An Improved Frequent Items Algorithm with Applications to Web Caching

Kevin Chen and Satish Rao

UC Berkeley

Abstract. We present a simple, intuitive algorithm for the problem of finding an approximate list of the k most frequent items in a data stream when the item frequencies are Zipf-distributed. Our result improves the previous result of Charikar, Chen and Farach-Colton [CCF02] for the same problem when the Zipf parameter is low. We also highlight an application of the algorithm to web caching that may be of practical interest.

1 Introduction

A number of papers [ABCO96,BCF$^+$99,CBC95,G94,SKW00,XO02] in the networking community have studied large traces of web accesses or search engine queries and concluded that the frequency of requests is well-approximated by a Zipf distribution, i.e. the frequency of the ith most frequent request is $\frac{c}{i^z}$ where c is a normalizing constant and $z > 0$ is referred to as the *Zipf parameter*[1]. They generally reported values of z in the range $(0, 1]$, which is very low compared to the Zipf parameters reported for other properties of the internet. These results have important implications in the design of efficient web caching schemes, since the distribution of requests determines, among other things, the size of the cache needed to achieve a given cache hit-ratio.

For the purposes of motivating this paper, suppose that we have a single cache for a group of web servers with enough storage space for k files. We make the assumption that each file has the same size. It is known that if each request is modelled as being drawn independently at random from a specific distribution, an assumption supported by [BCF$^+$99], then the optimum cache-replacement algorithm is LFU (Least Frequently Used) [CD73]. In other words, we should cache exactly the k most frequently accessed files. In practice, however, this replacement algorithm is complicated by the fact that it is too expensive to store a separate counter for each distinct file, and hence other cache-replacement algorithms are used, such as LRU (Least Recently Used). However, since there is good evidence that LFU can outperform LRU, especially when the cache size is small [BCF$^+$99], it would be interesting to find a way to implement it efficiently.

We cast this problem in the framework of finding frequent items in a data stream, a problem on which there is now a considerable body of work (see Section

[1] Some papers refer to this distribution as a *generalized Zipf* or *Zipf-like* distribution, reserving the name *Zipf distribution* for the case $z = 1$.

2, Related Work). In the data stream model, a stream S is an ordered sequence of n items drawn from an universe of m elements ($S \in [m]^n$). Define n_i to be the frequency of the ith most frequent item in the stream, so that A data stream algorithm is allowed one pass over the stream and must use sub-linear (preferably poly-logarithmic space and per-item processing time. We consider the following version of the frequent items problem, first defined in [CCF02]:
FindApproxTop(S, k, ϵ)

- Input: An input stream S, integer k, and real number ϵ.
- Output: A list of k elements from S such that every item o_i in the list has frequency $n_i > (1 - \epsilon)n_k$ and every item o_i with $n_i > (1 + \epsilon)n_k$ will be in the list with high probability.

In other words the algorithm must return a list of items, all of whose frequencies are relatively large with respect to the kth most frequent element and it cannot output items with very low frequencies.

Our solution to this problem is a simple data structure that builds off the Count Sketch data structure of Charikar, Chen and Farach-Colton [CCF02]. Our algorithm achieves better space bounds for small values of the Zipf parameter, z, as is appropriate for our web caching application.

2 Related Work

The general problem of finding frequent items in data streams has many obvious applications, including mining the most popular queries to a search engine [CCF02], internet congestion control [KPS03,DLM02], iceberg queries [FSG$^+$96] and association rules [MM02]. In general, the problem cannot be solved exactly in sublinear space since there is an $\Omega(m)$ lower bound for the problem of finding the maximum element exactly [AMS99].

As a result, numerous approximate versions of the problem have been formalized in the data stream literature. A good survey of frequent items algorithms can be found in [CM03]. Here we merely list several of the more important ones. Gibbons and Mattias [GM98] gave two simple algorithms for maintaining fixed-sized random samples of a stream which can then be used to estimate the most frequent items. However no theoretical guarentees for their algorithms are available. Fang et al. [FSG$^+$96] considered "iceberg queries" which in our framework correspond to items whose frequency is above some user-specified threshold. Their algorithms generally require multiple passes over the data stream. Manku and Motwani [MM02] gave improved algorithms for the iceberg query problem. Their algorithms have the guarentee that items with small frequency are never output. Later, Karp et al. [KPS03] and Demaine et al. [DLM02] independently showed that a classic algorithm for finding a *majority element* (an element that occurs at least half the time) can be generalized to find all elements whose frequency exceeds some user-specified fraction θ of the total length of the stream in space $\frac{1}{\theta}$. Finally, Cormode and Muthukrishan [CM03] gave an algorithm for finding items with frequency above some fraction of the total length of the stream based on the idea of *group testing*. Unlike all the previous algorithms, their algorithm works in the presence of delete operations, a feature more suited for their database applications than for the networking application that we consider here.

None of the above algorithms solve our version of the problem. All of them return elements with frequency above a certain threshold and thus are not guarenteed to return a list of k elements. In addition, for the algorithms of [KPS03,DLM02], the list of items returned may include items with arbitrarily small frequency. As a result, none of these solutions is suitable for our application. The only paper which studies the version of the problem that we consider is the original paper of Charikar, Chen and Farach-Colton [CCF02]. We describe their algorithm in depth in Section 3.

On the experimental side, a number of papers have studied the distribution of web accesses or search engine queries and concluded that the distribution was some kind of Zipf distribution. [ABCO96] found a Zipf parameter of 0.85 while [G94] and [CBC95] found Zipf parameters slightly under, but very close to 1. [XO02] did not explictly mention the Zipf parameter but it is implicit from their plots that the parameter is less than 1. [BCF$^+$99] measured six traces from a variety of sources and found good fits for all with various parameters, all less than 1.

3 The CCF Algorithm

Since our work is based heavily on the algorithm of Charikar, Chen and Farach-Colton (hereafter referred to as the CCF algorithm) and their Count Sketch data structure[CCF02], we begin by reviewing their work. Recall that n is the total length of the data stream seen so far, m is the total number of distinct elements in the data stream seen so far and n_i is the frequency of element i, ordered such that $n_1 \geq n_2 \geq \ldots \geq n_m$.

The Count Sketch data structure is a small sketch of the data stream which allows us to estimate the frequency of each item to within an additive factor of ϵn_k with probability $1-\delta$. Observe that this guarentee is sufficient to solve the $\textsc{FindApproxTop}(S, k, \epsilon)$ problem defined above. It consists of an array of $O(\log \frac{n}{\delta})$ hash tables plus two pairwise independent hash functions, $s(\cdot)$ and $h(\cdot)$, for each hash table. $h(\cdot)$ hashes items to buckets, while $s(\cdot)$ hashes items to ± 1 uniformly at random. Assume all hash functions are independent of one another.

The data structure implements two operations: *ADD* and *ESTIMATE*. To *ADD* a new element i to the Count Sketch, for each of the $O(\log \frac{n}{\delta})$ hash tables, we hash the item into a bucket using $h(i)$, and then update the count in that bucket with either $+1$ or -1 depending on the outcome of $s(i)$. It turns out that the pairwise independence of the hash functions implies that $s(i) \cdot h(i)$ is an unbiased estimator of n_i. To see this, suppose we want to estimate the frequency of item i. For each distinct item j, we define an "indicator" random variable x_j to be the item's frequency, n_j, if the item hashes into the same bucket as i and 0 otherwise. Then by the pairwise independence of $s(\cdot)$ and the independence of $h(\cdot)$ and $s(\cdot)$.

$$\begin{aligned}\mathbf{E}[s(i)\cdot h(i) \mid x_i = n_i] &= \mathbf{E}[s(i)\cdot\{n_i s(i) + \sum_{j\neq i} s(j)x_j\}] \\ &= n_i + \mathbf{E}[\sum_{j\neq i} s(i)s(j)x_j] \\ &= n_i + \sum_{j\neq i}\mathbf{E}[s(i)]\cdot\mathbf{E}[s(j)]\cdot\mathbf{E}[x_j] \\ &= n_i\end{aligned}$$

This proves the following lemma:

Lemma 1. *$s(i)\cdot h(i)$ is an unbiased estimator for n_i.*

We call each estimator $s(i)\cdot h(i)$ for each of our hash tables an *individual estimator*. Given the individual estimators, to *ESTIMATE* the value of n_i given an item i from the Count Sketch, we take the median of the $O(\log \frac{n}{\delta})$ individual estimators and return it as the final estimate.

The ultimate goal of the CCF algorithm is to return an approximate list of the top k items seen so far at any point in the data stream. To do so, it maintains a Count Sketch and a heap of the top k items as follows: whenever it sees a new item, it *ADDS* it to the Count Sketch and queries the data structure for an *ESTIMATE* of the frequency of the item. If the item is already in the heap, it increments its count. Otherwise, it compares the estimate to the frequency of the smallest item in the heap and if it is larger, adds the new item to the heap. It is easy to see that this procedure ensures that at any point in the stream, the heap maintains the top k elements seen so far.

3.1 Analysis of the CCF Algorithm

Intuitively, the algorithm works well if there are a few items in the stream that are very frequent and a large number of items with very low frequency. By choosing the hash table size large enough, we can ensure that the large items get spread out and do not corrupt each other's estimates with reasonable probability. Conditioning on this event occurring, each large item only collides with small items which then cancel each other out because of the random ± 1's.

The crucial parameter in the data structure is b, the number of buckets in each hash table. First, we set $b = \Omega(k)$. This ensures that with constant probability, none of the top k elements collide. Conditioning on this event occurring, we then proceed to compute the variance of the estimator and apply Chebyshev's inequality to bound the deviation of each estimator.

Lemma 2. *Conditioned on the event that none of the top k items collide with i, the variance of each individual estimator for n_i is bounded by $\frac{\sum_{j>k} n_j^2}{b}$.*

Proof.

$$\begin{aligned}\mathbf{Var}[s(i) \cdot h(i)|x_i = n_i] &= \mathbf{E}[(n_i + s(i)\sum_{j>k}\{s(j)x_j\} - n_i)^2] \\ &= \frac{1}{b}\,\mathbf{E}[\sum_{j>k} n_j^2 + \sum_{j\neq\ell}[n_j \cdot n_l \cdot s(j) \cdot s(\ell)]] \\ &= \frac{\sum_{j>k} n_j^2}{b}\end{aligned}$$

Again, by pairwise independence of the $s(k)$ and $s(j)$, the cross terms cancel.

Recall that our definition of FindApproxTop(S, k, ϵ) requires us to estimate the frequency of each element to within an additive error of ϵn_k. So, plugging our bound on the variance from Lemma 2 into Chebyshev's inequality, setting the deviation to be the ϵn_k and setting the error probability to be constant implies the following bound on b:

Lemma 3. $b = \Omega(\frac{\sum_{j>k} n_j^2}{(\epsilon n_k)^2})$

Proof.

$$\begin{aligned}\mathbf{Pr}[|h(i) \cdot s(i) - n_i| < \epsilon n_k] &< \frac{\mathbf{Var}[h(i) \cdot s(i)]}{(\epsilon n_k)^2} \\ &= \frac{1}{b} \cdot \frac{\sum_{j>k} n_j^2}{(\epsilon n_k)^2}\end{aligned}$$

Setting the probability to a constant and solving for b gives the desired result.

Thus far we have shown that the there are no collisions with constant probability and that conditioned on that, the deviation of the estimator is less than ϵn_k with constant probability. To achieve a high probability result, we take the median of $O(\log \frac{n}{\delta})$ individual estimators. The proof is by a standard Chernoff bound argument showing that if the expected number of individual estimators with small deviation is bounded away from $\frac{1}{2}$, then with high probability, at least $\frac{1}{2}$ of $O(\log \frac{n}{\delta})$ estimators have small deviation, which in turn implies that the median has small deiviation. We omit details from this extended abstract.

Lemma 4. *The median of the individual estimators has small deviation with probability* $O(1 - \frac{1}{poly(n)})$.

We note that the use of the median, as opposed to the mean, is crucial to the algorithm. Intuitively, each individual estimate could deviate from its expectation by a large amount, for example, if two large elements happen to collide in the same bucket. These "outliers" have a large affect the mean, making it unsuitable for a final estimator, whereas the median is robust to these outliers.

The final piece of the analysis is to take a union bound over all n positions in the stream to ensure that at every point in time, we can estimate the current item's frequency accurately and with high probability.

The final space bound is b multiplied by $O(\log \frac{n}{\delta})$, the number of hash tables. Putting together our bounds on b, this proves the main theorem of [CCF02]:

Theorem 1. *The CCF algorithm solves* $\textsc{FindApproxTop}(S, k, \epsilon)$ *in space*

$$O(k \log \frac{n}{\delta} + \frac{\sum_{j>k} n_j^2}{(\epsilon n_k)^2} \log \frac{n}{\delta})$$

While this theorem holds for arbitrary distributions on the item frequencies, we shall be particularly interested in analyzing the space bounds implied for low parameter Zipf distributions. We show these calculations in the next section after we present our algorithm.

4 Our Algorithm

Our algorithm keeps the basic outline of the CCF algorithm and merely modifies the Count Sketch data structure. We keep the $h(\cdot)$ hash functions from the Count Sketch data structure and the array of hash tables, but we discard the $s(\cdot)$ hash functions. Now, to *ADD an item to the sketch, we simply update the count of the appropriate bucket in each hash table by* $+1$. *To* ESTIMATE the frequency of an element, we take the count in the bucket, and subtract from it a constant $c = \frac{n}{b}$. Intuitively, c is the expected count of the bucket up to that point in the stream.

Although this algorithm seems much simpler than the CCF algorithm, somewhat surprisingly, it yields slightly better bounds, as we shall now show.

4.1 Analysis

For clarity, we shall denote our hash function $\hat{h}(\cdot)$ to differentiate it from $h(\cdot)$ for the CCF algorithm. Similarly we shall denote the size of each of our hash tables by $\hat{b}$. Define our individual estimator of item i to be

$$\hat{h}(i) = (\sum_j x_j) - \frac{n}{\hat{b}}$$

We start by again setting $b = \Omega(k)$ so that with constant probability, our item i does not collide with any of the other k top items. Conditioning on this event occurring, we see that

Lemma 5. *The expectation of each individual estimator of* i*, conditioned on the event that none of the top* k *items collide with* i*, is*

$$n_i - \frac{\sum_{j \leq k} n_j}{\hat{b}}$$

Proof.

$$\begin{aligned}\mathbf{E}[\hat{h}(i)] &= n_i + \frac{\sum_{j>k} n_j}{\hat{b}} - \frac{n}{\hat{b}} \\ &= n_i - \frac{\sum_{j\leq k} n_j}{\hat{b}}\end{aligned}$$

While this is not quite what we wanted since the estimator is biased downwards, we shall show later that we can correct for this at the end of the algorithm. However, for the remainder of the analysis, it will be convenient to work with the biased version of the estimator. Note however that since b will generally be fairly large and for low parameter Zipf's, the k most frequent items will carry only a small fraction of the total weight of the stream, this estimator should be quite close to the actual value in practice.

We now compute the variance of the estimator.

Lemma 6. *Conditioned on the event that none of the top k items collide with i, the variance of $\hat{h}(i)$ is bounded by $\frac{\sum_{j>k} n_j^2}{\hat{b}}$.*

Proof.

$$\begin{aligned}\mathbf{Var}[\hat{h}(i)] &= \mathbf{E}[(\hat{h}(i) - n_i + \frac{\sum_{j\leq k} n_j}{\hat{b}})^2] \\ &= \mathbf{E}[(n_i + \sum_{j>k} x_j - \frac{n}{\hat{b}} - n_i + \frac{\sum_{j\leq k} n_j}{\hat{b}})^2] \\ &= \mathbf{E}[(\sum_{j>k} x_j - \frac{\sum_{j>k} n_j}{\hat{b}})^2] \\ &= \mathbf{Var}[\sum_{j>k} x_j] \\ &= \sum_{j>k} \mathbf{Var}[x_j] \\ &= \sum_{j>k} [\frac{1}{\hat{b}}(n_j - \frac{n_j}{\hat{b}})^2 + \frac{\hat{b}-1}{\hat{b}}(0 - \frac{n_j}{\hat{b}})^2] \\ &= \sum_{j>k} [\frac{n_j^2}{\hat{b}} + \frac{n_j^2}{\hat{b}^2} - 2\frac{n_j^2}{\hat{b}^2}] \\ &< \frac{\sum_{j>k} n_j^2}{\hat{b}}\end{aligned}$$

The linearity of the variance is implied by the pairwise independance of the hash function $h(\cdot)$.

So our estimator has the same variance as the Count Sketch estimator. By stepping through their analysis, we can obtain the same space bounds. However, when we analyze the algorithm with higher moments instead of the variants and apply deviation bounds analogous to Chebyshev's Inequality for these higher moments, we can achieve a slight improvement. By contrast, higher moment analysis does not yield better results for the CCF algorithm. In the rest of this section, we will work only with even moments because of the difficulty of working with the absolute values in odd moments.

As an example, consider the Fourth Moment of $\hat{h}(i)$. For ease of presentation, in the rest of this paper we let μ be the expectation of $s(i) \cdot h(i)$ and $\hat{\mu}$ be the expectation of $\hat{h}(i)$. The calculation of the Fourth Moment follows in exactly the same way as the variance calculation. The only change needed is for $\hat{h}(\cdot)$ to be four-wise independent. This gives

$$\mathbf{E}[(\hat{h}(i) - \hat{\mu})^4] < \frac{\sum_{j>k} n_j^4}{\hat{b}}$$

which implies the following bound on b:

$$b = \frac{\sum_{j>k} n_j^4}{(\epsilon n_k)^4}$$

using a Fourth Moment generalization of Chebyshev's Inequality.

In general, for arbitrary even p, our algorithm has a pth moment of

$$\mathbf{E}[(\hat{h}(i) - \hat{\mu})^p] = \frac{\sum_{j>k} n_j^p}{\hat{b}}$$

which implies a bound on b of

$$b = \frac{\sum_{j>k} n_j^p}{(\epsilon n_k)^p}$$

Putting all this together gives us the following theorem:

Theorem 2. *Our algorithm solves* FINDAPPROXTOP(S, k, ϵ) *in space*

$$O(k \log \frac{n}{\delta} + \frac{\sum_{j>k} n_j^p}{(\epsilon n_k)^p} \log \frac{n}{\delta})$$

when $h(\cdot)$ *is chosen to be a* p*-wise independent hash function for some even number* p*.*

4.2 Correcting the Estimator

Recall from Lemma 5 that our estimator $\hat{h}(i)$ is a biased estimator.

$$\mathbf{E}[\hat{h}(i)] = n_i - \frac{\sum_{j\leq k} n_j}{\hat{b}}$$

We can correct the estimator by observing that our algorithm as stated gives us good estimates of the quantity $Y_i = n_i - \frac{\sum_{j\leq k} n_j}{\hat{b}}$ for each of the k most frequent items. Summing these up we get

$$[\sum_{i\leq k} Y_i] = (1-k)\frac{\sum_{j\leq k} n_j}{\hat{b}}$$

and we can approximate this quantity to within $(1+\epsilon)$ precision. k is known so this gives us a good estimate of the sum and we can use this to correct our estimator, $\hat{h}(i)$.

4.3 Analysis for Zipf distributions

We now analyze the improvement the algorithm gives for Zipf distributions. We omit the parameter c in the definition of the Zipf distribution from our calculations since all occurrences of c eventually cancel.

Recall that our bounds from Theorem 2 depend on the sum of the pth power of the n_j's. Accordingly we compute

$$\sum_{j>k} n_j^p = \sum_{j>k} \frac{1}{(j)^{pz}} = \begin{cases} O(m^{1-pz}), & z < \frac{1}{p} \\ O(\log m), & z = \frac{1}{p} \\ O(k^{1-pz}), & z > \frac{1}{p} \end{cases}$$

Plugging these values into Theorem 2, we obtain the following values for $\hat{b}$

$$\hat{b} = \begin{cases} O(\frac{m^{1-pz}k^{pz}}{\epsilon^{pz}})), & z < \frac{1}{p} \\ O(\frac{k\log m}{\epsilon})), & z = \frac{1}{p} \\ O(\frac{k}{\epsilon^{pz}}), & z > \frac{1}{p} \end{cases}$$

The final bounds or obtained by multiplying these values for b by the number of hash tables, $O(\log \frac{n}{\delta})$.

In comparison to our algorithm, the CCF algorithm does not achieve better space bounds using higher moment analysis. We refer the reader to Appendix A for this analysis. The bounds for both algorithms are reported in Table 1.

When reading the Table 1, note that $k < m$ always and for our target applications, $k << m$ since m will generally be extremely large, possibly even unbounded (e.g. the number of distinct files on a group of web servers or the number of distinct queries to a search engine). while k will generally be relatively small (e.g. the size of a cache or the 10 most popular queries on a given day). It can be seen that our bounds are better than those of the CCF algorithm for small values of z and never worse. It was shown in [CCF02] that for Zipf parameter above 1, the trivial sampling algorithm performs better than the CCF algorithm. Likewise, it performs better than our algorithm. However, our focus in this paper is on low parameter Zipf's which are seen in practice in web access traces.

Zipf parameter	CCF Algorithm	Our Algorithm
$z < \frac{1}{p}$	$m^{1-2z}k^{2z}\log\frac{n}{\delta}$	$m^{1-pz}k^{pz}\log\frac{n}{\delta}$
$z = \frac{1}{p}$	$m^{1-2z}k^{2z}\log\frac{n}{\delta}$	$k\log m\log\frac{n}{\delta}$
$\frac{1}{p} < z < \frac{1}{2}$	$m^{1-2z}k^{2z}\log\frac{n}{\delta}$	$k\log\frac{n}{\delta}$
$z = \frac{1}{2}$	$k\log m\log\frac{n}{\delta}$	$k\log\frac{n}{\delta}$
$z > \frac{1}{2}$	$k\log\frac{n}{\delta}$	$k\log\frac{n}{\delta}$

Table 1. Comparison of space requirements for CCF algorithm vs. Our algorithm. Assume $\frac{1}{p} < \frac{1}{2}$.

We note that our algorithm avoids polynomial dependence on m for all but the lowest parameter Zipf distributions ($z < \frac{1}{p}$), and furthermore, we can raise p to an arbitrarily large constant to make this threshold as low as we like.

5 Discussion

We presented a simple, new algorithm for FindApproxTop(S, k, ϵ) that improves on the space bounds of the CCF algorithm when the Zipf parameter is very small. This is a somewhat surprising result since the algorithm is apparently much simpler. In addition, we motivated our work by describing a web caching application for which our version of the frequent items problem is necessary and for which there is evidence that good algorithms for low parameter Zipf distributions are desirable.

Our algorithm is very simple to implement, essentially involving only hashing, median finding and maintaining a heap. p-wise independent hash functions can be trivially implemented by choosing a random degree-p polynomial over an appropriately sized finite field. As p gets large, though, the amount of storage required for the hash functions may come in to play. Therefore, the total per-item processing time is essentially only $O(\log\frac{n}{\delta})$, assuming a linear time median-finding algorithm. It practice, sorting the individual estimators to find the median is presumbly fast enough.

6 Acknowledgements

This work was funded by NSF grant EF 03-31494.

References

[ABCO96] Virgilio Almeida, Azer Bestavros, Mark Crovella and Adriana de Oliveira. Characterizing Reference Locality in the WWW. In *IEEE Internation Conference in Parallel and Distributed Information Systems*, 1996.

[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.

[BCF+99] L. Breslau, P. Cao, L. Fan, G. Phillips and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. IEEE INFOCOM 1999*

[CCF02] Moses Charikar, Kevin Chen and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of ICALP*, 2002.

[CM03] Graham Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *PODS 2003*

[CBC95] Carlos Cunha, Azer Bestavros and Mark Crovella. Characteristics of WWW client-based traces. Tech Report TR-95-010, Boston University, April 1995.

[CD73] Edward Coffman, Jr. and Peter Denning. Operating Systems Theory Prentice-Hall,Inc. 1973

[DLM02] Erik Demaine, A. Lopez-Ortiz and J. I. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In ESA 2002.

[FSG+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.

[GGI+02] Anna Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Fast, small-space algorithms for approximate histogram maintenance. 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.

[G94] Steven Glassman. A Caching Relay for the World Wide Web. In *WWW 1*, 1994.

[KPS03] Richard Karp, Christos Papadimitrious and Scott Shenker. A simple algorithm for finding frequent elements in streams and bags. Unpublished.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate Frequency Counts over Streaming Data. In *Proc. 28th VLDB Conference*, 2002.

[SKW00] D. Serpanos, G. Karakostas, W. Wolf. Effective Caching of Web Objects using Zipf's Law In *Proc. IEEE ICME 2000*

[XO02] Yinglian Xie and David O'Hallaron. Locality for Search Engine Queries and its Implications for Caching. In *Prof. INFOCOM 2002*

A Higher Moment Analysis for CCF

In this Appendix, we show that the CCF estimator does not achieve good bounds for higher moments. Consider again the Fourth Moment and recall that that our estimator had a Fourth Moment

$$\mathbf{E}[(\hat{h}(i) - \hat{\mu})^4] = \frac{\sum_{j>k} n_j^4}{\hat{b}}$$

The CCF estimator, by contrast, has the following Fourth Moment:

$$\mathbf{E}[(s(i) \cdot h(i) - n_i)^4] = \frac{\sum_{j>k} n_j^4 + \sum_{j \neq \ell > k} n_j^2 n_\ell^2}{b}$$

This is because the ± 1 random variables have the unusual property that even powers have expectation 1 while odd powers have expectation 0. This means that in general, the pth moment of the Count Sketch estimator is

$$\mathbf{E}[(s(i) \cdot h(i) - \mu)^p] \geq \frac{\sum_{j>k} n_j^p + \sum_{j \neq \ell, j, \ell > k} n_j^{\frac{p}{2}} n_\ell^{\frac{p}{2}}}{b}$$

Recall the p is assumed to be an even number.

When the distribution is very skewed, the first term dominates. On the other hand, if the distribution is uniform, say, the second term dominates. We will now show that for Zipf distributions, the second term dominates.

$$\begin{aligned}\sum_{j \neq \ell, j, \ell > k} n_j^{\frac{p}{2}} n_\ell^{\frac{p}{2}} = \sum_{j>k} j^{-\frac{pz}{2}} \sum_{\ell > j} \ell^{\frac{pz}{2}} &= \begin{cases} \sum_{j>k} j^{-\frac{pz}{2}} m^{1-\frac{pz}{2}}, \ z < \frac{2}{p} \\ \sum_{j>k} j^{-\frac{pz}{2}} \log m, \ \ z = \frac{2}{p} \\ \sum_{j>k} j^{-\frac{pz}{2}} j^{1-\frac{pz}{2}}, \ \ z > \frac{2}{p} \end{cases} \\ &= \begin{cases} m^{1-pz}, & z < \frac{2}{p} \\ \log^2 m, & z = \frac{2}{p} \\ \sum_{j>k} j^{1-pz}, & z > \frac{2}{p} \end{cases} \\ &= \begin{cases} m^{1-pz}, \ z < \frac{2}{p} \\ \log^2 m, \ z = \frac{2}{p} \\ k^{2-pz}, \ \ z > \frac{2}{p} \end{cases}\end{aligned}$$

Comparing these bounds to those in Table 1, we can see that they do no better than the bounds achieved by using only the Second Moment Method on the CCF algorithm.