Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Theoretical Computer Science 312 (2004) 3–15

Theoretical Computer Science
www.elsevier.com/locate/tcs

Finding frequent items in data streams

Moses Charikar[a,*,1], Kevin Chen[b,1], Martin Farach-Colton[c]

[a] *Department of Computer Science, Princeton University, Princeton, NJ 08544, USA*
[b] *Computer Science Division, University of California, Berkeley, CA 94720, USA*
[c] *Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA*

Abstract

We present a 1-pass algorithm for estimating the most frequent items in a data stream using limited storage space. Our method relies on a data structure called a COUNT SKETCH, which allows us to reliably estimate the frequencies of frequent items in the stream. Our algorithm achieves better space bounds than the previously known best algorithms for this problem for several 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 latter problem has not been previously studied in the literature.

Keywords: Frequent items; Streaming algorithm; Approximation

1. Introduction

One of the most basic problems on a data stream [12,2] is that of finding the most frequently occurring items in the stream. We shall assume 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 only afford to process the data by making one or more passes over it. This problem comes up in the context

* Corresponding author.
E-mail addresses: moses@cs.princeton.edu, (M. Charikar), kevinc@cs.berkeley.edu (K. Chen), farach@cs.rutgers.edu (M. Farach-Colton).
[1] Most of this work was done while the authors were at Google Inc. Moses Charikar gratefully acknowledges support from NSF ITR Grant CCR-0205594 and DOE Early Career Principal Investigator award DE-FG02-02ER25540.

0304-3975/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0304-3975(03)00400-6

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. Other applications include load balancing in a distributed database and identifying large packet flows in a network router.

A wide variety of heuristics for this problem have been proposed, all involving some combination of sampling, hashing, and counting (see [8] 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 frequencies of the items in the data stream. Our main contribution is a simple algorithm with good theoretical bounds on its space requirements that beats the naïve sampling approach for a certain class of common distributions.

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

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

CandidateTop(S, k, l)

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

Note that for a general input distribution, CandidateTop(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 by scaling up the n_i's towards infinity, an adversary can make it arbitrarily difficult for an algorithm to distinguish between n_k and n_{l+1}. We therefore define the following variant:

ApproxTop(S, k, ε)

Given: An input stream S, integer k, and real number ε.
Output: A list of k elements from S such that every element o_i in the list has frequency $n_i > (1 - \varepsilon)n_k$.

A somewhat stronger guarantee on the output is that every item o_i with $n_i > (1 + \varepsilon)n_k$ will be in the output list, w.h.p. Our algorithm will, in fact, achieve this stronger guarantee. In other words, 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 ApproxTop(S, k, ε). The Sampling algorithm does not give any bounds for this version of the problem because there is no way to guarantee that the Sampling algorithm does not pick up low-frequency items and the problem requires a list of k high-frequency elements. For the special case of Zipfian distributions, we also give bounds on using our algorithm to solve CandidateTop(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 are increasing or decreasing in popularity at the fastest rate [10]. In the networking context, max-change items correspond to the flows whose transmission rate is changing most rapidly.

We defer the actual space bounds of our algorithms to Section 4. In Section 2, we survey related work. 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 CandidateTop(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 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 $x/n > O((\log k)/n_k)$, thus $x > O((\log k)/f_k)$. This guarantees that w.h.p. all top k elements will be in the sample, and thus gives a solution to CandidateTop$(S, k, O((\log k)/f_k))$.

Two variants of the basic sampling algorithm were given by Gibbons and Matias [7]. 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 of τ's 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 and the sequence of τ's 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. [4] 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. Manku and Motwani [15] study 1-pass sampling algorithms for iceberg queries while Karp et al. [14] study the related problem of finding all flows which make up more than some user-specified fraction of the total flow in the datastream. Their algorithm can be used for CandidateTop(S,k,l) but not ApproxTop(S,k,ε). We discuss their algorithm further in Section 4.1.

Though not directly connected, our algorithm also draws on a quite substantial body of work in data stream algorithms [5,6,9,11–13]. In particular, Alon et al. [2] 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 only applies to CandidateTop$(S,1,1)$ and not to the relaxed versions of the problem we consider, for which our algorithms use significantly less space. In addition, they give an algorithm for estimating the second frequency moment, $F_2=\sum_{i=1}^{m} n_i^2$, in which they use the idea of random ± 1 hash functions that we use in our algorithm (see also Achlioptas [1], who uses such hash functions for dimension reduction). Saks and Sun [16] prove an optimal approximation-space tradeoff of approximating the L^∞ distance between two vectors in the data stream model. This is related to our problem because the frequency of the most frequent item corresponds to the L^∞ norm of the data stream viewed as a vector, and the maximum absolute change in frequency of any item between two streams corresponds to the L^∞ distance between the vector representations of the two streams.

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 (pair-wise independent) 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 $\mathbf{E}[c\cdot s[q_i]]=\mathbf{E}\left[\sum_{j=1}^{m}(n_j\cdot s[q_j]\cdot s[q_i])\right]=n_i$ by pair-wise independence. 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 independent hash functions $s_1,\dots,s_t$ and keep t counters, $c_1,\dots,c_t$. 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 $\mathbf{E}[c_j \cdot s_j[q_i]] = n_i$ for each s_j. We can then take the mean or median of these estimates to achieve an estimate with lower variance.

However, the high-frequency items, like o_1, make large contributions to the variance in the estimates of lower frequency elements. This could introduce large errors in the estimates of certain elements we are interested in reporting, such as o_k. To fix this problem, rather than having each element update every counter, we replace each counter with a hash table of b counters plus an associated hash function h_j. Now the items update one of the b counters for each such 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, $\mathbf{E}[h_i[q] \cdot s_i[q]] = n_q$. (Here $h_i[q]$ denotes (the value of) the counter that object q maps to in the ith hash table.) 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.

Note that since the parameters of the data structure depend on the distribution, one needs to know some properties of the distribution before hand in order to actually implement the algorithm. On the other hand, implementing the SAMPLING algorithm also requires prior knowledge of the distribution.

3.2. Our algorithm

Let t and b be parameters with values to be determined later. Let $h_1,\dots,h_t$ be hash functions from objects to $\{1,\dots,b\}$ and $s_1,\dots,s_t$ be hash functions from objects to $\{+1,-1\}$. The s_i's should be pair-wise independent and all hash functions should be independent of each other. The COUNT SKETCH 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 (counters). The hash function h_i maps an object q to one of the b counters in the ith hash table. We use the notation $h_i[q]$ to represent this counter.

The data structure, denoted by C, supports two operations:

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

The operation ADD takes a new item q and increments or decrements the appropriate counter in each hash table, depending on the value of $s_i[q]$. The operation ESTIMATE returns the estimated count of an item by taking the median over all the counters associated with the item of the counter value multiplied by $s_i[q]$.

Note that our final estimate is obtained by taking the median of the estimates from the t hash tables, instead of the mean. This is because even in the final scheme, we have not completely eliminated the problem of collisions with high-frequency elements, and these will still introduce large errors in 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 COUNT SKETCH data structure to estimate its count, and keep a heap of the top k elements seen so far. More formally:

Given a data stream $q_1, \dots, q_n$, for each $j = 1, \dots, n$:

(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 APPROXTOP(S, k, ε), 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 $O(t \log m)$ bits. We will use $t = \Theta(\log(n/\delta))$, where the algorithm fails with probability at most δ. Hence the total number of random bits needed is $O(\log m \log n/\delta)$.

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 hash table as q does, i.e. $A_i[q] = \{q' \colon q' \neq q, h_i[q'] = h_i[q]\}$. Let $A_i^{>k}[q]$ be the elements of $A_i[q]$ other than the k most frequent elements, i.e. $A_i^{>k}[q] = \{q' \colon q' \neq q, q' > k, h_i[q'] = h_i[q]\}$. 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 $v_i[q]$.*

Proof. Recall that

$$h_i[q]s_i[q] = n_q(\ell) + s_i[q] \sum_{q' \in A_i[q]} n_{q'}(\ell) s_i[q']$$

and that by pairwise independence we get

$$\mathbf{E}[h_i[q]s_i[q]] = n_q(\ell).$$

Now, again by pairwise independence,

$$\mathbf{Var}[h_i[q]s_i[q]] = \mathbf{E}[(h_i[q]s_i[q] - n_q(\ell))^2] = \sum_{q' \in A_i[q]} n_{q'}(\ell)^2 = v_i[q]. \qquad \square$$

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

Let $\textsc{Small-Variance}_i[q]$ be the event that $v_i^{>k}[q] \leqslant (8\sum_{q'=k+1}^{m} n_{q'}^2)/b$. By the Markov inequality,

$$\mathbf{Pr}[\textsc{Small-Variance}_i[q]] \geqslant 1 - \tfrac{1}{8}. \tag{1}$$

Let $\textsc{No-Collisions}_i[q]$ be the event that $A_i[q]$ does not contain any of the top k elements.

If $b \geqslant 8k$,

$$\mathbf{Pr}[\textsc{No-Collisions}_i[q]] \geqslant 1 - \tfrac{1}{8}. \tag{2}$$

Let $\textsc{Small-Deviation}_i[q](\ell)$ be the event that

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

Then, again by Markov's inequality,

$$\mathbf{Pr}[\textsc{Small-Deviation}_i[q](\ell)] \geqslant 1 - \tfrac{1}{8}. \tag{3}$$

By the union bound,

$$\begin{aligned}&\mathbf{Pr}[\textsc{No-Collisions}_i[q] \text{ and } \textsc{Small-Variance}_i[q]\\ &\quad \text{and } \textsc{Small-Deviation}_i[q]] \geqslant \tfrac{5}{8}.\end{aligned} \tag{4}$$

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. *When t is set to $\Theta(\log n/\delta)$, then with probability $(1 - \delta/n)$,*

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

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

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

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 a particular index i, if all three events $\textsc{No-Collisions}_i[q]$, SMALL-VARIANCE$_i[q]$, and SMALL-DEVIATION$_i[q]$ occur, then $|h_i[q]s_i[q] - n_q(\ell)| \leqslant 8\gamma$. Hence, for a fixed i

$$\mathbf{Pr}[|h_i[q]s_i[q] - n_q(\ell)| \leqslant 8\gamma] \geqslant \tfrac{5}{8}.$$

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 $1 - e^{-\Theta(t)}$. Setting $t = \Omega(\log(n/\delta))$, the lemma follows. □

Lemma 4. *When t is set to $\Theta(\log n/\delta)$, then with probability $1-\delta$, for all $\ell \in [1,n]$,*

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

where q is the element that occurs in position ℓ.

Lemma 5. *If $b \geqslant 8\max(k, (32\sum_{q'=k+1}^{m} n_{q'}^2)/(\varepsilon n_k)^2)$, then the estimated top k elements occur at least $(1-\varepsilon)n_k$ times in the sequence; further all elements with frequencies at least $(1+\varepsilon)n_k$ 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 $16\gamma \leqslant \varepsilon n_k$, 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 $(1-\varepsilon)n_k$.

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

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

Putting together our setting of the parameter t with the bound on b from Lemma 5, we conclude with the following summarization.

Theorem 1. *The* COUNT SKETCH *algorithm, with t set to $\Theta(\log n/\delta)$, solves* APPROXTOP (S, k, ε) *in space*

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

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 $(1-\varepsilon)n_k$. Hence, by keeping track of $l \geqslant k$ 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 $n_{l+1} < (1-\varepsilon)n_k$. When the distribution is Zipfian with parameter z, $l = \mathrm{O}(k)$ (in fact $l = k/(1-\varepsilon)^{1/z}$). If the algorithm is allowed one more pass, the true frequencies of all the l elements in the algorithm's list can be determined, so the actual list of k most frequent elements can be correctly identified.

In this section, we analyze the space complexity of our algorithm for Zipfian distributions. We expect that Zipfian distributions will be good fits for the actual distributions seen in practice (e.g. search engine query streams or streams of packets [3]). For a Zipfian distribution with parameter z, $n_q = c/q^z$ for some scaling factor c. 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. We will compare the space requirements of our algorithm with that of the SAMPLING algorithm for the problem CANDIDATETOP(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 algorithm's list of $l = \mathrm{O}(k)$ elements is guaranteed to contain the most frequent k elements. First note that

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

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 $\mathrm{O}(\log n/\delta)$.

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 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 SAMPLING algorithm by the expected number of distinct elements in the random sample. (Note that the actual size 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 CANDIDATETOP(S,k,x), where x is the number of distinct elements in the sample. This does not constitute a solution of CANDIDATETOP$(S,k,\mathrm{O}(k))$, 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.

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

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

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

When $z > 1$ the sum is dominated by the first term, so

$$\mathbf{E}[\text{no. of distinct items in } S] = \mathrm{O}\left(k\left(\log\frac{n}{\delta}\right)^{1/z}\right).$$

When $z \leqslant 1$ the sum is dominated by the second term, so we get

$$\mathbf{E}[\text{no. of distinct items in } S] = \begin{cases} \mathrm{O}\left(m\left(\dfrac{k}{m}\right)^{z} \log\dfrac{k}{\delta}\right), & z < 1, \\ \mathrm{O}\left(k \log m \log\dfrac{k}{\delta}\right), & z = 1. \end{cases}$$

The bounds for the two algorithms are compared in Table 1. Our algorithm generally beats the SAMPLING algorithm for Zipfian distributions with parameter less than 1. Note that recent experimental work from the networking community indicates that the actual distribution of search engine queries is less than 1 [17].

Table 1
Comparison of space requirements for Sampling, KPS and our algorithm

Zipf parameter	Sampling	KPS	count sketch algorithm
$z<\frac{1}{2}$	$m\left(\frac{k}{m}\right)^{z}\log\frac{k}{\delta}$	$k^{z}m^{1-z}$	$m^{1-2z}k^{2z}\log\frac{n}{\delta}$
$z=\frac{1}{2}$	$\sqrt{km}\log\frac{k}{\delta}$	$\sqrt{km}$	$k\log m\log\frac{n}{\delta}$
$\frac{1}{2}<z<1$	$m\left(\frac{k}{m}\right)^{z}\log\frac{k}{\delta}$	$k^{z}m^{1-z}$	$k\log\frac{n}{\delta}$
$z=1$	$k\log m\log\frac{k}{\delta}$	$k^{z}\log m$	$k\log\frac{n}{\delta}$
$z>1$	$k\left(\log\frac{k}{\delta}\right)^{\frac{1}{z}}$	k^{z}	$k\log\frac{n}{\delta}$

In addition, we compare our results with those in the recent unpublished manuscript of Karp et al. [14], which gives a simple 1-pass deterministic algorithm for finding a superset of all items with frequency at least Θn for a user-specified parameter $0\leqslant\Theta\leqslant 1$, in $O(1/\Theta)$ space. While the algorithm does not solve ApproxTop since in general it returns many low frequency elements along with the high frequency ones, it does give a solution to CandidateTop by setting $\Theta = n_k/n$.

4.2. Finding items with the 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 during the course of the algorithm, once an item is removed it is 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 $O(k \log m \log(k/\delta)\Psi)$ space while the Count Sketch algorithm uses $O(k \log(n/\delta) + k\Psi)$ space. If $\Psi \gg \log n$, as it will often be in practice, this give the COUNT SKETCH algorithm an 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.

Acknowledgements

We thank the anonymous referees for their comments that helped improve the presentation.

References

[1] D. Achlioptas, Database-friendly random projections, in: Proc. 20th ACM Symp. on Principles of Database Systems, 2001, pp. 274–281.

[2] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, J. Comput. System Sci. 58 (1) (1999) 137–147.

[3] M.E. Crovella, M.S. Taqqu, A. Bestavros, Heavy-tailed probability distributions in the world wide web, in: Adler, Feldman, Taqqu (Eds.), A Practical Guide to Heavy Tails, Birkhäuser, Basel, 1998.

[4] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. Ullman, Computing iceberg queries efficiently, in: Proc. 22nd Internat. Conf. on Very Large Data Bases, 1996, pp. 307–317.

[5] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, An approximate l_1-difference algorithm for massive data streams, in: Proc. 40th IEEE Symp. on Foundations of Computer Science, 1999, pp. 501–511.

[6] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, Testing and spot-checking of data streams, in: Proc. 11th ACM-SIAM Symp. on Discrete Algorithms, 2000, pp. 165–174.

[7] P. Gibbons, Y. Matias, New sampling-based summary statistics for improving approximate query answers, in: Proc. ACM SIGMOD Internat. Conf. on Management of Data, 1998, pp. 331–342.

[8] P. Gibbons, Y. Matias, Synopsis data structures for massive data sets, in: Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms, 1999, pp. 909–910.

[9] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M. Strauss, Fast, small-space algorithms for approximate histogram maintenance, in: Proc. 34th ACM Symp. on Theory of Computing, 2002.

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

[11] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan, Clustering data streams, in: Proc. 41st IEEE Symp. on Foundations of Computer Science, 2000, pp. 359–366.

[12] M. Henzinger, P. Raghavan, S. Rajagopalan, Computing on data streams, Tech. Report SRC TR 1998-011, December 1998.

[13] P. Indyk, Stable distributions, pseudorandom generators, embeddings and data stream computation, in: Proc. 41st IEEE Symp. on Foundations of Computer Science, 2000, pp. 148–155.

[14] R. Karp, S. Shenker, C.H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags, ACM Transactions on Database Systems 28 (2003) 51–55.

[15] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proc. 28th Internat. Conf. on Very Large Data Bases, 2002.

[16] M. Saks, X. Sun, Space lower bounds for distance approximation in the data stream model, in: Proc. 34th ACM Symp. on Theory of Computing, 2002.

[17] Y. Xie, D. O'Hallaron, Locality for search engine queries and its implications for caching, in: Proc. INFOCOM, 2002.