Approximating Data with the Count-Min Sketch

Graham Cormode, AT&T Labs Research

S. Muthukrishnan, Rutgers University

// The Count-Min sketch concurrently tracks several item counts with surprisingly strong accuracy for a variety of applications. //

ALGORITHMIC PROBLEMS SUCH as tracking a set's contents arise frequently when building systems. Given the variety of possible solutions, the choice of appropriate data structures for such tasks is at the heart of building efficient, effective software. Large libraries of algorithms and data structures augment modern languages to help programmers without forcing them to reinvent the wheel. These familiar data structures and methods address a host of problems using heaps, hash tables, tree structures, stacks, and so on.

Consider the following membership problem, which is fundamental in computer science. Starting from an empty set S, a series of update operations are drawn from insert(S, i) (that is, the operation performs $S \leftarrow S \cup \{i\}$); or delete($S$, i) (this operation performs $S \leftarrow S - \{i\}$). Interleaved therein are queries check(S, i) (which return yes if $i \in S$, and no otherwise). The task is to maintain S under the update operations and respond to queries using a small amount of space, making each operation as fast as possible. Extensive research has covered solving this problem using data structures such as hash tables, balanced trees, and B-tree indices. These form the backbone of many systems, including OSs, compilers, and databases. Many of these data structures have seen widespread use for 40 years or more.

Many data structures that are suitable for main memory or disk-resident data no longer suffice for modern applications that must handle massive data. Such applications present new problems and require a new set of data structures that should be relatively small, ideally sublinear in the size of the input. Also, in many cases, approximate responses suffice for various queries, and applications will work with imprecise answers.

A fundamental problem at the heart of such applications is count tracking, which generalizes the membership problem. This problem has led to data structures called *sketches*, which approximate massive datasets. In particular, we present the Count-Min sketch,[1] which concurrently tracks several items with surprisingly strong accuracy. It has wide applications from IP networking to machine learning, distributed computing, and even signal processing and beyond.

Count Tracking and Sketches

Count tracking involves many items, each with an associated frequency that changes over time. A query specifies an item; the response to it is the item's *frequency*.

Consider a popular website that wants to keep track of statistics on the queries used to search the site. It could keep track of the full log of queries and know any search query's exact frequency. However, the log would quickly become huge. This is an instance of the count-tracking problem. Even known, sophisticated solutions for fast querying (such as a tree structure or hash table to count multiple occurrences of the same query) can be slow and waste resources.

In this scenario, we can tolerate a

0740-7459/12/$31.00 © 2012 IEEE

little imprecision. Generally, we're interested in only the frequently asked queries, so it's okay if the counts aren't exact. Thus, we can trade some precision for a more efficient, lightweight solution. This tradeoff is at the heart of sketches.

Other real-world applications of count tracking abound. In an online-retailer scenario, items might be goods for sale, and the associated frequency would be the number of purchases of each item. In a stock-trading setting, the items might be stocks, and the associated frequency would be the total number of shares traded on a given day. We can further apply count tracking to derived data, such as the difference of data between distributed sites or different time periods. Even more generally, we can use count tracking for many tasks on massive data distributions, such as detecting anomalies, summarizing, mining, or classifying.

Formally, any data structure for count tracking must provide two operations:

- **update**(i, c) updates the frequency of item i by c.
- **estimate**(i) estimates the current frequency of i.

We can solve this problem exactly using traditional data structures; for example, a hash table can keep the set of items and associated frequencies.[2] We can implement both the update and estimate operations directly through standard hash table methods. However, such solutions have disadvantages: the amount of memory the data structure uses can become very large as more items are added. Owing to such a data structure's large size, accessing it can be slow because it resides in slow or virtual memory. Furthermore, because the data structure grows over time, it must be periodically resized, which can affect real-time processing. Finally, in distributed applications, if we must communicate the entire frequency distribution, the overhead can be prohibitive.

Sketches overcome these problems by replacing the exact answer with a high-quality approximation. For instance, in presenting statistics on customer buying patterns, an uncertainty of 0.1 percent (or less) isn't significant. Such sketch data structures can accurately summarize arbitrary data distributions with a compact, fixed-memory footprint that's often small enough to fit within cache, ensuring fast processing of updates and quick communication between sites.

The Count-Min Sketch

The Count-Min sketch provides a different solution to count tracking. It allocates a fixed amount of space to store count information. This space doesn't vary over time, even as more and more counts are updated. Nevertheless, the Count-Min sketch can provide useful estimated counts because the accuracy scales with the sum of the stored counts.

If N represents the current sum of all the counts (that is, the sum of all the c values in update operations), the Count-Min sketch promises distortion that's a small fraction of N. This fraction is controlled by a parameter of the data structure: the smaller the possible uncertainty, the larger the sketch.

Sketch Internals

With all data structures, it's important to understand the data organization and algorithms for updating the structure to make clear the relative merits of different choices of structure for a given task. The Count-Min sketch consists primarily of a fixed array of counters of width w and depth d. The counters are all initially zero. Each row of counters is associated with a different hash function that maps items uniformly onto the range $\{1, 2, \ldots, w\}$. The hash functions don't need to be particularly strong (they aren't as complex as cryptographic hash functions). For items represented as integers i, the hash functions can be of the form ($a * i + b \bmod p \bmod w$), where p is a prime number larger than the maximum i value (say, $p = 2^{31} - 1$ or $p = 2^{61} - 1$), and a and b are values chosen randomly from the range of 1 to $p - 1$. Each hash function must be different; otherwise, the repetition provides no benefit.

Update(i, c) updates the data structure in a straightforward way. In each row, it applies the corresponding hash function to i to determine a corresponding counter. Then it adds the update c to that counter.

Figure 1 shows an update for $w = 9$ and $d = 4$. The first hash function maps the i update to an entry in the first row, then adds the c update to the current counter. Similarly, it maps the item to different locations in each of the other three rows. So, in this example, we evaluate four hash functions on i and update four counters accordingly.

For **estimate**(i), the process is similar. In each row, it applies the corresponding hash function to i to look up one of the counters. The estimate is the minimum of all the probed counters across all rows. For the example in the previous paragraph, we examine each place where the hash functions mapped i. In

Sketch data structures can accurately summarize arbitrary data distributions with a compact, fixed-memory footprint that's often small enough to fit within cache.

FURTHER READING ON SKETCHES

Here are a few sources for more information on the Count-Min sketch and other sketches.

THE COUNT-MIN SKETCH

For information on the Count-Min data structure, visit https://sites.google.com/site/countminsketch. More technical coverage appears in our original paper describing the structure,[1] textbooks on randomized algorithms,[2] and a survey of techniques for the count-tracking problem and its variations.[3]

Implementations of the Count-Min sketch are available for different languages. The MassDal code bank contains a C implementation (www.cs.rutgers.edu/~muthu/massdal-code-index.html). Marios Hadjieleftheriou offers one in C++ (www2.research.att.com/~marioh/frequent-items/index.html). Edward Yang provides one in OCaml (https://github.com/ezyang/ocaml-cminsketch). Berkeley and Greenplum researchers have developed a SQL implementation for distributed, high-performance settings (http://doc.madlib.net/v0.2beta/sketch_8sql__in_source.html).

OTHER SKETCHES

The Bloom filter solves the slightly simpler "membership" problem discussed at the start of the article. Bloom filters have a size linear with the number of items but use less memory than the standard hashing approach. In networking applications, Bloom filters are popular for tracking which flows have been seen and which objects are stored in caches. For more information, see Andrei Broder and Michael Mitzenmacher's survey.[4]

An AMS (Alon, Matias, and Szegedy) sketch can represent high-dimensional vectors in a small space. Such vectors often occur in machine-learning applications, in which each entry of the vector encodes the presence or absence of some feature. The sketch allows accurate estimation of the inner product (the cosine distance) of two vectors.[5]

Various sketches track other functions of item sets. The most basic problem they solve is to estimate the number of different elements in the set. For example, they can track the number of unique visitors to a website. Unlike the Bloom filter, they don't accurately track exactly which items are in the set. By removing this requirement, they can be much smaller than the number of items in the set.[6]

References

1. G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *J. Algorithms*, vol. 55, no. 1, 2005, pp. 58–75.
2. M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge Univ. Press, 2005.
3. G. Cormode and M. Hadjieleftheriou, "Finding the Frequent Items in Streams of Data," *Comm. ACM*, vol. 52, no. 10, 2009, pp. 97–105.
4. A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, 2005, pp. 485–509.
5. N. Alon et al., "Tracking Join and Self-Join Sizes in Limited Storage," *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems* (PODS 99), ACM Press, 1999, pp. 10–20.
6. K. Beyer et al., "Distinct-Value Synopses for Multiset Operations," *Comm. ACM*, vol. 52, no. 10, 2009, pp. 87–95.

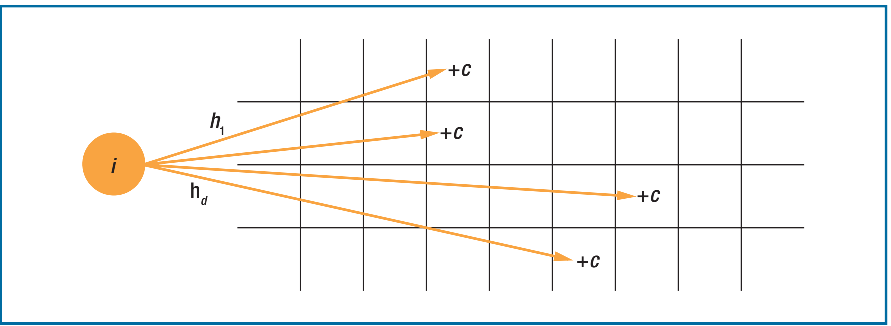

FIGURE 1. A schematic of the Count-Min sketch. This schematic shows an update to a sketch with $w = 9$ and $d = 4$, where w is width and d is depth.

Figure 1, this is the fourth entry in the first row, the fourth entry in the second row, the seventh entry in the third row, and the sixth entry in the last row. Each of these entries' counters added up all the updates that were mapped there. We picked the smallest of these as the estimate.

Why It Works

At first, it might be unclear why this process should give any usable estimate of counts. It seems that because each counter is counting the updates for many different items, the estimates will inevitably be inaccurate. However, the hash functions spread out the different items, so on average, the inaccuracy can't be too high. With different hash functions, the chance of getting an estimate that's much more inaccurate than average is small. (The "Sketch Accuracy" sidebar explains this further.) Thus, for a sketch of size $w \times d$ with a total count N, any estimate has an error of at most $2N/w$, with a probability of at least $1 - (1/2)^d$.

```
Algorithm 1: CountMinInit(w, d, p)
1  C[1, 1] … C[d, w] ← 0;
2  for j ← 1 to d do
3      Pick a_j, b_j uniformly from [1 … p − 1];
4  N = 0

Algorithm 2: CountMinUpdate(i, c)
1  N ← N + c
2  for j ← 1 to d do
3      h_j(i) = (a_j × i + b_j mod p) mod w;
4      C[j, h_j(i)] ← C[j, h_j(i)] + c;

Algorithm 3: CountMinEstimate(i)
1  e ← ∞;
2  for j ← 1 to d do
3      h_j(i) = (a_j × i + b_j mod p) mod w;
4      e ← min(e, C[j, h_j(i)]);
5  return e
```

FIGURE 2. Pseudocode for Count-Min sketch. **CountMinInit** initializes the structure. **CountMinUpdate** applies an update of c to item i. **CountMinEstimate** provides the estimated count of item i.

SKETCH ACCURACY

Statistical analysis can verify the quality of the estimates given by the Count-Min sketch. In a given row, the counter probed by the **estimate** operation includes the current frequency of item i. However, because w (width) is typically smaller than the total number of summarized items, hash collisions will occur: the count found will be the sum of the frequencies of all items mapped by the hash function to that location. In traditional hash tables, such collisions are problematic; in this case, they're tolerable because they still provide an accurate estimate.

Because the hash function spreads items uniformly throughout the row, we expect that a uniform fraction of items will collide with i. This translates into a uniform fraction of the sum of frequencies: if the sum of all counts is N, the expected fraction of weight colliding with i is at most N/w. If we're lucky, the colliding weight will be less than this, but it could be more. However, it will unlikely be much larger than the expected amount. The probability of seeing more than twice the expected amount is at most 1/2 (this follows from the Markov inequality in statistics). So, the value of this counter is at most $2N/w$ more than the true frequency of i with a probability of at least 1/2.

The same process repeats in each row with different hash functions. Because the hash functions are different each time, they give a different mapping of items to counters. So, a different collection of items collide with i in each row. Each time, there is (at most) a 50 percent chance of getting an error of more than $2N/w$, and (at least) a 50 percent chance of having an error of less than this. Because we take the minimum of counters for x over all rows, the final result will have an error of more than $2N/w$ only if all d rows give a "large" error, which happens with a probability of at most $(1/2)^d$.

From this analysis, we know how to set the parameters for the sketch's size. Suppose we want an error of at most 0.1 percent (of the sum of all frequencies) with 99.9 percent certainty. Then, we want $2/w = 1/1{,}000$. We set $w = 2{,}000$ and $(1/2)^d = 0.001$; that is, $d = \log 0.001/\log 0.5 \leq 10$. Using 32-bit counters, the array of counters requires $w \times d \times 4 = 80$ Kbytes.

So, setting w and d large enough lets us achieve high accuracy while using relatively little space.

Implementations of the Sketch

We now discuss using Count-Min sketches in a traditional single-threaded implementation, in a parallel and distributed implementation, and in hardware implementations.

A Single-Threaded Implementation

Implementing the sketch in a traditional single-CPU environment is straightforward. Indeed, several libraries are available for the data structure in common languages such as C, C++, Java, and Python.

Figure 2 shows skeletal pseudocode for initializing and updating the sketch. The code in Algorithm 1 initializes the array C of $w \times d$ counters to 0 and picks values for the hash functions on the basis of the prime p. For each **update**(i, c) in Algorithm 2, the total count N is updated with c, and the loop in lines 2 to 4 hashes i to its counter in each row and updates the counter there. The procedure for **estimate**(i) in Algorithm 3 is almost identical to this loop. Given i, we perform the hashing in line 3 and keep track of the smallest value of $C[j, h_j(i)]$ over the d values of j.

Several tricks can make the code as fast as possible. In particular, the hash function in line 3 of Algorithms 2 and 3 has two modulus operations, which appear to be slow. We can remove both: we replace mod p with a shift and an add for certain choices of p (Mikkel Thorup wrote a concise guide[3]) and replace mod w with a bitmask operation when w is chosen to be a power of 2. Under these settings, we can easily update the sketch at a rate of millions of operations per second, approaching the I/O limit.

A Parallel and Distributed Implementation

The operations that manipulate the sketch are largely oblivious to the data structure's current state—that is, sketch updates don't require inspecting the current state. This means that the data structure is highly suitable for parallelization and distributed computation.

First, we update each row of the sketch independently so that we can partition the sketch by row among threads on a single machine. But more than this, we can build sketches of

ABOUT THE AUTHORS

GRAHAM CORMODE is a principal member of the technical staff at AT&T Labs Research. His research interests include managing and working with massive data, including summarization, sharing, and privacy. Cormode has a PhD in computer science from the University of Warwick. Contact him at graham@research.att.com.

S. MUTHUKRISHNAN is a professor of computer science at Rutgers University. His research interests include algorithms and game theory. Muthukrishnan has a PhD in computer science from New York University. He's a fellow of the ACM. Contact him at muthu@cs.rutgers.edu.

different subsets of the data (after agreeing on the values for w and d and on the hash functions to use) and combine these sketches to give the sketch of the union of the data. Sketch combination is straightforward: given sketch arrays of size $w \times d$, we combine them by summing them up, entry by entry.

This implies that sketches can be useful for large-scale data analysis in a distributed model such as MapReduce. Each machine can build and emit a sketch of its local data. We can then combine these at a single machine (a single reducer simply sums up the sketches by entry) to generate the sketch of a potentially huge collection of data. This approach can be dramatically more efficient in terms of network communication (and hence time and other resources) than computing exact counts for each item and filtering out the low counts.

Hardware Implementations

Researchers have implemented Count-Min sketches in hardware to further accelerate their performance. The sketching algorithms' simplicity and parallelism make such implementations convenient. Yu-Kuen Lai and Gregory Byrd implemented Count-Min sketches on a low-power stream processor that could process 40-byte packets at up to 13 Gbps throughput.[4] This was equivalent to approximately 44 million updates per second. When using IBM's cell processor, Dina Thomas and colleagues observed nearly perfect parallel speedup (for example, using eight processing units would produce nearly eightfold speedup).[5] Using a platform based on field-programmable gate arrays, Lai and his colleagues implemented Count-Min sketches for anomaly detection.[6] Their implementation scaled easily to 4-Gbps network data streams.

Applications Using Count Tracking

The Count-Min sketch data structure goes beyond the task of approximating data distributions. Let's look at some examples.

A possible query is to identify heavy hitters—that is, the query HH(k) returns those items with a large frequency (say, $1/k$ of the overall count N). Count tracking can directly answer this query by estimating each item's frequency in turn and finding those with the largest estimated counts. When many items are possible, answering the query in this way can be slow. Keeping additional information about the frequencies of groups of items can speed up the process[1] (at the expense of storing additional sketches).

Finding heavy hitters is also of interest in signal processing. Viewing the signal as defining a data distribution, recovering the heavy hitters is key to building the best approximation of the signal. So, we can use the Count-Min sketch in compressed sensing, a signal acquisition paradigm that has recently revolutionized signal processing.[7]

One application that can involve very large datasets is natural language processing. Here, it's important to keep statistics on the frequency of word combinations, such as pairs or triplets of word sequences. In one experiment, researchers compacted a 90-Gbyte corpus down to a memory-friendly 8-Gbyte Count-Min sketch.[8] This proved just as effective for their word similarity tasks as using the exact data.

Another application is to design mechanisms to help users pick safe passwords. To make password guessing difficult, we can track the frequency of passwords online and disallow popular ones. This is precisely the count-tracking problem. Recently, researchers put this into practice using the Count-Min sketch (see www.youtube.com/watch?v=qo1cOJFEF0U). With this solution, a false positive—erroneously declaring a rare password choice to be too popular and therefore disallowing it—only mildly inconveniences users.

The Count-Min sketch has found a plethora of applications in a variety of different areas. It demonstrates that it's possible to build compact summary representations of data that retain key properties.

As we increasingly have to deal with "big data," this will become even more important. Over time, we expect to see sketches like Count-Min being increasingly used and available in most standard language libraries.

Acknowledgments
US National Science Foundation grant 0916782 supported this research.

References

1. G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *J. Algorithms*, vol. 55, no. 1, 2005, pp. 58–75.
2. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms,* MIT Press, 1990.
3. M. Thorup, "Even Strongly Universal Hashing Is Pretty Fast," *Proc. 11th Ann. ACM-SIAM Symp. Discrete Algorithms* (SODA 00), SIAM, 2000, pp. 496–497.
4. Y.-K. Lai and G.T. Byrd, "High-Throughput Sketch Update on a Low-Power Stream Processor," *Proc. ACM/IEEE Symp. Architecture for Networking and Communications Systems* (ANCS 06), ACM Press, 2006, pp. 123–132.
5. D. Thomas et al., "On Efficient Query Processing of Stream Counts on the Cell Processor," *Proc. 25th IEEE Int'l Conf. Data Eng.* (ICDE 09), IEEE CS Press, 2009, pp. 748–759.
6. Y.-K. Lai et al., "Implementing On-line Sketch-Based Change Detection on a NetFPGA Platform," *Proc. 1st Asia NetFPGA Developers Workshop*, ACM, 2010.
7. A. Gilbert and P. Indyk, "Sparse Recovery Using Sparse Matrices," *Proc. IEEE*, vol. 98, no. 6, 2010, pp. 937–947.
8. A. Goyal et al., "Sketch Techniques for Scaling Distributional Similarity to the Web," *Proc. Workshop Geometrical Models of Natural Language Semantics* (GEMS 10), Assoc. Computational Linguistics, 2010, pp. 51–56.

IEEE Software

HOW TO REACH US

WRITERS
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

LETTERS TO THE EDITOR
Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

ON THE WEB
www.computer.org/software

SUBSCRIBE
www.computer.org/software/subscribe

SUBSCRIPTION CHANGE OF ADDRESS
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software.*

MEMBERSHIP CHANGE OF ADDRESS
Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

MISSING OR DAMAGED COPIES
If you are missing an issue or you received a damaged copy, contact help@computer.org.

REPRINTS OF ARTICLES
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

REPRINT PERMISSION
To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.