, accurate sketching algorithm. This key property of bloom filters will return in our analysis of the Count-Min Sketch.

2 The Count-Min Sketch Algorithm

2.1 Setup

The goal of the Count-Min Sketch is to summarize streaming updates to a vector of numbers called a. We want to support the following queries:

(A) a_i (the value of the i-th component of a), called the "point query" in the paper;

(B) $a \cdot b$ (the dot product of two sketched vectors a and b), the "inner product query;"

(C) $a.sum(i..j)$ (the sum of elements i through j), the "range query."

One of the most interesting aspects of the Count-Min Sketch is that it enables several different queries; past sketching schemes often required maintaining a separate sketch for each query, using up much of the space savings gained from using sketching in the first place.

We represent a Count-Min Sketch as a 2-dimensional array with width $w = ceil[e/\epsilon]$ and depth $d = ceil[\lg(1/\delta)]$, increasing the size of the sketch as we tighten the error bounds.

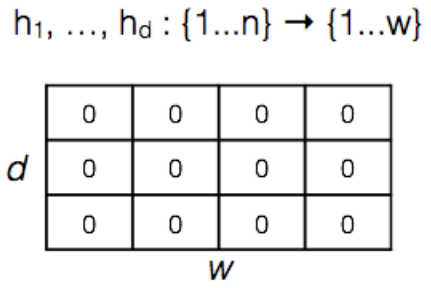

Figure 2: A visualization of a Count-Min Sketch, where n is the dimension of the data vector a

In addition, we choose d hash functions uniformly at random from a pairwise-independent family. (As an aside, universal hashing suffices for the Count-Min Sketch–we will show this is true for the Point Query

operation later–but Cormode and Muthukrishnan's paper lists pairwise hashing as a requirement.)

2.2 Pairwise-Independent Hashing: Definition

A set H of hash functions from $\mathbb{U}$ to $S = 1, 2, ..., n$ is **pairwise-independent** if for any $x \neq y$ with $x, y \in \mathbb{U}$ and any $v_1, v_2 \in S$ we have:

$$Pr_{h \in H}[h(x) = v_1 \; and \; h(y) = v_2] = \frac{1}{n^2}$$

Informally, a pairwise-independent hash function's outputs seem random for any two distinct inputs (but offers no guarantees for sets of three or more inputs).

Pairwise-independent hashing is a stronger assumption than universal hashing, but it is easy to see that it implies universal hashing, because for a pairwise-independent family H:

$$Pr_{h \in H}[h(x) = h(y)] = \sum_{v=1}^{n} Pr_{h \in H}[h(x) = v \; and \; h(y) = v] = n \cdot \frac{1}{n^2} = \frac{1}{n} \leq \frac{1}{n}$$

Once we have chosen our hash functions and allocated our w by d matrix, we're ready to load the data.

2.3 Updating the Sketch

As this is a streaming data structure, updates to a appear one at a time with no clearly delimited stopping point. Updates are represented by the authors in the form

$$update(i_t, c_t)$$

which means "add c to the i-th component of a at time t." We process $update(i_t, c_t)$ as follows:

$$\forall 1 \leq j \leq d, \; set \; CMS[j][h_j(i_t)] += c_t$$

That is, we use the hash function associated with each row to pick a cell based on i_t, then add c_t to that cell.

As you can see in Figure 3, the fact that we hash i_t means that updates to the same element of a will be added to the same d cells (one per row) every time. However, due to the small size of the sketch matrix, there will inevitably be some hash collisions which lead to overlap, as in the purple-outlined cell in Figure 3.

These two facts suggest a one-sided error bound for each row; we can get hash collisions but we'll never "miss" when looking for an element. We'll use this intuition to develop queries for the Count-Min Sketch.

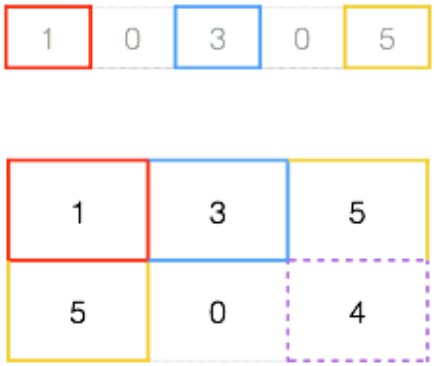

Figure 3: A $2x3$ Count-Min Sketch of a 5-dimensional vector, with insertions color-coded

2.4 Queries

(1.) The Point Query (current value of a_i):

$$p_{est} = min_{1 \leq k \leq d} CMS[j][h_h(i)]$$

We simply hash the index we're interested in, and return the minimum value of its d associated cells (one per row of the sketch).

The error bounds of this query are

$$p_{true} \leq p_{est} \leq p_{true} + \epsilon ||a||_1$$

(This makes sense from our intuition above, but we'll prove these bounds in the last sections of the paper.)

(2.) The Inner Product Query (value of $a \cdot b$ for two sketched vectors):

$$ip_{est} = min_{1 \leq k \leq d} (CMS_a[j] \cdot CMS_b[j])$$

We take the minimum dot product of matching rows in the two sketch matrices (rows 1 and 1, 2 and 2 etc). This feels correct based on how the point query works, but let's examine it a little more closely.

The key observation (and requirement) for the Inner Product Query is that the two CMS structures use **the same hash functions for corresponding rows**. This means that corresponding values in a and b still correspond in CMS_a and CMS_b but are "scrunched together" by collisions. Elements

that collide contribute to the dot product of the sketch rows despite not contributing to the true dot product. Taking the *min* over the row dot products gets us the estimate with the fewest collisions.

The bounds of this query look similar to those of the Point Query:

$$ip_{true} \leq ip_{est} \leq ip_{true} + \epsilon||a||_1||b||_1$$

(3.) The Range Query (current value of $a_i + a_{i+1} + ... + a_{j-1} + a_j$):

The naive approach would be to make a Point Query at each of $a_i + a_{i+1} + ... + a_{j-1} + a_j$. This would get us an error bound of

$$sum_{true} \leq sum_{est} \leq sum_{true} + n\epsilon||a||_1$$

since each point query can contribute up to an epsilon of error, and in the worst case we're querying for a sum over the entire vector a.

However, this query is answerable with a much tighter error bound, but doing so requires some extra setup.

2.5 Dyadic Ranges

A Dyadic Range is a class of interval with some useful mathematical properties. An interval is a dyadic range if it has a power-of-two length $l = 2^k$ and its start index is 1 (mod l) (assuming 1-indexing). That is, each point in $1..n$ is a member of exactly one diadic range of each length $2^0, 2^1, ...2^{\lg n}$. It turns out that any range within $0..n$ has a unique, non-overlapping, complete covering consisting of at most $2\lg(n)$ dyadic ranges.

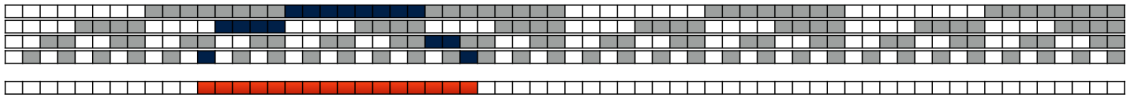

Figure 4: The range $[12, 27]$ for $n = 64$ is covered by dyadic ranges $[12, 12], [13, 16], [17, 24], [25, 26], [27, 27]$.

We can maintain $\lg(n)$ Count-Min Sketches for dyadic range lengths $2^0, 2^1, ...2^{\lg n}$. This means the first CMS will sketch individual elements, the second CMS sketches pairs of elements (summing updates to either element), and so on. We can then use at most $2\lg n$ Point Queries over these sketches to assemble a Range Query. Since we have reduced the number of point queries, our error bound tightens; the error

bounds for the dyadic range-based Range Query are:

$$sum_{true} \leq sum_{est} \leq sum_{true} + 2(\lg n)\epsilon||a||_1$$

Which is much better. Note, though, that we're now maintaining $\lg(n)$ Count-Min Sketches, so we've somewhat increased our space usage. However, since the sketches are simulating vectors of size $n, n/2, ..., 1$ this is only a constant-factor increase of 2 (or a bit more including overhead), which shouldn't be a problem.

2.6 Selected Applications

- Quantiles

  The problem of finding quantiles reduces to CMS range sums. For example, if we're tracking incomes from an ongoing survey and want to see the number of people in the top 10%, we simply run the range query $a.sum(\frac{9}{10}n, ..., n)$.

- Heavy Hitters (most frequent elements), Cash Register Model

  Keeping track of the most frequent elements is easy in the "cash register" model (purely positive updates) by taking advantage of the fact that updates stream in one at a time.

  First, we determine our threshold for "frequent:" in the language of Cormode/Muthukrishnan, the ϕ-heavy hitters of a are $\{a_i \mid a_i \geq \phi||a||_1\}$. For example, for $\phi = \frac{1}{2}$ a heavy-hitter is an element contributing over half the total value of the vector. Notice that there can be as few as 0 and no more than $\frac{1}{\phi}$ heavy hitters.

  We keep a small min-heap (max size $\frac{1}{\phi}$) of likely Heavy Hitters, and maintain an exact value of $||a||_1$ by summing every incoming update.

  After each update, we run a CMS Point Query on the updated element, and if the value meets the threshold of $\phi||a||_1$ we push it into the heap, popping any elements whose value has fallen below the updated threshold.

  The paper proves that this algorithm finds every ϕ-heavy hitter, and with probability $1 - \delta$ outputs no elements with value below $(\phi - \epsilon)||a||_1$. Informally, this means we get no false negatives, and our false positives will likely be elements with high values that are "close" to being Heavy Hitters.

- Heavy Hitters, Turnstile Model

The paper also specifies a related but more complicated algorithm for heavy hitters in the "turnstile" model (negative updates, positive totals) which finds every $(\phi + \epsilon)$-heavy hitter and with probability $1 - \delta$ outputs no elements with value below $\phi||a||_1$. The details of this algorithm are not a central result of the paper, however, so I've omitted them for space.

3 Selected Query Runtime Proofs

3.1 The Point Query (Turnstile Model)

Authors' Theorem: the estimate ai has the following guarantees: $a_i \leq \hat{a}_i$; and, with probability at least $1 - \delta$,

$$\hat{a}_i \leq a_i + \epsilon||a||_1$$

Key points:

- In each row of the Count-Min Sketch, $Pr[hash\ collision] \leq \frac{\epsilon}{e}$

  Proof: let $I_{i,j,k}$ be an indicator variable for $i \neq j$ colliding under hash function h_k.the authors use pairwise-independence to justify $E[I_{i,j,k}] = Pr[h_j(i) = h_j(k)] \leq \frac{1}{range(h_j)}$, and of course $range(h_j) = w = \frac{e}{\epsilon}$. Note that universal hashing is sufficient for this step, as this inequality follows directly from the definition.

- Next, they define $X_{i,j} = \sum_{k=1}^{n} I_{i,j,k} a_k = \sum_{\{k|h_j(i)=h_j(k)\}} a_k$, the sum of all components of a that collide with the i-th component when hashed by h_j. Notice that this means $PointQuery[a, i] = min_j(a_i + X_{i,j})$.

  Proving the lower bound is easy; since the turnstile model guarantees us positive $a[i]$ for all i, so $PointQuery[a, i] = min_j(a_i + X_{i,j}) \geq a_i$. ■

- Finding the upper bound takes more effort. By the linearity of expectation we can see that

$$E[X_{i,j}] = E[\sum_{k=1}^{n} I_{i,j,k} a_k] = \sum_{k=1}^{n} a_k E[I_{i,j,k}]$$

  Next, we just plug in our value for $E[I_{i,j,k}]$:

$$\sum_{k=1}^{n} a_k E[I_{i,j,k}] \leq \sum_{k=1}^{n} \frac{\epsilon}{e} a_k = \sum_{k=1}^{n} a_k$$

Finally, since $\sum_i a_i \leq ||a||_1$ (with equality for the special case of a_i all positive), we have

$$E[X_{i,j}] = \frac{\epsilon}{e}||a||_1$$

- This expectation of the error of any single row seems promising! We can use this to calculate the probability of the overall query missing our promised $\epsilon||a||_1$ error bound.

  The probability that this happens is equivalent to the probability that the error in every row exceeds the error bound (since we're taking a min):

$$Pr[\hat{a}_i > a_i + \epsilon||a||_1] = Pr[\forall_j . a_i + X_{i,j} > a_i + \epsilon||a||_1]$$

  Subtracting a_i from both sides of the inequality gets us to

$$= Pr[\forall_j . X_{i,j} > \epsilon||a||_1]$$

  Note that $\epsilon||a||_1 = e \cdot E[X_{i,j}]$, so we have

$$= Pr[\forall_j . X_{i,j} > eE[X_{i,j}]]$$

  By the Markov inequality, usable since $X_{i,j}$ are always positive, we have

$$Pr[X_{i,j} > eE[X_{i,j}]] \leq \frac{E[X_{i,j}]}{eE[X_{i,j}]} = \frac{1}{e}$$

  And since our hash functions were chosen independently-at-random, these probabilities are independent. Therefore

$$Pr[\forall_j . X_{i,j} > eE[X_{i,j}]] \leq Pr[X_{i,j} > eE[X_{i,j}]]^d = \frac{1}{e^d} = \delta$$

  And we have proved that our upper error bound is $\epsilon||a||_1$ with probability at least $1 - \delta$. ■

3.2 The Point Query (General Model)

Most of the paper operates under the assumption of strictly positive values of a (though individual negative updates are fine). In addition to making the math easier and the bounds tighter, the one-sided error bound–

as you saw above–relies on this assumption. Just like in bloom filters, having a one-sided error bound is very powerful in many applications, and allows considerably more precision in tuning our parameters.

However, sometimes we can't avoid negative elements; in these cases, the Count-Min sketch is still useful for its space-efficiency with a fairly tight two-sided error bound. We make a slight modification: we now take the median of the row results instead of the minimum. This makes intuitive sense, but let's prove it.

The proof for the error bounds of the Point Query in the general case is a surprisingly (to me, at least) painless extension of the turnstile proof.

Given our intermediate result from above, $|E[X_{i,j}]| \leq \frac{\epsilon}{e}||a||_1$, and loosening it slightly gives us

$$8|E[X_{i,j}]| < 3e|E[X_{i,j}]| \leq 3\epsilon||a||_1$$

so the chance of any row's error exceeding $3\epsilon||a||_1$ is less than $\frac{1}{8}$. Let P^* denote the probability that more than half of the rows are within this bound. This probability can be calculated using a Chernoff bound with $n = d = \lg(1/\delta)$ and $p = \frac{7}{8}$:

$$P^* \geq 1 - e^{-\frac{1}{2p}n(p-0.5)^2} = 1 - e^{-\frac{4}{7}n\left(\frac{3}{8}\right)^2} = 1 - e^{-\frac{9}{112}\lg(1/\delta)} = 1 - \left(\frac{1}{\delta}\right)^{-\frac{9}{112}} = 1 - \delta^{\frac{9}{112}}$$

So by choosing the median row, we stay within our triple-epsilon error bound with very high probability ($1 - \delta^{\frac{9}{112}} >> 1 - \delta^{\frac{1}{4}}$). ■

4 Wrap-up

The Count-Min Sketch provides several major advantages over past work in the area of streaming algorithms. The CMS:

- Handles several queries, providing greater flexibility and broader use cases
- Scales space usage with $(1/\epsilon)$ rather than $(1/\epsilon)^2$
- Uses only pairwise-independent (or even just universal) hashing rather than p-wise independent
- Provides error bounds with small constants without using asymptotic notation
- Uses fast updates, touching only $\frac{1}{w}$ of the sketch on each update

One potential drawback is the use of the L_1 norm instead of the (smaller) L_2 norm in the error bounds, although this can often be offset by the small constants. For a large number of applications the Count-Min Sketch is a practical and efficient algorithm for sketching large streaming data sets.