Approximating Data with the Count-Min Data Structure

Graham Cormode S. Muthukrishnan

August 12, 2011

1 Introduction

Algorithmic problems such as tracking the contents of a set arise frequently in the course of building systems. Given the variety of possible solutions, the choice of appropriate data structures for such tasks is at the heart of building efficient and effective software. Modern languages are now augmented with large libraries of algorithms and data structures to help the programmer without forcing her to reinvent the wheel. These offer familiar data structures and methods that address a host of problems using heaps, hash tables, tree structures, stacks, and so on.

For a precise example, consider the following *membership* problems, which is fundamental in Computer Science. Starting from an empty set S, there is a series of update operations drawn from $\textsc{Insert}(S, i)$ (performs $S \leftarrow S \cup \{i\}$);), or $\textsc{Delete}(S, i)$ (performs $S \leftarrow S - \{i\}$)), and interleaved therein are queries $\textsc{Check}(S, i)$ (which returns `yes` if $i \in S$ and `no` otherwise). The membership problem is to maintain the set S under the update operations and respond to queries, using small space and making each operation as fast as possible. There is extensive research on solving the membership problem using data structures from hash tables to balanced trees and B-tree indices, and these form the backbone of systems from OSs, compilers, databases and beyond. Many of these data structures have been in widespread use for forty or more years.

Modern applications now face the need to handle massive data. For these applications, many of the existing data structures that are suitable for main memory or disk resident data no longer suffice; they motivate new problems and have required an entirely new set of data structures. It is desirable that these data structures be relatively small, and in many cases we require them to be *sub-linear* in the size of the input. Also, in many cases, it suffices to provide approximate responses to various queries, and applications work without precise answers. In what follows, we present a fundamental problem at the heart of such applications. The data structures they inspire are called "sketches" and these approximate massive data sets. We present one such data structure called the Count-Min Sketch [6] that has been invented recently and has found wide applications from IP networking to machine learning, distributed computing and even signal processing and beyond.

We begin by introducing the problem of *Count Tracking*, which generalizes membership, and is at the heart of modern data processing scenarios. In this problem, there are a large number of items, and an associated frequency for each item which changes over time. The query specifies an item and the response to the query is the frequency of the item. Here is an example use-case in the context of tracking search queries:

Consider a popular website which wants to keep track of statistics on the queries used to search the site. One could keep track of the full log of queries, and answer exactly the frequency of any search query at the site. However, the log can quickly become very large.. This problem 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 up the multiple occurrences of the same query, can prove to be slow and wasteful of resources. Notice that in this scenario, we can tolerate a little imprecision. In general, we are interested only in the queries that are asked frequently. So it is acceptable if there is some fuzziness in the counts. Thus, we can tradeoff some precision in the answers for a more efficient and lightweight solution. This tradeoff is at the heart of sketches.

Other use-case examples of count tracking abound. In an online retailer scenario, the items might represent goods for sale, and the associated frequency is 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 of that stock within a given day; and so on. Further, count tracking can be applied on derived data such as the difference of data between distributed sites or between different time periods and so on. Even more generally, many tasks on massive data distributions such as summarizing, mining, classification, anomaly detection, and others, can be solved using count tracking.

Formally, any data structure for Count Tracking must provide two methods: UPDATE(i,c), which updates the frequency of item i by c; and ESTIMATE(i), which provides an estimate of the current frequency of i.

This problem can be solved exactly using traditional data structures: a hash table, for example, can be used to keep the set of items and associated frequencies [4]. UPDATE and ESTIMATE can both be implemented directly via standard hash table methods. However, there are disadvantages to such solutions: the amount of memory used by the data structure can become very large over time as more items are added. Because its size is large, accessing such a data structure can be quite slow, since it resides in slow memory or in virtual memory. Further, since its size grows over time, periodically the data structure has to be resized, which can affect real-time processing. Finally, in distributed applications, if the entire frequency distribution has to be communicated, this can be a prohibitive overhead.

Sketch data structures overcome this problem by observing that in many cases it is reasonable to replace the exact answer with a high-quality approximation. For instance, in presenting statistics on customer buying patterns, an uncertainty of 0.1% (or less) is not considered significant. Such sketch data structures can accurately summarize arbitrary data distributions with a compact, fixed memory footprint that is often small enough to fit within cache, ensuring fast processing of updates, or quick communication across sites.

2 The Count-Min Sketch

Sketch overview. The Count-Min sketch provides a different kind of solution to count tracking. It allocates a fixed amount of space to store count information, which does not vary over time even as more and more counts are updated. Nevertheless, it is able to provide useful estimated counts, because the accuracy scales with the total sum of all the counts stored. If N represents the current

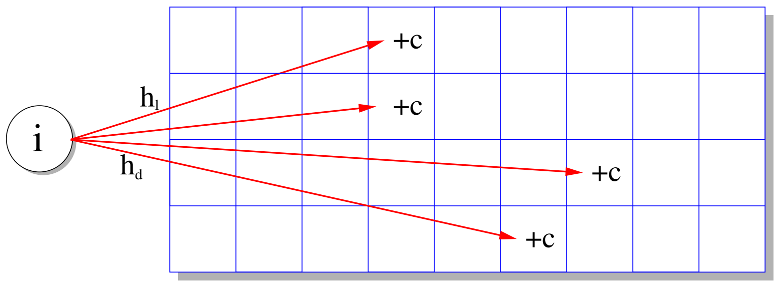

Figure 1: Schematic of Count-Min sketch data structure

sum of all the counts (i.e. the sum of all the c values in UPDATE operations), then it promises a distortion which is a very 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 is 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 data structure primarily consists of a fixed array of counters, of width w and depth d. The counters are initialized to all zeros. Each row of counters is associated with a different hash function. The hash function maps items uniformly onto the range $\{1, 2, \ldots w\}$. The hash functions do not need to be particularly strong (i.e. they are not as complex as cryptographic hash functions). For items represented as integers i, the hash functions can be of the form $(a * i + b \mod p \mod w)$, where p is a prime number larger than the maximum i value (say, $p = 2^{31} - 1$ or $p = 2^{61} - 1$), and a, b are values chosen randomly in the range 1 to $p - 1$. It is important that each hash function is different, otherwise there is no benefit from the repetition.

UPDATE(i,c) operations update the data structure in a straightforward way. In each row, the corresponding hash function is applied to i to determine a corresponding counter. Then the update c is added on to that counter. Figure 1 shows an example of an update operation on a sketch with $w = 9$ and $d = 4$. The update of item i is mapped by the first hash function to an entry in the first row, where the update of c is added on to the current counter there. Similarly, the item is mapped to different locations in each of the other three rows. So in this example, we evaluate four different hash functions on i, and update four counters accordingly.

For ESTIMATE(i) operations, the process is quite similar. For each row, the corresponding hash function is applied to i to look up one of the counters. Across all rows, the estimate is found as the minimum of all the probed counters. In the example above, we examine each place where i was mapped by the hash functions: in 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 so on. Each of these entries has a counter which has added up all the updates that were mapped there, and the estimate returned is the smallest of these.

Why it works. At first, it may be unclear why this process should give any usable estimate of counts. It seems that since each counter is counting the updates to many different items, the estimates will inevitably be inaccurate. However, the hash functions work to spread out the different items, so on average the inaccuracy cannot be too high. Then the use of different hash functions ensures that actually the chance of getting an estimate that is much more inaccurate than average is actually quite small. The sidebar on Sketch Accuracy makes this mathematically precise. As a result, for a sketch of size $w \times d$ with total count N, it follows that any estimate has error at most $2N/w$, with probability at least $1 - \frac{1}{2}^d$. So setting the parameters w and d large enough allows us to achieve very high accuracy while using relatively little space.

Sketch Accuracy. The quality of the estimates given by this sketch can be guaranteed using some statistical analysis. Observe that in a given row, the counter probed by the ESTIMATE operation includes the current frequency of item i. However, because w is typically smaller than the total number of items summarized, there will be hash collisions: the count found will be the sum of frequencies of all items mapped by the hash function to that location. In traditional hash tables, such hash collisions are problematic, but in this case, they can be tolerated, since overall an accurate estimate is still found.

Because the hash function spreads items uniformly around 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 total sum of all counts is N, then the expected fraction of weight colliding with i is at most N/w. In some cases, we might be lucky, and the colliding weight will be less than this; in other cases, we might be unlucky, and the colliding weight will be more. However, the colliding weight is unlikely to be much larger than expected amount: the probability of seeing more than twice the expected amount is at most $\frac{1}{2}$ (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 probability at least $\frac{1}{2}$.

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

From this analysis, we know how to set the parameters for the size of the sketch. Suppose we want an error of at most 0.1% (of the sum of all frequencies), with 99.9% certainty. Then we want $2/w = 1/1000$, we set $w = 2000$, and $\frac{1}{2}^d = 0.001$, i.e. $d = \log 0.001 / \log 0.5 \leq 10$. Using 32 bit counters, the space required by the array of counters is $w \times d \times 4 = 80KB$.

3 Implementations of the Sketch

We now discuss implementations of the Count-Min Sketch in different settings: a traditional single-threaded case, parallel and distributed implementation, and implementation using other hardware.

3.1 Single Threaded Implementation

As indicated above, it is straightforward to implement the sketch in a traditional single CPU environment. Indeed, several libraries are available for the data structure, in common languages

```
Algorithm 1: CountMinInit($w, d, p$)
1 $C[1,1] \ldots C[d,w] \leftarrow 0$;
2 for $j \leftarrow 1$ to $d$ do
3     Pick $a_j, b_j$ uniformly from [1... p];
4 $N = 0$;
```

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

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

Figure 2: Pseudocode for Count-Min Sketch

such as C, C++, Java and Python (see pointers in Section 5). Some skeletal pseudocode to initialize and update the sketch is shown in Figure 2. The code in Algorithm 1 initializes the array C of $w \times d$ counters to 0, and picks values for the hash functions based on the prime p. For each Update(i, c) shown 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) shown 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 be used to make the code as fast as possible. In particular, the hash function used in line 3 has two "modulus" operations, which are perceived as being slow. Both can be removed: the $\mod p$ operation can be replaced with a shift and an add for certain choices of p (see [13] for a concise guide), and the $\mod w$ can be replaced with a bitmask operation when w is chosen to be a power of 2. Under these settings, the sketch can be updated at the rate of millions of operations per second, approaching the IO limit.

3.2 Parallel and Distributed Implementation

A key feature of the operations which manipulate the sketch is that they are largely *oblivious* to the current state of the data structure. That is, the updates to the sketch do not require inspection of the current state. This means that data structure is highly suitable for parallelization and distributed computation.

Firstly, each row of the sketch is updated independently of others, so the sketch can be partitioned row-wise among threads on a single machine. But more than this, one can build sketches of different subsets of the data (after agreeing on the same parameters w, d and set of hash functions to use), and these sketches can be combined to give the sketch of the union of the data. Sketch combination is straightforward: given sketch arrays of size $w \times d$, they are combined by summing them up, entry-wise.

This implies that sketches can be a useful tool in large scale data analysis, within a distributed model such as MapReduce. Each machine can build and "emit" a sketch of its local data, and these can then be combined at a single machine (i.e. a single reducer simply sums up all the sketches, entry-wise) 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 the solution of computing exact counts for each item, and filtering out the low counts.

3.3 Hardware Implementation

There have been several efforts to implement sketch data structures in hardware, to further accelerate their performance. The simplicity and parallelism of the sketching algorithms makes such implementations convenient. Here, we outline some of the approaches taken:

— Lai and Byrd report on an implementation of Count-Min sketches on a low-power stream processor [9], capable of processing 40 byte packets at a throughput rate of up to 13Gbps. This is equivalent to about 44 million updates per second.

— Thomas *et al.* [12] describe their experience using IBM's cell processor. They observe near perfect parallel speed-up, i.e. using 8 processing units, there is a nearly 8-fold speed up.

— Lai *et al.* present an implementation of sketching techniques using an FPGA-based platform, for the purpose of anomaly detection [10]. Their implementation scales easily to network data stream rates of 4Gbps.

4 Applications using Count Tracking

There are dozens of applications of count tracking and in particular, the Count-Min sketch data structure that goes beyond the task of approximating data distributions. We give three examples.

1. A more general query is to identify the *Heavy-Hitters*, that is, the query HH(k) returns the set of items which have large frequency (say $1/k$ of the overall frequency). Count tracking can be used to directly answer this query, by considering the frequency of each item. When there are very many possible items, answering the query in this way can be quite slow. The process can be sped up immensely by keeping additional information about the frequencies of groups of items [6], at the expense of storing additional sketches. As well as being of interest in mining applications, finding heavy-hitters is also of interest in the context of signal processing. Here, viewing the signal as defining a data distribution, recovering the heavy-hitters is key to building the best approximation of the signal. As a result, the Count-Min sketch can be used in compressed sensing, a signal acquisition paradigm that has recently revolutionized signal processing [7].

2. One application where very large data sets arise is in Natural Language Processing (NLP). Here, it is important to keep statistics on the frequency of word combinations, such as pairs or triplets of words that occur in sequence. In one experiment, researchers compacted a large

90GB corpus down to a (memory friendly) 8GB Count-Min sketch [8]. This proved to be just as effective for their word similarity tasks as using the exact data.

3. A third example is in designing a mechanism to help users pick a safe password. To make password guessing difficult, we can track the frequency of passwords online and disallow currently popular ones. This is precisely the count tracking problem. Recently, this was put into practice using the Count-Min data structure to do count tracking (see `http://www.youtube.com/watch?v=qo1cOJFEF0U`). A nice feature of this solution is that the impact of a false positive—erroneously declaring a rare password choice to be too popular and so disallowing it—is only a mild inconvenience to the user.

5 Further Reading

Example Implementations. Several example implementations of the Count-Min sketch are available. C code is given by the MassDal code bank: `http://www.cs.rutgers.edu/~muthu/massdal-code-index.html`. C++ code due to Marios Hadjieleftheriou is available from `http://research.att.com/~marioh/sketches/index.html`. OCAML code due to Edward Yang is at `https://github.com/ezyang/ocaml-cminsketch`. There is a SQL implementation for distributed, high performance settings, due to Berkeley and Greenplum researchers `http://doc.madlib.net/sketches_8sql__in_source.html`.

More Reading on Count-Min Data Structure. For more on the Count-Min sketch, see the web-page collecting information on the data structure, at `https://sites.google.com/site/countminsketch/`. There is more technical coverage in the original paper describing the structure [6], in textbooks on randomized algorithms [11], and in a survey of techniques for the count tracking problem and its variations [5].

More Reading on Other Sketches. There are several other useful and compact sketch data structures which solve different problems. These include:

- The *Bloom Filter* is a popular sketch data structure that solves the slightly simpler membership problem. Bloom filters have size linear in the number of items, but use smaller memory than standard hashing approach to the membership problem. The Bloom Filter is highly popular in networking applications, for tracking which flows have been seen, and which objects are stored in caches. For more information, see the survey of Mitzenmacher and Broder [3].
- The *AMS Sketch* can be used to represent very high dimensional *vectors* in a small amount of space. Such vectors often occur in machine learning applications, where each entry of the vector encodes the presence or absence of some feature. The sketch allows the inner product (aka the cosine distance) of two vectors to be estimated accurately [1].
- Various *Distinct Sketches* have been introduced to track other functions of sets of items. The most basic question they answer is to estimate the number of different elements within the set. For example, they can track the number of unique visitors to a website. Unlike the Bloom Filter, they do not 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 [2].

While several sketch data structures have been proposed in the literature differing in how they summarize data, their approximation guarantees, and how they apply to specific problems, we have found the Count-Min sketch data structure to be uniformly good in performance, and versatile in broad applicability to many problems.

About the Authors

Graham Cormode is a principal member of technical staff at AT&T Labs–Research. His research interests include all aspects of 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, and a Fellow of the ACM. His research interests include algorithms and game theory. Muthukrishnan has a PhD in Computer Science from New York University. Contact him at muthu@cs.rutgers.edu.

References

[1] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *ACM Principles of Database Systems*, pages 10–20, 1999.

[2] K. Beyer, R. Gemulla, P. J. Haas, B. Reinwald, and Y. Sismanis. Distinct-value synopses for multiset operations. *Communications of the ACM*, 52(10):87–95, 2009.

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

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] G. Cormode and M. Hadjieleftheriou. Finding the frequent items in streams of data. *Communications of the ACM*, 52(10):97–105, 2009.

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

[7] A. Gilbert and P. Indyk. Sparse recovery using sparse matrices. *Proceedings of the IEEE*, 98(6):937–947, June 2010.

[8] A. Goyal, J. Jagarlamudi, H. D. III, and S. Venkatasubramanian. Sketch techniques for scaling distributional similarity to the web. In *Workshop on GEometrical Models of Natural Language Semantics*, 2010.

[9] Y.-K. Lai and G. T. Byrd. High-throughput sketch update on a low-power stream processor. In *Proceedings of the ACM/IEEE symposium on Architecture for networking and communications systems*, 2006.

[10] Y.-K. Lai, N.-C. Wang, T.-Y. Chou, C.-C. Lee, T. Wellem, and H. T. Nugroho. Implementing on-line sketch-based change detection on a netfpga platform. In *1st Asia NetFPGA Developers Workshop*, 2010.

[11] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[12] D. Thomas, R. Bordawekar, C. C. Aggarwal, and P. S. Yu. On efficient query processing of stream counts on the cell processor. In *IEEE International Conference on Data Engineering*, 2009.

[13] M. Thorup. Even strongly universal hashing is pretty fast. In *ACM-SIAM Symposium on Discrete Algorithms*, 2000.