Finding Frequent Items in Data Streams

Moses Charikar*[1], Kevin Chen**[2], and Martin Farach-Colton[3]

[1] Princeton University
moses@cs.princeton.edu
[2] UC Berkeley
kevinc@cs.berkeley.edu
[3] Rutgers University and Google Inc.
martin@google.com

Abstract. We present a 1-pass algorithm for estimating the most frequent items in a data stream using very limited storage space. Our method relies on a novel data structure called a COUNT SKETCH, which allows us to estimate the frequencies of all the items in the stream. Our algorithm achieves better space bounds than the previous best known algorithms for this problem for many natural distributions on the item frequencies. In addition, our algorithm leads directly to a 2-pass algorithm for the problem of estimating the items with the largest (absolute) change in frequency between two data streams. To our knowledge, this problem has not been previously studied in the literature.

1 Introduction

One of the most basic problems on a data stream [HRR98,AMS99] is that of finding the most frequently occurring items in the stream. We shall assume here that the stream is large enough that memory-intensive solutions such as sorting the stream or keeping a counter for each distinct element are infeasible, and that we can afford to make only one pass over the data. This problem comes up in the context of search engines, where the streams in question are streams of queries sent to the search engine and we are interested in finding the most frequent queries handled in some period of time.

A wide variety of heuristics for this problem have been proposed, all involving some combination of sampling, hashing, and counting (see [GM99] and Section 2 for a survey). However, none of these solutions have clean bounds on the amount of space necessary to produce good approximate lists of the most frequent items. In fact, the only algorithm for which theoretical guarantees are available is the straightforward SAMPLING algorithm, in which a uniform random sample of the data is kept. For this algorithm, the space bound depends on the distribution of the frequency of the items in the data stream. Our main contribution is a simple algorithm with good theoretical bounds on its space requirements that also beats the naive sampling approach for a wide class of common distributions.

* This work was done while the author was at Google Inc.
** This work was done while the author was at Google Inc.

Before we present the details of our result, however, we need to introduce some definitions. Let $\mathcal{S} = q_1, q_2, \ldots, q_n$ be a data stream, where each $q_i \in \mathcal{O} = \{o_1, \ldots, o_m\}$. Let object o_i occur n_i times in $\mathcal{S}$, and order the o_i so that $n_1 \geq n_2 \geq \cdots \geq n_m$. Finally, let $f_i = n_i/n$.

We consider two notions of approximating the frequent-elements problem:

FindCandidateTop$(\mathcal{S}, k, l)$

- Given: An input stream $\mathcal{S}$, and integers k and l.
- Output: A list of l elements from $\mathcal{S}$ such that the k most frequent elements occur in the list.

Note that for a general input distribution, FindCandidateTop$(\mathcal{S}, k, l)$ may be very hard to solve. Suppose, for example, that $n_k = n_{l+1} + 1$, that is, the kth most frequent element has almost the same frequency as the $l + 1$st most frequent element. Then it would be almost impossible to find only l elements that are likely to have the top k elements. We therefore define the following variant:

FindApproxTop$(\mathcal{S}, k, \epsilon)$

Given: An input stream $\mathcal{S}$, integer k, and real ϵ.
Output: A list of k elements from $\mathcal{S}$ such that every element i in the list has $n_i > (1 - \epsilon)n_k$.

A somewhat stronger guarantee on the output is that every item o_i with $n_i > (1 + \epsilon)n_k$ will be in the output list, w.h.p.. Our algorithm will, in fact, achieve this stronger guarantee. Thus, it will only err on the boundary cases.

A summary of our final results are as follows: We introduce a simple data structure called a COUNT SKETCH, and give a 1-pass algorithm for computing the count sketch of a stream. We show that using a count sketch, we reliably estimate the frequencies of the most common items, which directly yields a 1-pass algorithm for solving FindApproxTop$(\mathcal{S}, k, \epsilon)$. The Sampling algorithm does not give any bounds for this version of the problem. For the special case of Zipfian distributions, we also give bounds on using our algorithm to solve FindCandidateTop$(\mathcal{S}, k, ck)$ for some constant c, which beat the bounds given by the Sampling algorithm for reasonable values of n, m and k.

In addition, our count sketch data structure is additive, i.e. the sketches for two streams can be directly added or subtracted. Thus, given two streams, we can compute the difference of their sketches, which leads directly to a 2-pass algorithm for computing the items whose frequency changes the most between the streams. None of the previous algorithms can be adapted to find max-change items. This problem also has a practical motivation in the context of search engine query streams, since the queries whose frequency changes most between two consecutive time periods can indicate which topics people are currently most interested in [Goo].

We defer the actual space bounds of our algorithms to the Section 4. In Section 2, we survey previous approaches to our problem. We present our algorithm for constructing count sketches in Section 3, and in Section 4, we analyze the space requirements of the algorithm. In Section 4.2, we show how the algorithm can be adapted to find elements with the largest change in frequency. We conclude in Section 5 with a short discussion.

2 Background

The most straightforward solution to the FindCandidateTop$(\mathcal{S}, k, l)$ problem is to keep a uniform random sample of the elements stored as a list of items plus a counter for each item. If the same object is added more than once, we simply increment its counter, rather than adding a new object to the list. We refer to this algorithm as the Sampling algorithm. If x is the size of the sample (counting repetitions), to ensure that an element with frequency f_k appears in the sample, we need to set x/n, the probability of being included in the sample, to be $x/n > O(\log n/n_k)$, thus $x > O(\log n/f_k)$. This guarantees that all top k elements will be in the sample, and thus gives a solution to FindCandidateTop$(\mathcal{S}, k, O(\log n/f_k))$.

Two variants of the basic sampling algorithm were given by Gibbons and Matias [GM98]. The concise samples algorithm keeps a uniformly random sample of the data, but does not assume that we know the length of the data stream beforehand. Instead, it begins optimistically assuming that we can include elements in the sample with probability $\tau = 1$. As it runs out of space, it lowers τ until some element is evicted from the sample, and continues the process with this new, lower τ'. The invariant of the algorithm is that, at any point, each item is in the sample with the current threshold probability. The sequence can be chosen arbitrarily to adapt to the input stream as it is processed. At the end of the algorithm, there is some final threshold τ_f, and the algorithm gives the same output as the Sampling algorithm with this inclusion probability. However, the value of τ_f depends on the input stream in some complicated way, and no clean theoretical bound for this algorithm is available.

The counting samples algorithm adds one more optimization based on the observation that so long as we are setting aside space for a count of an item in the sample anyway, we may as well keep an exact count for the occurrences of the item after it has been added to the sample. This change improves the accuracy of the counts of items, but does not change who will actually get included in the sample.

Fang et al. [FSGM$^+$96] consider the related problem of finding all items in a data stream which occur with frequency above some fixed threshold, which they call *iceberg queries*. They propose a number of different heuristics, most of which involve multiple passes over the data set. They also propose a heuristic 1-pass multiple-hash scheme which has a similar flavor to our algorithm.

Though not directly connected, our algorithm also draws on a quite substantial body of work in data stream algorithms [FKSV99,FKSV00,GG$^+$02]

[GMMO00,HRR98,Ind00]. In particular, Alon, Matias and Szegedy [AMS99] give an $\Omega(n)$ lower bound on the space complexity of any algorithm for estimating the frequency of the largest item given an arbitrary data stream. However, their lower bound is brittle in that it only applies to the FindCandidateTop$(\mathcal{S}, 1, 1)$ problem and not to the relaxed versions of the problem we consider, for which we achieve huge space reduction. In addition, they give an algorithm for estimating the second frequency moment, $F_2 = \Sigma_{i=1}^{m} n_i^2$, in which they use the idea of random ± 1 hash functions that we use in our algorithm (see also [Ach01]).

3 The Count Sketch Algorithm

Before we give the algorithm itself, we begin with a brief discussion of the intuition behind it.

3.1 Intuition

Recall that we would like a data structure that maintains the approximate counts of the high frequency elements in a stream and is compact.

First, consider the following simple algorithm for finding estimates of all n_i. Let s be a hash function from objects to $\{+1, -1\}$ and let c be a counter. While processing the stream, each time we encounter an item q_i, update the counter $c \mathrel{+}= s[q_i]$. The counter then allows us to estimate the counts of all the items since $\mathbf{E}[c \cdot s[q_i]] = n_i$. However, it is obvious that there are a couple of problems with the scheme, namely that, the variance of every estimate is very large, and $O(m)$ elements have estimates that are wrong by more than the variance.

The natural first attempt to fix the algorithm is to select t hash functions $s_1, \ldots, s_t$ and keep t counters, $c_1, \ldots, c_t$. Then to process item q_i we need to set $c_j \mathrel{+}= s_j[q_i]$, for each j. Note that we still have that each $\mathbf{E}[c_i \cdot s_i[q_i]] = n_i$. We can then take the mean or median of these estimates to achieve an estimate with lower variance.

However, collisions with high frequency items, like o_1, can spoil most estimates of lower frequency elements, even important elements like o_k. Therefore rather than having each element update every counter, we replace each counter with a hash table of b counters and have the items update different subsets of counters, one per hash table. In this way, we will arrange matters so that every element will get enough high-confidence estimates – those untainted by collisions with high-frequency elements – to estimate its frequency with sufficient precision.

As before, $\mathbf{E}[h_i[q] \cdot s[q]] = n_q$. We will show that by making b large enough, we will decrease the variance to a tolerable level, and that by making t large enough – approximately logarithmic in n – we will make sure that each of the m estimates has the desired variance.

3.2 Our algorithm

Let t and b be parameters with values to be determined later. Let $h_1, \ldots, h_t$ be hash functions from objects to $\{1, \ldots, b\}$ and $s_1, \ldots, s_t$ be hash functions from objects to $\{+1, -1\}$. The CountSketch data structure consists of these hash functions along with a $t \times b$ array of counters, which should be interpreted as an array of t hash tables, each containing b buckets.

The data structure supports two operations:

ADD(C, q)**:** For $i \in [1, t], h_i[q] \mathrel{+}= s_i[q]$.
ESTIMATE(C, q)**:** return median$_i\{h_i[q] \cdot s_i[q]\}$.

Why do we take the median instead of the mean? The answer is that even in the final scheme, we have not eliminated the problem of collisions with high-frequency elements, and these will still spoil some subset of the estimates. The mean is very sensitive to outliers, while the median is sufficiently robust, as we will show in the next section.

Once we have this data structure, our algorithm is straightforward and simple to implement. For each element, we use the CountSketch data structure to estimate its count, and keep a heap of the top k elements seen so far. More formally:
Given a data stream $q_1, \ldots, q_n$, for each $j = 1, \ldots, n$:

1. ADD(C, q_j)
2. If q_j is in the heap, increment its count. Else, add q_j to the heap if ESTIMATE(C, q_j) is greater than the smallest estimated count in the heap. In this case, the smallest estimated count should be evicted from the heap.

This algorithm solves FINDAPPROXTOP$(\mathcal{S}, k, \epsilon)$, where our choice of b will depend on ϵ. Also, notice that if two sketches share the same hash functions – and therefore the same b and t – that we can add and subtract them. The algorithm takes space $O(tb + k)$. In the next section we will bound t and b.

4 Analysis

To make the notation easier to read, we will sometimes drop the subscript of q_i and simply write q, when there is no ambiguity. We will further abuse the notation by conflating q with its index i.

We will assume that each hash function h_i and s_i is pairwise independent. Further, all functions h_i and s_i are independent of each other. Note that the amount of randomness needed to implement these hash functions is $O(t \log m)$. We will use $t = O\left(\log \frac{n}{\delta}\right)$, where the algorithm fails with probability at most δ. Hence the total randomness needed is $O\left(\log m \log \frac{n}{\delta}\right)$.

Consider the estimation of the frequency of an element at position ℓ in the input. Let $n_q(\ell)$ be the number of occurrences of element q up to position ℓ. Let $A_i[q]$ be the set of elements that hash onto the same bucket in the ith row as q does, i.e. $A_i[q] = \{q' : q' \neq q, h_i[q'] = h_i[q]\}$. Let $A_i^{>k}[q]$ be the elements of

$A_i[q]$ other than the k most frequent elements, i.e. $A_i^{>k}[q] = \{q' : q' \neq q, q' > k, h_i[q'] = h_i[q]\}$. Let $v_i[q] = \sum_{q' \in A_i[q]} n_{q'}^2$. We define $v_i^{>k}[q]$ analogously for $A_i^{>k}[q]$.

Lemma 1. *The variance of $h_i[q]s_i[q]$ is bounded by $v_i[q]$.*

Lemma 2. $E[v_i^{>k}[q]] = \dfrac{\sum_{q'=k+1}^{m} n_{q'}^2}{b}$.

Let SMALL-VARIANCE$_i[q]$ be the event that $v_i^{>k}[q] \leq \dfrac{8\sum_{q'=k+1}^{m} n_{q'}^2}{b}$. By the Markov inequality,

$$\mathbf{Pr}[\text{SMALL-VARIANCE}_i[q]] \geq 1 - \frac{1}{8} \tag{1}$$

Let NO-COLLISIONS$_i[q]$ be the event that $A_i[q]$ does not contain any of the top k elements.

If $b \geq 8k$,

$$\mathbf{Pr}[\text{NO-COLLISIONS}_i[q]] \geq 1 - \frac{1}{8} \tag{2}$$

Let SMALL-DEVIATION$_i[q](\ell)$ be the event that

$$|h_i[q]s_i[q] - n_q(\ell)|^2 \leq 8\,\mathbf{Var}[h_i[q]s_i[q]].$$

Then,

$$\mathbf{Pr}[\text{SMALL-DEVIATION}_i[q](\ell)] \geq 1 - \frac{1}{8}. \tag{3}$$

By the union bound,

$$\begin{aligned}\mathbf{Pr}[\text{NO-COLLISIONS}_i[q] \text{ and } \text{SMALL-VARIANCE}_i[q] \\ \text{and } \text{SMALL-DEVIATION}_i[q]] \geq \frac{5}{8}\end{aligned} \tag{4}$$

We will express the error in our estimates in terms of a parameter γ, defined as follows:

$$\gamma = \sqrt{\frac{\sum_{q'=k+1}^{m} n_{q'}^2}{b}}. \tag{5}$$

Lemma 3. *With probability $(1 - \frac{\delta}{n})$,*

$$|median\{h_i[q]s_i[q]\} - n_q(\ell)| \leq 8\gamma \tag{6}$$

Proof. We will prove that, with high probability, for more than $\frac{t}{2}$ indices $i \in [1,t]$,

$$|h_i[q]s_i[q] - n_q(\ell)| \leq 8\gamma$$

This will imply that the median of $h_i[q]s_i[q]$ is within the error bound claimed by the lemma. First observe that for an index i, if all three events NO-COLLISIONS$_i[q]$, SMALL-VARIANCE$_i[q]$, and SMALL-DEVIATION$_i[q]$] occur, then $|h_i[q]s_i[q] - n_q(\ell)| \leq 8\gamma$. Hence, for a fixed i,

$$\mathbf{Pr}[|h_i[q]s_i[q] - n_q(\ell)| \leq 8\gamma] \geq \frac{5}{8}.$$

The expected number of such indices i is at least $5t/8$. By Chernoff bounds, the number of such indices i is more than $t/2$ with probability at least $1 - e^{O(t)}$. Setting $t = \Omega(\log(n) + \log(\frac{1}{\delta}))$, the lemma follows.

Lemma 4. *With probability $1-\delta$, for all $\ell \in [1,n]$,*

$$|median\{h_i[q]s_i[q]\} - n_q(\ell)| \leq 8\gamma \tag{7}$$

where q is the element that occurs in position ℓ.

Lemma 5. *If $b \geq 8 \max\left(k, \frac{32\sum_{q'=k+1}^{m} n_{q'}^2}{(\epsilon n_k)^2}\right)$, then the estimated top k elements occur at least $(1-\epsilon)n_k$ times in the sequence; further all elements with frequencies at least $(1+\epsilon)n_k$ occur amongst the estimated top k elements.*

Proof. By Lemma 4, the estimates for number of occurrences of all elements are within an additive factor of 8γ of the true number of occurrences. Thus for two elements whose true number of occurrences differ by more than 16γ, the estimates correctly identify the more frequent element. By setting $16\gamma \leq \epsilon n_k$, we ensure that the only elements that can replace the true most frequent elements in the estimated top k list are elements with true number of occurrences at least $(1-\epsilon)n_k$.

$$\begin{aligned} 16\gamma &\leq \epsilon n_k \\ \Leftrightarrow 16\sqrt{\frac{\sum_{q'=k+1}^{m} n_{q'}^2}{b}} &\leq \epsilon n_k \\ \Leftrightarrow b &\geq \frac{256\sum_{q'=k+1}^{m} n_{q'}^2}{(\epsilon n_k)^2} \end{aligned}$$

This combined with the condition $b \geq 8k$ used to prove (2), proves the lemma.

We conclude with the following summarization:

Theorem 1. *The* COUNT SKETCH *algorithm solves* FINDAPPROXTOP*$(\mathcal{S}, k, \epsilon)$ in space*

$$O\left(k\log\frac{n}{\delta} + \frac{\sum_{q'=k+1}^{m} n_{q'}^2}{(\epsilon n_k)^2}\log\frac{n}{\delta}\right)$$

4.1 Analysis for Zipfian distributions

Note that in the algorithm's (ordered) list of estimated most frequent elements, the k most frequent elements can only by preceded by elements with number of occurrences at least $(1-\epsilon)n_k$. Hence, by keeping track of $l \geq k$ estimated most frequent elements, the algorithm can ensure that the most frequent k elements are in the list. For this to happen l must be chosen so that $n_{l+1} < (1-\epsilon)n_k$. When the distribution is Zipfian with parameter z, $l = O(k)$ (in fact $l = k/(1-\epsilon)^{1/z}$). If the algorithm is allowed one more pass, the true frequencies of all the l elements in the algorithms list can be determined allowing the selection of the most frequent k elements.

In this section, we analyze the space complexity of our algorithm for Zipfian distributions. For a Zipfian distribution with parameter z, $n_q = \frac{c}{q^z}$ for some scaling factor c. (We omit c from the calculations)[1]. We will compare the space requirements of our algorithm with that of the sampling based algorithm for the problem FindCandidateTop$(\mathcal{S}, k, l)$. We will use the bound on b from Lemma 5, setting ϵ to be a constant so that, with high probability, our algorithms' list of $l = O(k)$ elements is guaranteed to contain the most frequent k elements. First note that

$$\sum_{q'=k+1}^{m} n_{q'}^2 = \sum_{q'=k+1}^{m} \frac{1}{(q')^{2z}} = \begin{cases} O(m^{1-2z}), & z < \frac{1}{2} \\ O(\log m), & z = \frac{1}{2} \\ O(k^{1-2z}), & z > \frac{1}{2} \end{cases}$$

Substituting this into the bound in Lemma 5 (and setting ϵ to be a constant), we get the following bounds on b (correct up to constant factors). The total space requirements are obtained by multiplying this by $O(\log \frac{n}{\delta})$.

Case 1: $z < \frac{1}{2}$.

$$b = m^{1-2z}k^{2z}$$

Case 2: $z = \frac{1}{2}$.

$$b = k \log m$$

Case 3: $z > \frac{1}{2}$.

$$b = k$$

We compare these bounds with the space requirements for the random sampling algorithm. The size of the random sample required to ensure that the k most frequent elements occur in the random sample with probability $1-\delta$ is

$$\frac{n}{n_k} \log(k/\delta).$$

We measure the space requirement of the random sampling algorithm by the expected number of distinct elements in the random sample. (Note that the size

[1] While c need not be a constant, it turns out that all occurrences of c cancel in our calculations, and so, for ease of presentation, we omit them from the beginning.

of the random sample could be much larger than the number of distinct elements due to multiple copies of elements).

Furthermore, the sampling algorithm as stated, solves the FindCandidateTop$(\mathcal{S}, k, x)$, where x is the number of distinct elements in the sample. This does not constitute a solution of FindCandidateTop$(\mathcal{S}, k, O(k))$, as does our algorithm. We will be reporting our bounds for the latter and the sampling bounds for the former. However, this only gives the sampling algorithm an advantage over ours.

We now analyze the space usage of the sampling algorithm for Zipfians. It turns out that for Zipf parameter $z \leq 1$, the expected number of distinct elements is within a constant factor of the sample size. We analyze the number of distinct elements for Zipf parameter $z > 1$.

Items are placed in the random sample S with probability $\frac{\log(k/\delta)}{n_k}$.

$$\mathbf{Pr}[q \in S] = 1 - \left(1 - \frac{\log(k/\delta)}{n_k}\right)^{n_q}$$

$$\begin{aligned}
\mathbf{E}[\text{no. of distinct elements in } S] &= \sum_{q=1}^{m} \mathbf{Pr}[q \in S] = \sum_{q=1}^{m} 1 - \left(1 - \frac{\log(k/\delta)}{n_k}\right)^{n_q} \\
&= \sum_{q=1}^{m} O\left(\min\left(1, \frac{n_q \log(k/\delta)}{n_k}\right)\right) \\
&= \sum_{q=1}^{m} O\left(\min\left(1, \frac{k^z \log(k/\delta)}{q^z}\right)\right) \\
&= O\left(k\left(\log \frac{k}{\delta}\right)^{1/z}\right)
\end{aligned}$$

The bounds for the two algorithms are compared in Table 1.

Zipf parameter	random sampling	Count Sketch Algorithm
$z < \frac{1}{2}$	$m\left(\frac{k}{m}\right)^z \log \frac{k}{\delta}$	$m^{1-2z}k^{2z} \log \frac{n}{\delta}$
$z = \frac{1}{2}$	$\sqrt{km} \log \frac{k}{\delta}$	$k \log m \log \frac{n}{\delta}$
$\frac{1}{2} < z < 1$	$m\left(\frac{k}{m}\right)^z \log \frac{k}{\delta}$	$k \log \frac{n}{\delta}$
$z = 1$	$k \log m \log \frac{k}{\delta}$	$k \log \frac{n}{\delta}$
$z > 1$	$k\left(\log \frac{k}{\delta}\right)^{\frac{1}{z}}$	$k \log \frac{n}{\delta}$

Table 1. Comparison of space requirements for random sampling vs. our algorithm

4.2 Finding items with largest frequency change

For object q and sequence S, let n_q^S be the number of occurrences of q in S. Given two streams S_1, S_2, we would like to find the items q such that the values of $|n_q^{S_2} - n_q^{S_1}|$ are the largest amongst all items q. We can adapt our algorithm for finding most frequent elements to this problem of finding elements whose frequencies change the most.

We make two passes over the data. In the first pass, we only update counters. In the second pass, we actually identify elements with the largest changes in number of occurrences.

We first make a pass over S_1, where we perform the following step:

For each q, for $i \in [1, t], h_i[q] -= s_i[q]$.

Next, we make a pass over S_2, doing the following:

For each q, for $i \in [1, t], h_i[q] += s_i[q]$.

We make a second pass over S_1 and S_2:
For each q,

1. $\hat{n}_q = \text{median}\{h_i[q]s_i[q]\}$.
2. Maintain set A of the l objects encountered with the largest values of $|\hat{n}_q|$.
3. For every item $q \in A$ maintain an exact count of the number of occurrences in S_1 and S_2.

(Note that though A can change, items once removed are never added back. Thus accurate exact counts can be maintained for all q currently in A).

Finally, we report the k items with the largest values of $|n_q^{S_2} - n_q^{S_1}|$ amongst the items in A.

We can give a guarantee similar to Lemma 5 with n_q replaced by $\Delta_q = |n_q^{S_1} - n_q^{S_2}|$.

5 Conclusions

We make a final note comparing the Count Sketch algorithm with the Sampling algorithm. So far, we have neglected the space cost of actually storing the elements from the stream. This is because different encodings can yield very different space use. Both algorithms need counters that require $O(\log n)$ bits, however, we only keep k objects from the stream, while the Sampling algorithm keeps a potentially much larger set of items from the stream. For example, if the space used by an object is Ψ, and we have a Zipfian with $z = 1$, then the sampling algorithm uses $O(k \log m \log \frac{k}{\delta} \Psi)$ space while the Count Sketch algorithm uses $O(k \log \frac{n}{\delta} + k\Psi)$ space. If $\Psi >> \log n$, as it will often be in practice, this give the Count Sketch algorithm a large advantage over the Sampling algorithm.

As for the max-change problem, we note that there is still an open problem of finding the elements with the max-percent change, or other objective functions that somehow balance absolute and relative changes.

References

[Ach01] Dimitris Achlioptas. Database-friendly random projections. In *Proc. 20th ACM Symposium on Principles of Database Systems*, pages 274–281, 2001.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[FKSV99] Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. An approximate l_1-difference algotihm for massive data streams. In *Proc. 40th IEEE Symposium on Foundations of Computer Science*, pages 501–511, 1999.

[FKSV00] Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. Testing and spot-checking of data streams. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 2000.

[FSGM+96] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey Ullman. Computing iceberg queries efficiently. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 307–317, 1996.

[GG+02] Anna Gilbert, , Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *to appear in Proc. 34th ACM Symposium on Theory of Computing*, 2002.

[GM98] Phillip Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 331–342, 1998.

[GM99] Phillip Gibbons and Yossi Matias. Synopsis data structures for massive data sets. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 909–910, 1999.

[GMMO00] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[Goo] Google. Google zeitgeist - search patterns, trends, and surprises according to google. http://www.google.com/press/zeitgeist.html.

[HRR98] Monika Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. Technical Report SRC TR 1998-011, DEC, 1998.

[Ind00] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, pages 148–155, 2000.