Sketching and Streaming Algorithms for Processing Massive Data

Jelani Nelson*

Several modern applications require handling data so massive that traditional algorithmic models do not provide accurate means to design and evaluate efficient algorithms. Such models typically assume that all data fits in memory, and that running time is accurately modeled as the number of basic instructions the algorithm performs. However in applications such as online social networks, large-scale modern scientific experiments, search engines, online content delivery, and product and consumer tracking for large retailers such as Amazon and Walmart, data too large to fit in memory must be analyzed. This consideration has led to the development of several models for processing such large amounts of data: the *external memory model* [3, 21] and *cache-obliviousness* [13, 19], where one aims to minimize the number of blocks fetched from disk, *property testing* [14], where it is assumed that the data is so massive that we do not wish to even look at it all and thus aim to minimize the number of probes made into the data, and *massively parallel algorithms* operating in such systems as MapReduce and Hadoop [6, 9]. Also in some applications, data arrives in a *streaming* fashion and must be processed on the fly as it arrives. Such cases arise for example with packet streams in network traffic monitoring, or query streams arriving at a web-based service such as a search engine.

In this article we focus on this latter *streaming model* of computation, where a given algorithm must make one pass over a data set to then compute some function. We pursue such streaming algorithms which use memory that is significantly sublinear in the amount of data, since we assume that the data is too large to fit in memory. Sometimes it is also useful to consider algorithms that are allowed not just one, but a few passes over the data, in cases where for example the data set lives on disk and the number of

*`minilek@princeton.edu`. Supported by NSF CCF-0832797.

passes may dominate the overall running time. We also occasionally discuss *sketches*. A sketch is with respect to some function f, and a sketch of a data set x is a compressed representation of x from which one can compute $f(x)$. Of course under this definition $f(x)$ is itself a valid sketch of x, but we often require more of our sketch than just being able to compute $f(x)$. For example, we typically require that it be possible for the sketch to be updated as more data arrives, and sometimes we also require that sketches of two different data sets that were prepared independently can be compared to compute some function of the aggregate data, or similarity or difference measures across different data sets.

Our goal in this article is not to be comprehensive in our coverage of streaming algorithms. Rather, we discuss in some detail just a few surprising results in order to convince the reader that it is possible to obtain some non-trivial algorithms in this model. Those interested in learning more about this area are encouraged to read the surveys [8, 18], or view the notes online for streaming courses taught by Amit Chakrabarti at Dartmouth, Piotr Indyk at MIT, and Andrew McGregor at UMass Amherst.

1 Probabilistic counting

How many bits does it take to store an integer between 1 and n? The answer is clearly $\lceil \log_2 n \rceil$ bits, else two integers would map to the same bitstring and be indistinguishable. But what if we only care about recovering the integer up to a constant factor? Then it suffices to only recover $\lceil \log n \rceil$, and storing $\lceil \log n \rceil$ only requires $O(\log \log n)$ bits.

This observation was behind one of the oldest known streaming algorithms, invented in 1977 by Robert Morris [17], former chief scientist of a division of the NSA (and father of the inventor of the first Internet worm). Consider the streaming problem where we see a stream of n increments. We would like to compute n, though approximately, and with some potential small probability of failure. We could keep an explicit counter in memory and increment it after each stream update, but that would require $\lceil \log_2 n \rceil$ bits. Morris' clever algorithm works as follows: initialize a counter c to 1, and after each update increment c with probability $1/2^c$ and do nothing otherwise. Philippe Flajolet showed that the expected value of 2^c is $n+2$ after n updates [11], and thus $2^c - 2$ is an unbiased estimator of n. The same work showed the variance is bounded such that $2^c - 2$ will be within a constant factor of n with constant probability. By a combination of averaging many independent estimators, as well as attempting to store $\log_{1+\gamma} n$ in memory instead of $\log_2 n$ for some small $\gamma > 0$ by incrementing with higher probability, it is possible to obtain more precise approximations of n in small memory with very large probability.

2 Frequent Items

A common desired ability in many software systems is the ability to track "hot" items. For example, Google Trends tracks which search queries and topics have been the most popular over a recent time window. Large ISPs like AT&T want to monitor IP traffic being routed through their network to understand, for example, which servers are receiving the largest amounts of traffic. Such knowledge can help in detecting Denial of Service attacks, as well as designing their network infrastructure to minimize costs. For companies serving similar or identical content to large numbers of users, such as Akamai or Dropbox, it may be beneficial to detect whether certain content becomes hot, i.e. frequently downloaded, to know which files to place on servers that are faster or have connections with higher bandwidth.

The formal setup of this problem is as follows. There is some stream of tokens $i_1, \ldots, i_m$ with each i_j coming from some fixed set of size n (e.g. the set of all 2^{32} IPv4 IP addresses, or the set of all queries in some dictionary). Let us just suppose that this fixed set is $[n]$.[1] For some $0 < \varepsilon \leq 1/2$ known to the algorithm at the beginning of the stream, we would like to report all indices $i \in [n]$ such that i appeared in the stream more than εm times. This formalization models the examples above: a query stream coming into Google, a packet stream going through a router, or a stream of downloads over time made from some content delivery service.

One of the oldest streaming algorithms for detecting frequent items is the MJRTY algorithm invented by Boyer and Moore in 1980 [7]. MJRTY makes the following guarantee: if some $i \in [n]$ appears in the stream a strict majority of the time, it will be found. If this guarantee does not hold, MJRTY may output anything. Note that if given a second pass over the stream, one can verify whether the output index actually is a majority index. Thus, MJRTY solves the frequent items problem for $\varepsilon = 1/2$.

[1] By $[n]$ we mean the set $\{1, \ldots, n\}$.

Before describing MJRTY, first consider the following means of carrying out an election. We have m voters in a room, each voting for some candidate $i \in [n]$. We ask the voters to run around the room and find one other voter to pair up with who voted for a different candidate (note: some voters may not be able to find someone to pair with, for example if everyone voted for the same candidate). Then, we kick everyone out of the room who *did* manage to find a partner. A claim whose proof we leave to the reader as an exercise is that *if* there actually was a candidate with a strict majority, then some non-zero number of voters will be left in the room at the end, and furthermore all these voters will be supporters of the majority candidate.

The MJRTY algorithm is simply the streaming implementation of the election procedure in the previous paragraph. We imagine an election official sitting at the exit door, processing each voter one by one. When the next voter is processed, he will either be asked to sit aside amongst a pool of people waiting to be paired off (clearly everyone in this pool supports the same candidate, else the official could pair two people in the pool with each other and kick them out of the room), or he will be paired up with someone in the pool and removed. Now when a new voter approaches the official, one of several things may happen. If the pool is empty, the official adds him to the pool. If the pool is not empty and he is voting for a different candidate than everyone in the pool, then the official grabs someone from the pool, pairs them off, and kicks them both out of the room. Else if his vote agrees with the pool, the official adds him to the pool. If the pool is non-empty at the end, then the candidate the pool supports is labeled the majority candidate. Note that this algorithm can be implemented to discover the majority by keeping track of only two things: the size of the pool, and the name of the candidate everyone in the pool supports. Maintaining these two integers requires at most $\lceil \log_2 n \rceil + \lceil \log_2 m \rceil$ bits of memory.

What about general $\varepsilon < 1/2$? A natural generalization of the MJRTY algorithm was invented by Jayadev Misra and David Gries in 1982 [16] (and has been rediscovered at least a couple times since then [10, 15]). Rather than pair off voters supporting different candidates, this algorithm tells the voters to form groups of size exactly k such that no two people in the same group support the same candidate. Then everyone who managed to make it into a group of size exactly k is kicked out of the room. It can be shown that any candidate receiving strictly more than m/k votes will be supported by one of the last candidates standing, so we can set $k = \lceil 1/\varepsilon \rceil$. Furthermore, a simple extension of the MJRTY algorithm implementation using $k - 1$

ID/counter pairs (and thus using $O(k \log(n+m))$ bits of space) provides a streaming algorithm. When a new voter comes, if he matches any candidate in the pool then we increment that counter by one. Else, we decrement *all* counters by one (corresponding to forming a group of size k and removing them).

3 Distinct Elements

On July 19, 2001 a variant of the Code Red worm began infecting machines vulnerable to a certain exploit in an older version of the Microsoft IIS webserver. The worm's activities included changing the website hosted by the infected webserver to display

HELLO! Welcome to http://www.worm.com! Hacked By Chinese!

as well as an attempted Denial of Service attack against `www1.whitehouse.gov`.

In August 2001 while trying to track the rate at which the worm was spreading, David Moore and Colleen Shannon at The Cooperative Association for Internet Data Analysis (CAIDA) needed to track the number of distinct IP addresses sending traffic on particular links whose packets contained the signature of the Code Red worm. This setup turns out to precisely be an instantiation of the *distinct elements* problem introduced and studied by Philippe Flajolet and G. Nigel Martin [12]. In this problem, one has a stream of elements $i_1, i_2, \ldots, i_m$ each being an integer in the set $\{1, \ldots, n\}$. Then, given one pass over this stream, one must compute F_0, the number of *distinct* integers amongst the i_j. In the case of tracking the Code Red worm, $n = 2^{32}$ is the number of distinct IP addresses in IPv4, and m is the number of packets traversing a monitored link while carrying the signature of the worm. Aside from network traffic monitoring applications, the distinct elements problem naturally arises in several other domains: estimating the number of distinct IP addresses visiting a website, or number of distinct queries made to a search engine, or to estimate query selectivity in the design of database query optimizers.

An obvious solution to the distinct elements problem is to maintain a bitvector x of length n, where we initialize $x = 0$ then set $x_i = 1$ if we ever see i in the stream. This takes n bits of memory. Another option is to remember the entire stream, taking $O(m \log n)$ bits. In fact Alon, Matias, and Szegedy [2] showed that $\Omega(\min\{n, m\})$ bits is necessary for this problem unless slack is allowed in two ways:

1. **Approximation**: We do not promise to output F_0 exactly, but rather some estimate $\tilde{F}_0$ such that $|\tilde{F}_0 - F_0| \le \varepsilon F_0$.

2. **Randomization**: Our algorithm may output a wrong answer with some small probability.

Our goal is now to produce such an estimate $\tilde{F}_0$ which is within εF_0 of F_0 with probability at least $2/3$. This success probability can be amplified arbitrarily by taking the median estimate of several independent parallel runs of the algorithm. Our goal is to design an algorithm using $f(1/\varepsilon) \cdot \log n$ bits of memory, e.g. $O(\log n)$ memory for a constant factor approximation. Note that just storing an index in $[n]$ requires $\lceil \log_2 n \rceil$ bits, which we presume fits in a single machine word, so we are aiming to use just a constant number of machine words!

In fact such an algorithm is possible. Suppose for a minute that we had access to a perfectly random hash function h mapping $[n]$ to the continuous interval $[0, 1]$[2]. We maintain a single number X in memory: the minimum value of $h(i)$ we have ever encountered over all i appearing in the stream. One can show that the expected minimum value satisfies

$$\mathbb{E}X = 1/(F_0 + 1).$$

Thus a natural estimator is to output $1/X - 1$. Unfortunately a calculation shows that the standard deviation of X is almost equal to its expectation, so that $1/X - 1$ is poorly concentrated around a good approximation of F_0. This can be remedied by letting $\bar{X}$ be the average of many independently such X's obtained independently at random in parallel, then instead returning $1/\bar{X} - 1$. A more efficient remedy was found by Bar Yossef et al. [4], and further developed (and named, as the *KMV algorithm*[3]) by Beyer et al. [5]. The algorithm maintains the k minimum hash values for $k = O(1/\varepsilon^2)$. Now let X_k denote the kth minimum hash value. Then

$$\mathbb{E}X_k = k/(F_0 + 1),$$

[2] In reality one would work with a sufficiently fine discretization of this interval since computers can only store numbers to finite precision

[3] KMV stands for "k minimum values".

and the returned estimate is thus given as $k/X_k - 1$. This algorithm can be shown to return a value satisfying the desired approximation guarantees with large constant probability.

4 Linear Sketches

In some situations we do not simply want to compute on data coming into a single data stream, but on multiple datasets coming from multiple data streams. For example, we may want to compare traffic patterns across two different time periods, or collected at two different parts of the network. Another motivating force is parallelization: split a single data stream into several to farm out computation to several machines, then combine the sketches of the data these machines have computed later to recover results on the entire data set.

One way of accomplishing the above is to design streaming algorithms that use *linear sketches*. Suppose we are interested in a problem which can be modeled in the following *turnstile model*. We have a vector $x \in \mathbb{R}^n$ that receives a stream of coordinate-wise updates of the form $x_i \leftarrow x_i + v$ (v may be positive or negative). We then at the end of the stream want to approximate $f(x)$ for some function f. For example in the distinct elements problem, v is always 1 and $f(x) = |\{i : x_i \neq 0\}|$. A streaming algorithm using a linear sketch is then one whose memory contents can be viewed as Ax for some (possibly random) matrix A. Unfortunately the algorithms discussed above do not operate via linear sketches, but now we will see examples where this is the case.

Join size estimation When querying a relational database there can be multiple ways of executing the query to obtain the result, for example by taking advantage of associativity. Database query optimizers then try to cheaply estimate a plan to use to answer the query so as to minimize the time required. For queries involving joins or self-joins, such optimizers make use of size estimates of these joins to estimate intermediate table sizes. Ideally we would like to obtain these estimates from a short sketch that can fit in cache and thus be updated quickly as data is inserted into the database.

Let us formally define this problem. We have an attribute A and domain D, and for $i \in D$ we let x_i denote the frequency of i in A. We will assume that $D = [n]$. The self-join size on this attribute is then $\|x\|_2^2 = \sum_i x_i^2$, and thus we simply want to estimate the squared ℓ_2 norm of a vector being updated in a data stream. In fact this general problem has a wider range of applicability. For example, noting that $\|x\|_2^2$ is sensitive to heavy coordinates, AT&T used ℓ_2-norm estimation to detect traffic anomalies where servers were receiving too much traffic, signaling potential Denial of Service attacks (in this case x_i is the number of packets sent to IP address i).

The "AMS sketch" of Alon, Matias, and Szegedy [1, 2] provides a low-memory streaming algorithm for estimating $\|x\|_2^2$. Suppose we had a random hash function $h : [n] \to \{-1, 1\}$. We initialize a counter X to 0, and when some value v is added to x_i we increment X by $v \cdot h(i)$. Thus at the end of the stream, $X = \sum_i x_i \cdot h_i$. It can be shown that $\mathbb{E}X^2 = \|x\|_2^2$ and that the variance satisfies $\mathbb{E}(X^2 - \mathbb{E}X^2)^2 \le 2\|x\|_2^4$. By keeping track of k such counters $X_1, \ldots, X_k$ each using independent random hash functions h_i and averaging the X_i^2, we obtain an unbiased estimator with smaller variance. Standard tools like Chebyshev's inequality then imply that if $k = O(1/\varepsilon^2)$ then the average of the X_i^2 will be within $\varepsilon\|x\|_2^2$ of $\|x\|_2^2$ with large constant probability. Note that this is a linear sketch using a $k \times n$ matrix A, where $A_{i,j} = h_i(j)/\sqrt{k}$ and our estimate of $\|x\|_2^2$ is $\|Ax\|_2^2$.

5 Pseudorandomness

One caveat in many of the algorithms presented above is our assumption that the hash functions used be *random*. There are t^n functions mapping $[n]$ to $[t]$, and thus a random such function requires at least $n \log_2 t$ bits to store. In applications where we care about small-memory streaming algorithms, n is large, and thus even if we find an algorithm using sublinear space it would then not be acceptable to use an additional n bits of space or more to store the hash function needed by the algorithm.

The above consideration thus pushes streaming algorithm designers to look for hash functions which are not actually fully random, but only "random enough" to ensure that the algorithms in which they are being used will still perform correctly. There are at the highest level two directions one can then take to find such hash functions. One is to make some complexity theoretic assumption, for example to assume that no efficient algorithm exists for some concrete computational problem, then to construct hash functions that can be proven sufficient based on the assumption made. The other direction is to construct hash functions that are provably sufficiently random without making any such assumptions. This latter direction is for obvious reasons typically harder to implement but is possible in certain applications, such as for all the problems mentioned above. This area of constructing objects (such as functions) that look "random enough" for various computational tasks is known as *pseudorandomness*, and the interested reader may wish to read [20].

6 Biography

Jelani Nelson recently finished his PhD in computer science at the Massachusetts Institute of Technology in 2011. He is currently a postdoctoral researcher in theoretical computer science at Princeton University and the Institute for Advanced study and will begin as an assistant professor of computer science at Harvard University in 2013. His research interests are primarily in algorithms for processing massive data.

References

[1] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.*, 64(3):719–747, 2002.

[2] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[3] Lars Arge. External memory data structures. In *Handbook of Massive Data Sets*. Kluwer, 2002.

[4] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop (RANDOM)*, pages 1–10, 2002.

[5] Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, Berthold Reinwald, and Yannis Sismanis. Distinct-value synopses for multiset operations. *Commun. ACM*, 52(10):87–95, 2009.

[6] Dhruba Borthakur. The Hadoop distributed file system: Architecture and design. http://hadoop.apache.org/common/docs/r0.17.2/hdfs_design.html (last accessed June 25, 2012).

[7] Robert S. Boyer and J Strother Moore. MJRTY - a fast majority vote algorithm. In R.S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[8] Graham Cormode. Sketch techniques for massive data. In Graham Cormode, Minos Garofalakis, Peter Haas, and Chris Jermaine, editors, *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*, Foundations and Trends in Databases. NOW publishers, 2011.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[10] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of Internet packet streams with limited space. In *ESA*, pages 348–360, 2002.

[11] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985.

[12] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[13] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[14] Oded Goldreich. Combinatorial property testing (a survey). *Electronic Colloquium on Computational Complexity (ECCC)*, 4(56), 1997.

[15] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[16] Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[17] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978.

[18] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[19] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[20] Salil Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science, to appear.*

[21] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.