Identifying Frequent Items in Sliding Windows over On-Line Packet Streams*

(Extended Abstract)

Lukasz Golab
School of Comp. Sci.
University of Waterloo
lgolab@uwaterloo.ca

David DeHaan
School of Comp. Sci.
University of Waterloo
dedehaan@uwaterloo.ca

Erik D. Demaine
Lab. for Comp. Sci.
M.I.T.
edemaine@mit.edu

Alejandro López-Ortiz
School of Comp. Sci.
University of Waterloo
alopez-o@uwaterloo.ca

J. Ian Munro
School of Comp. Sci.
University of Waterloo
imunro@uwaterloo.ca

ABSTRACT

Internet traffic patterns are believed to obey the power law, implying that most of the bandwidth is consumed by a small set of heavy users. Hence, queries that return a list of frequently occurring items are important in the analysis of real-time Internet packet streams. While several results exist for computing frequent item queries using limited memory in the infinite stream model, in this paper we consider the limited-memory sliding window model. This model maintains the last N items that have arrived at any given time and forbids the storage of the entire window in memory. We present a deterministic algorithm for identifying frequent items in sliding windows defined over real-time packet streams. The algorithm uses limited memory, requires constant processing time per packet (amortized), makes only one pass over the data, and is shown to work well when tested on TCP traffic logs.

Categories and Subject Descriptors

C.2.3 [**Communication Networks**]: Network Operations—*Network monitoring*

General Terms

Algorithms

*This research is partially supported by the Natural Sciences and Engineering Research Council of Canada, and by the Nippon Telegraph and Telephone Corporation.

IMC'03, October 27–29, 2003, Miami Beach, Florida, USA.

Keywords

Internet traffic monitoring, on-line stream analysis, sliding windows, frequent item queries

1. INTRODUCTION

On-line data streams possess interesting computational characteristics, such as unknown or unbounded length, possibly very fast arrival rate, inability to backtrack over previously arrived items (only one sequential pass over the data is permitted), and a lack of system control over the order in which the data arrive [10]. Real-time analysis of network traffic has been one of the primary applications of data stream management systems; examples include Gigascope [4], STREAM [1], and Tribeca [15]. A particular problem of interest—motivated by traffic engineering, routing system analysis, customer billing, and detection of anomalies such as denial-of-service attacks— concerns statistical analysis of data streams with a focus on newly arrived data and frequently appearing packet types. For instance, an ISP may be interested in monitoring streams of IP packets originating from its clients and identifying the users who consume the most bandwidth during a given time interval; see [6, 7] for additional motivating examples. These types of queries, in which the objective is to return a list of the most frequent items (called *top-k queries* or *hot list queries*) or items that occur above a given frequency (called *threshold queries*), are generally known as *frequent item queries*. However, to make such analysis meaningful, bandwidth usage statistics should be kept for only a limited amount of time—for example, one hour or a single billing period—before being replaced with new measurements. Failure to remove stale data leads to statistics aggregated over the entire lifetime of the stream, which are unsuitable for identifying recent usage trends.

A solution for removing stale data is to periodically reset all statistics. This gives rise to the *landmark window model*, in which a time point (called the landmark) is chosen and statistics are only kept for that part of a stream which falls between the landmark and the current time. A major disadvantage of this model is that the size of the window varies—

the window begins with size zero and grows until the next occurrence of the landmark, at which point it is reset to size zero. In contrast, the *sliding window model* expires old items as new items arrive. Two common types of sliding windows are *count-based* windows, which maintain the last N packets seen at all times and *time-based* windows, which include only those items which have arrived in the last t time units.

If the entire window fits in main memory, answering threshold queries over sliding windows is simple: we maintain frequency counts of each distinct item in the window and update the counters as new items arrive and old items expire. Unfortunately, Internet traffic on a high-speed link arrives so fast that useful sliding windows may be too large to fit in main memory (and the system cannot keep up with the stream if the window is stored on disk). In this case, the window must somehow be summarized and an answer must be approximated on the basis of the available summary information. One solution, initially proposed by Zhu and Shasha in [16] and also used in this work, is to divide the sliding window into sub-windows, only store a summary of each sub-window, and re-evaluate the query when the most recent sub-window is full. This reduces space usage, but induces a "jumping window" instead of a gradually sliding window, with the jump size equal to the sub-window size.

1.1 Our Contributions

We are interested in identifying frequent items (occurring with a frequency that exceeds a given threshold) in sliding windows over on-line data streams and estimating their true frequencies, while using as little space as possible and making only one pass over the data. We present a simple deterministic algorithm, FREQUENT, that identifies frequently occurring items in sliding windows and estimates their frequencies. Algorithm FREQUENT requires constant processing time per packet (amortized) and is shown to work well when tested on TCP connections logs.

1.2 Roadmap

The remainder of this paper is organized as follows: Section 2 presents relevant previous work, Section 3 introduces algorithm FREQUENT, Section 4 contains experimental results, and Section 5 concludes the paper with suggestions for future work.

2. PREVIOUS WORK

2.1 Frequent Item Algorithms for Infinite Streams

Frequent item algorithms in the infinite stream model employ sampling, counting, and/or hashing to generate approximate answers using limited space. The main difficulty lies in finding a small set of potentially frequent items to monitor and detecting unpopular items that suddenly become frequent. In this context, approximation may mean a number of things: an algorithm may either return all of the frequent item types (and some false positives), some frequent item types (and some false negatives), identities of the frequent items but no frequency counts, or identities and approximate counts of the frequent items. Note that the terms *packet types*, *item types*, and *item categories* are used interchangeably throughout the paper.

A naive counting method for answering threshold queries examines all items as they arrive and maintains a counter for each item type. This method takes $\Omega(n)$ space, where n is the number of packets seen so far—consider a stream with $n-1$ unique packet types and one of the types occurring twice. Random sampling reduces space usage, but may result in a large approximation error, especially in the presence of bursty TCP/IP traffic. Three hybrid counting-sampling algorithms have been proposed to address this trade-off. Estan and Varghese give an algorithm in [7] that uses sampling only to select whether an item is to be examined more thoroughly; once an item is selected, all of its occurrences are counted (this idea also appears in Gibbons and Matias [9]). Manku and Motwani give a similar algorithm that also decreases the sampling rate with time in order to bound memory usage [12]. The algorithm in Demaine et al. [6] finds items occurring above a relative frequency of $1/\sqrt{nm}$ with high probability, where n is the number of incoming items observed and m is the number of available counters. This algorithm divides the stream into a collection of rounds, and for each round counts the occurrences of $m/2$ randomly sampled categories. At the end of each round, the $m/2$ winners from the current round are compared with $m/2$ winners stored from previous rounds and if the count for any current winner is larger than the count for a stored category (from any of the previous rounds), the stored list is updated accordingly.

Demaine et al. and Manku and Motwani also give counting-only frequent item algorithms. The former uses only m counters and deterministically identifies all categories having a relative frequency above $1/(m+1)$, but may return false positives and therefore requires a re-scan of the data (forbidden in the on-line stream model) to determine the exact set of frequent items. The latter maintains a counter for each distinct item seen, but periodically deletes counters whose average frequencies since counter creation time fall below a fixed threshold. To ensure that frequent items are not missed by repeatedly deleting and re-starting counters, each frequency estimate includes an error term that bounds the number of times that the particular item could have occurred up to now.

Fang et al. present various hash-based frequent item algorithms in [8], but each requires at least two passes over the data. The one-pass sampled counting algorithm by Estan and Varghese may be augmented with hashing as follows. Instead of sampling to decide whether to keep a counter for an item type, we simultaneously hash each item's key to d hash tables and add a new counter only if all d buckets to which a particular element hashes are above some threshold (and if the element does not already have a counter). This reduces the number of unnecessary counters that keep track of infrequent packet types. A similar technique is used by Charikar et al. in [2] in conjunction with hash functions that map each key to the set $\{-1, 1\}$. Finally, Cormode and Muthukrishnan give a randomized algorithm for finding frequent items in a continually changing database (via arbitrary insertions and deletions) using hashing and grouping of items into subsets [3].

2.2 Sliding Window Algorithms

Many infinite stream algorithms do not have obvious counterparts in the sliding window model. For example, one counter suffices to maintain the minimum element in an infinite stream, but keeping track of the minimum element in a sliding window of size N takes $\Omega(N)$ space—consider an

increasing sequence of values, in which the oldest item in any window is the minimum and must be replaced whenever the window moves forward. The fundamental problem is that as new items arrive, old items must be simultaneously evicted from the window, meaning that we need to store some information about the order of the packets in the window.

Zhu and Shasha introduce *Basic Windows* to incrementally compute simple windowed aggregates in [16]. The window is divided into equally-sized Basic Windows and only a synopsis and a timestamp are stored for each Basic Window. When the timestamp of the oldest Basic Window expires, that window is dropped and a fresh Basic Window is added. This method does not require the storage of the entire sliding window, but results are refreshed only after the stream fills the current Basic Window. If the available memory is small, then the number of synopses that may be stored is small and hence the refresh interval is large.

Exponential Histograms (EH) have been introduced by Datar et al. [5] and recently expanded in [14] to provide approximate answers to simple window aggregates at all times. The idea is to build Basic Windows with various sizes and maintain a bound on the error caused by counting those elements in the oldest Basic Window which may have already expired. The algorithm guarantees an error of at most ϵ while using $O(\frac{1}{\epsilon}\log^2 N)$ space.

3. MOTIVATION AND ALGORITHM FOR FINDING FREQUENT ITEMS IN SLIDING WINDOWS

3.1 Motivation

The frequent item algorithms for infinite streams may be extended to the sliding window model using a Basic Window strategy. The counters used in the counting methods could be split and a timestamp assigned to each sub-counter; this essentially reduces to the Basic Window method with item counts stored in the synopses. Similarly, hash tables could be split in the same way, resulting in a Basic Window approach with hash tables stored in the synopses. Unfortunately, both the Basic Window approach and Exponential Histograms do not directly apply to the frequent item problem in sliding windows. Fundamentally, this is because these techniques are suitable for *distributive* and *algebraic* aggregates only [11]. That is, the aggregate must be computable either by partially pre-aggregating each Basic Window and combining the partial results to return the final answer (e.g. the windowed sum can be computed by adding up sums of the Basic Windows), or by storing some other constant-size Basic Window synopses that can be merged to obtain the final answer (e.g. the windowed average can be computed by storing partial sums and item counts in each Basic Window and dividing the cumulative sum by the cumulative count). However, frequent item queries are classified as *holistic* aggregates, which require synopses whose sizes are proportional to the sizes of the Basic Windows.

Answering frequent item queries using small-size Basic Window synopses is difficult because there is no obvious rule for merging the partial information in order to obtain the final answer. For instance, if each Basic Window stores counts of its top k categories, we cannot say that any item appearing in any of the top-k synopses is one of the k most frequent types in the sliding window—a bursty packet type that dominates one Basic Window may not appear in any other Basic Windows at all. We also cannot say that any frequent item must have appeared in at least one top-k synopsis—if k is small, say $k = 3$, we would ignore a frequent item type that consistently ranks fourth in each Basic Window and therefore never appears on any of the top-k synopses. Fortunately, we will show empirically that these problems are far less serious if the sliding window conforms to a power-law-like distribution, in which case we expect several very frequent categories (e.g. popular source IP addresses or protocol types) that will be repeatedly be included in nearly every top-k synopsis.

3.2 Algorithm

We propose the following simple algorithm, FREQUENT, that employs the Basic Window approach (i.e. the jumping window model) and stores a top-k synopsis in each Basic Window. We fix an integer k and for each Basic Window, maintain a list of the k most frequent items in this window. We assume that a single Basic Window fits in main memory, within which we may count item frequencies exactly. Let δ_i be the frequency of the kth most frequent item in the ith Basic Window. Then $\delta = \sum_i \delta_i$ is the upper limit on the frequency of an item type that does not appear on any of the top-k lists. Now, we sum the reported frequencies for each item present in at least one top-k synopsis and if there exists a category whose reported frequency exceeds δ, we are certain that this category has a true frequency of at least δ. The pseudocode is given below, assuming that N is the sliding window size, b is the number of elements per Basic Window, and N/b is the total number of Basic Windows. An updated answer is generated whenever the window slides forward by b packets.

Algorithm FREQUENT
Repeat:

1. For each element e in the next b elements:
   If a local counter exists for the type of element e:
   Increment the local counter.
   Otherwise:
   Create a new local counter for this element type and set it equal to 1.
2. Add a summary S containing identities and counts of the k most frequent items to the back of queue Q.
3. Delete all local counters.
4. For each type named in S:
   If a global counter exists for this type:
   Add to it the count recorded in S.
   Otherwise:
   Create a new global counter for this element type and set it equal to the count recorded in S.
5. Add the count of the kth largest type in S to δ.
6. If $sizeOf(Q) > N/b$:
   (a) Remove the summary S' from the front of Q and subtract the count of the kth largest type in S' from δ.
   (b) For all element types named in S':
   Subtract from their global counters the counts recorded in S'.
   If a counter is decremented to zero:
   Delete it.
   (c) Output the identity and value of each global counter $> \delta$.

3.3 Analysis

Note that as shown above, algorithm FREQUENT assumes that all Basic Windows have the same number of items, as is usually the case in count-based windows. This assumption, however, is not necessary to ensure the algorithm's correctness—we may replace line 6 with another condition for emptying the queue, say every t time units. Therefore, algorithm FREQUENT may be used with time-based windows (that may have uniformly sized Basic Windows in terms of time, but not necessarily in terms of tuple count) without any modifications. In the remainder of this section, we will maintain the assumption of equal item counts in Basic Windows to simplify the analysis.

Algorithm FREQUENT accepts three parameters: N, b, and k[1]. The choice of b is governed by the latency requirements of the application: choosing a small value of b increases the frequency with which new results are generated. However, the amount of available memory dictates the maximum number of Basic Windows and the synopsis size k.

The space requirement of algorithm FREQUENT consists of two parts: the working space needed to create a summary for the current Basic Window and the storage space needed for the top-k synopses. Let d be the number of distinct item types in a Basic Window (the value of d may be different for each Basic Window, but we ignore this point to simplify the analysis) and D be the number of distinct values in the sliding window. In the worst case, the working space requires d local counters of size $\log b$. For storage, there are N/b summaries, each requiring k counters of size at most $\log b$. There are also at most kN/b global counters of size at most $\log N$. This gives a total worst-case space bound of $O(d \log b + \frac{kN}{b}(\log b + \log N))$. The time complexity of algorithm FREQUENT is $O(\min(k, b) + b)$ for each pass through the outer loop. Since each pass consumes b arriving elements, this gives $O(1)$ amortized time per element.

Algorithm FREQUENT may return false negatives. Consider an item that appears on only a few top-k lists, but summing up its frequency from these top-k lists does not exceed δ—however, this item may be sufficiently frequent in other Basic Windows (but not frequent enough to register on the top-k lists of these other windows) that its true frequency count exceeds δ. The obvious solution for reducing the number of false negatives is to increase k, but this also increases space usage. Alternatively, decreasing b increases the number of Basic Windows, which may also help eliminate false negatives.

Another possible downside of algorithm FREQUENT is that if k is small, then δ may be very large and the algorithm will not report any frequent flows. On the other hand, if k is large and each synopsis contains items of a different type (i.e. there are very few repeated top-k "winners"), the algorithm may require a great deal of storage space, perhaps as much as the size of the sliding window. Notably, when b is only slightly larger than k, there may be fewer than k distinct items in any Basic Window. In this case, algorithm FREQUENT will track the exact frequencies of most (if not all) of the distinct packet types, essentially producing a compressed representation of the sliding window (or more precisely, the jumping window) that stores item frequencies in each Basic Window.

[1]Note that N and b are specified in units of time in time-based windows

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

We have tested algorithm FREQUENT on count-based windows over TCP traffic data obtained from the Internet Traffic Archive (`ita.ee.lbl.gov`). We used a trace that contains all wide-area TCP connections between the Lawrence Berkeley Laboratory and the rest of the world between September 16, 1993 and October 15, 1993 [13]. The trace contains 1647 distinct source IP addresses, which we treat as distinct item types. We set $N = 100000$ and experiment with three values of b: $b = 20$ (5000 Basic Windows in total), $b = 100$ (1000 Basic Windows in total), and $b = 500$ (200 Basic Windows in total). The size of the synopses, k, is varied from one to ten. In each experiment, we randomly choose one hundred starting points for sliding windows within the trace and execute our algorithm over those windows. We also run a brute-force algorithm to calculate the true item type frequencies. We have measured the average threshold δ, the average number of over-threshold flows reported, accuracy, and space usage over one hundred trials, as shown in Figure 1.

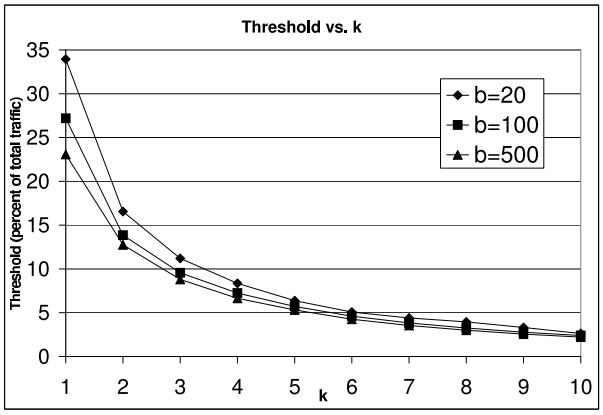
4.2 Accuracy

Recall that algorithm FREQUENT identifies a category as being over the threshold δ if this category's frequency count recorded in the top-k synopses exceeds δ. As k increases, the frequency of the kth most frequent item decreases and the overall threshold δ decreases, as seen in Figure 1 (a). Furthermore, increasing the number of synopses by decreasing b increases δ as smaller Basic Windows capture burstiness on a finer scale. Consequently, as k increases, the number of packet types that exceed the threshold increases, as seen in Figures 1 (b) and (c). The former plots the number of over-threshold IP addresses, while the latter shows the number of IP addresses that were identified by our algorithm as being over the threshold. For example, when $k = 5$, the threshold frequency is roughly five percent (Figure 1 (a)) and there are between three and four source IP addresses whose frequencies exceed this threshold (Figure 1 (b)).

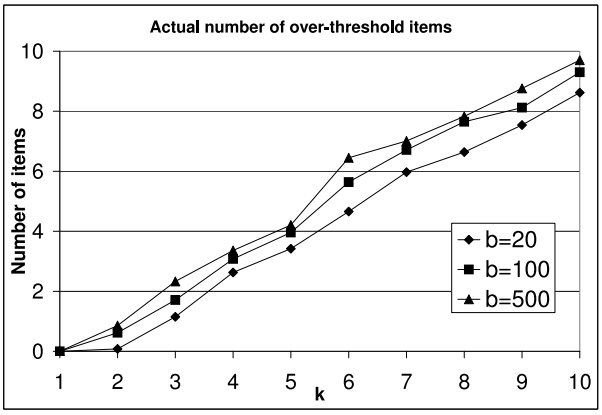
It can be seen in Figures 1 (b) and (c) that algorithm FREQUENT does not identify all the packet types that exceed the threshold (there may be false negatives, but recall that there are never any false positives). In Figure 1 (d), we show the percentage of over-threshold IP addresses that were identified by algorithm FREQUENT. The general trend is that for $k \geq 3$, at least 80% of the over-threshold IP addresses are identified. Increasing the number of Basic Windows (i.e. decreasing the Basic Window size b) also improves the chances of identifying all of the above-threshold packet types. For instance, if $k > 7$ and $b = 20$, false negatives occur very rarely.

4.3 Space Usage

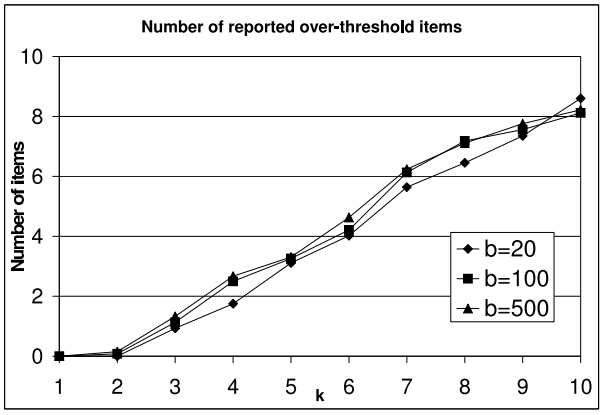
Figure 1 (e) shows the space usage of algorithm FREQUENT in terms of the number of attribute-value, frequency-count pairs that need to be stored. Recall that the sliding window size in our experiments is 100000, which may be considered as a rough estimate for the space usage of a naive technique that stores the entire window. The space usage of our algorithm is significantly smaller, especially when b is large and/or k is small. Because a top-k synopsis must be stored for each Basic Window, the number of Basic Windows has the greatest effect on the space requirements.



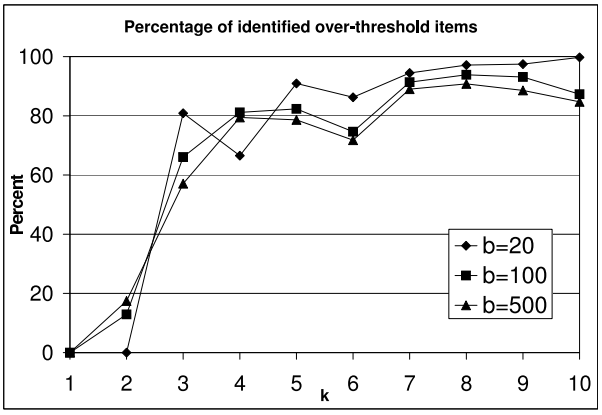
(a)



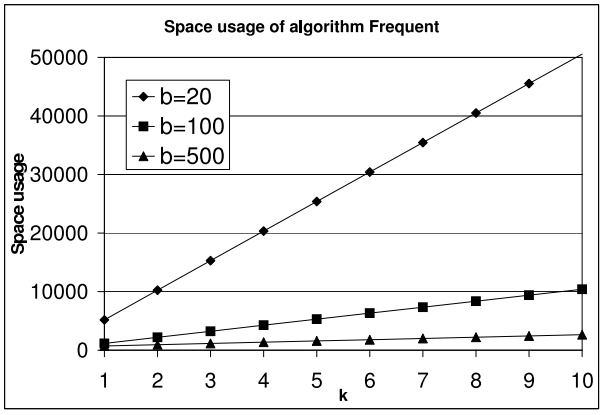
(b)



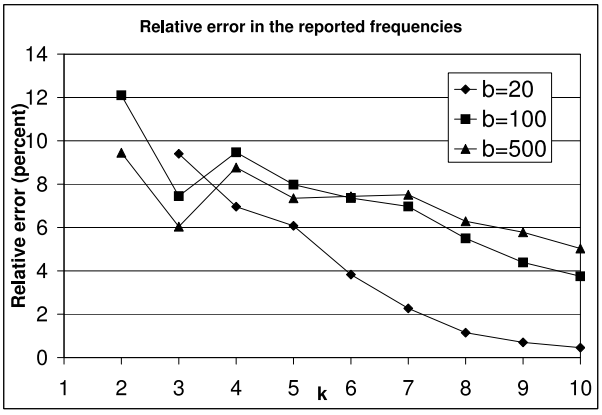
(c)



(d)



(e)



(f)

Figure 1: Analysis of frequent-item reporting capabilities, accuracy, and space usage of algorithm Frequent. Part (a) shows the average value of the threshold δ as a function of k, part (b) shows the number of packet types whose frequencies exceed the threshold as a function of k, and part (c) graphs the number of packet types reported by our algorithm as exceeding the threshold as a function of k. Furthermore, part (d) shows the percentage of over-threshold packets that were identified as a function of k, part (e) plots the space usage as a function of k, and part (f) shows the relative error in the frequency estimates of over-threshold items returned by our algorithm.

4.4 Precision

Recall from Figure 1 (d) that algorithm FREQUENT may report false negatives. We have discovered that unreported frequent types typically have frequencies that only slightly exceed the threshold, meaning that the most frequent types are always reported. Furthermore, the reported frequency estimates were in many cases very close to the actual frequencies, meaning that the reported frequent IP addresses were arranged in the correct order (though item types with similar frequencies were often ordered incorrectly). To quantify this statement, we have plotted in Figure 1 (f) the average relative error (i.e. the difference between the measured frequency and the actual frequency divided by the actual frequency) in the frequency estimation of the over-threshold IP addresses for ten values of k and three values of b. The relative error decreases as k increases and as b decreases. For example, when $b = 20$ and $k \geq 7$, the average relative error is below two percent. Therefore, the reported IP addresses are nearly always ordered correctly, unless there are two IP addresses with frequencies within two percent of each other, and only those IP addresses which exceed the threshold by less than two percent may remain unreported.

4.5 Lessons Learned

Algorithm FREQUENT works well as an identifier of frequent items and, to some extent, their approximate frequencies, when used on Internet traffic streams. As expected, increasing the size of the top-k synopses increases the number of frequent flows reported, decreases the number of false negatives, and improves the accuracy of the frequency estimates. Increasing the number of Basic Windows reduces the refresh delay, decreases the proportion of false negatives and increases the accuracy of the frequency estimates. However, space usage grows significantly when either k increases or b decreases.

5. CONCLUSIONS

We presented an algorithm for detecting frequent items in sliding windows defined over packet streams. Our algorithm uses limited memory (less than the size of the window) and works in the jumping window model. It performs well with bursty TCP/IP streams containing a small set of popular item types.

Future work includes theoretical analysis of algorithm FREQUENT in order to provide bounds on the probability of false negatives and the relative error in frequency estimation, given a fixed amount of memory and the allowed answer reporting latency. For instance, if the underlying data conform to a power law distribution, we suspect a correlation between k (the size of the synopses required to guarantee some error bound) and the power law coefficient. Another possible improvement concerns translating our results to the gradually sliding window model, where query results are refreshed upon arrival of each new packet. This may be done either by bounding the error in our algorithm due to under-counting the newest Basic Window and over-counting the oldest Basic Window that has partially expired, or perhaps by exploiting the Exponential Histogram approach and its recent extensions in order to extract frequently occurring values. Finally, this work may also be considered as a first step towards solving the more general problem of reconstructing a probability distribution of a random variable given only an indication of its extreme-case behaviour.

6. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data streams. *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 1–16, 2002.

[2] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Proc. 29th Int. Colloquium on Automata, Languages and Programming*, pages 693–703, 2002.

[3] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 296–306, 2003.

[4] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an sql interface. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 623, 2002.

[5] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *Proc. 13th SIAM-ACM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[6] E. Demaine, A. Lopez-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. *Proc. European Symposium on Algorithms*, pages 348–360, 2002.

[7] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *Proc. ACM SIGCOMM Internet Measurement Workshop*, pages 75–80, 2001.

[8] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. *Proc. 24th Int. Conf. on Very Large Data Bases*, pages 299–310, 1998.

[9] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 331–342, 1998.

[10] L. Golab and M. T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, Jun. 2003.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. *Proc. 12th Int. Conf. on Data Engineering*, pages 152–159, 1996.

[12] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 346–357, 2002.

[13] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, Jun. 1995.

[14] L. Qiao, D. Agrawal, and A. El Abbadi. Supporting sliding window queries for continuous data streams. *Proc. 15th Int. Conf. on Scientific and Statistical Database Management*, 2003, to appear.

[15] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. *Proc. USENIX Annual Technical Conf.*, 1998.

[16] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 358–369, 2002.