

Memory Efficient Algorithm for Mining Recent Frequent Items in a Stream ^{*}

Piotr Kolaczkowski

Warsaw University of Technology, Institute of Computer Science
P.Kolaczkowski@ii.pw.edu.pl

Abstract. In the paper we present an improved version of multistage hashing based algorithm, used to find frequent items in a stream. Our algorithm uses low-pass filters instead of simple counters, so it concentrates more on recent items and ignores the old ones. Such behaviour is similar to sliding window based algorithms, but requires less memory and is suitable for real-time applications. The algorithm continuously gives estimates of frequencies of the most frequent items. It was tested with streams having various frequency distributions and proved to work correctly.

1 Introduction

Identifying frequent items in a stream is an important task in various fields of computing, e.g. network traffic, database workload or search engine workload analysis. The problem is complex, because it is usually not possible to store even a small fraction of the data from the stream in the memory for later offline analysis. There is usually too much data per a unit of time. It is also difficult or even not possible to attach a counter to each item category. For example if we had to analyze network flows described as a pair of IPv4 addresses each, we would have 2^{64} potential flows to monitor. Even though we would never see most of them, the number of those seen would still reach several hundred thousands [1].

Fortunately, the number of interesting frequent item categories is usually relatively small compared to the number of all item categories. They are called *heavy-hitters*. In most applications, only the heavy-hitters are taken into consideration. For instance, a database administrator would like to know the queries having the largest impact on the system's load, i.e. the most frequent and the longest ones. These queries should be optimized first and, if further optimization is not possible, then probably they are good candidate for caching. The same applies to tuning a web query engine. In network traffic monitoring it is also required to know the largest packet flows in order to prevent *denial of service attacks* (DOS).

^{*} Research has been supported by grant No 3 T11C 002 29 received from Polish Ministry of Education and Science

Apart from that we observe a need for not only finding frequent items in the *whole* stream, but just in the *most recent part* of it. This is due to the fact, that frequencies of the items may significantly change over time. This is the case of a sudden DoS attack or when a new version of software using a database is installed. In the abovementioned applications we are usually more interested in the statistics from the last 10 minutes than from before 3 months.

2 Problem Definition

Consider a stream of items where each item belongs to one of N categories. The N can usually range from hundreds to several millions. Each item category can appear more than once. The number of items is unlimited and not known in advance. Each item can be read only once – it is not possible to rewind or restart the stream. The algorithm must answer queries about the k most frequent categories that were recently seen. It should also estimate the frequencies of these items. The queries can be submitted and must be immediately answered at any time. By the recently seen items we understand the items that appeared within a time window beginning at $t_q - \tau_w$ and ending at t_q , where t_q is the time of submission of the query and τ_w is a constant window size.

An ideal algorithm would give all and only k categories with their corresponding frequencies. If some frequent item is missing from the result set, it is called a *false negative*. If some non-frequent item is reported, it is called a *false positive*. Usually it is sufficient to have an algorithm that reports no false negatives and only a few false positives. False positives can be detected and removed, but this usually requires additional memory.

3 Prior Work

The problem of finding frequent items in a stream of n items has been studied for over past two decades. The earliest algorithm [2] guaranteed to find the item that occurred more often than for half of the time. This was further generalized by Misra and Gries to k items with frequencies higher than $1/(k + 1)$ by using k counters [3]. These algorithms required additional second pass to estimate the exact frequencies of found items and to prune the infrequent items that could be also found in the output. This second pass could be omitted in case false positives were tolerable and no estimation of item frequencies were required. The time complexity of the Misra-Gries algorithm was further improved by usage of more sophisticated data structures [4], but the other properties of the algorithm remained the same.

Manku and Motwani presented a one-pass algorithm [5] that give guarantees on the minimal frequencies of found items while still having low space requirements $O(k \log(n/k))$. Their algorithm uses similar approach to that of Misra-Gries – it associates a counter with each observed item and prune the counter list according to some special algorithm. Similar results have been also achieved by sketch-based algorithm proposed by Charikar et al. [6]. Their algorithm doesn't

store the individual counters, but uses a complex data structure that can estimate frequencies of the infrequent items. The infrequent items are added to the frequent items set whenever their estimated frequency exceeds some threshold.

Estan et al. [7] proposed a hash-based algorithm called a *multistage filter*. The basic idea is that the frequency counters are associated with item hashes, not the items themselves. Many items may hash to the same value, thus having a common counter. By using multiple, independent hashing functions, the algorithm assures the risk of false positives is very low, especially for real-world frequency distributions.

None of the abovementioned algorithms directly addresses the problem of *data aging*, that is the fact that the older items are not as important as the recent items. The data aging can be handled by periodically resetting the counters of a frequent item mining algorithm, so that it "forgets" the old aggregated data. This unfortunately disallows continuous monitoring, as the results should be retrieved only just before the counters are reset. Besides, the same algorithm applied to two identical streams, but having non-zero time offset between each other can produce different frequent item sets. The solution to this problem is using a sliding-window based algorithms like the one in [8]. However, for large windows, their high memory requirements are a huge disadvantage.

The problem of dynamic tracking of frequent items in streams has been recently studied by Cormode and Muthukrishnan [9]. Their work focuses on handling database inserts and deletes, while our approach addresses more general problem of changing frequency distribution.

4 Algorithm

Our algorithm uses low-pass filters to measure frequencies of events and a sketch based method, similar to that presented in [7], to filter out the infrequent items within a limited memory.

4.1 Low-pass Filters

A low-pass filter can be used to estimate a frequency of some events. This simple idea has been borrowed from the signal theory. For each item in the stream, the filter is given a Dirac-delta-shaped impulse on the input. It can be shown that the value of the output signal of such filter is nearly proportional to the frequency of the input signal, if only the signal frequency is high enough [10]. The time constant τ of the filter controls how fast can the output signal value follow the input frequency and how accurate is the estimation. The greater the τ is, the more time is required for the filter to react to frequency changes, but the more accurate the measurement of lower frequencies is. The filter state consists of a single precision counter c that stores the output signal value and a variable t_i storing the time, when the filter's state was last updated. Whenever the input impulse of value v comes to the filter, the state is updated according to the equation:

$$c' = ce^{\frac{t_i - t}{\tau}} + v, \quad (1)$$

where t is the current time. If only the frequency of events is measured, v should always be set to 1. Setting v to e.g. the size of a network packet would cause the filter to estimate the size of a data flow, and setting it to the duration of a database query execution would estimate the average database load. In the further part of the paper we are always referring to the frequency measurement but all the described methods are equally valid for the data flow or workload estimation.

The output value can be calculated at any time from the equation:

$$c' = ce^{\frac{t_i - t}{\tau}} \quad (2)$$

Such a filter has different properties than a simple event counter. A simple event counter can be used to measure the frequency of events in some period of time, but the exact result is known only at the end of the period. If a continuous monitoring is required, then the counter must be periodically restarted. The state of the counter reflects only the number of events seen since the time the counter was restarted, so the frequency cannot be estimated with the same accuracy each time. On the contrary, the low-pass filter can estimate average event frequency at any time with the same accuracy. This enables to trace frequency changes with higher time resolution, so that new frequent events can be discovered much earlier. This is illustrated in the figure 1. Note that the waveform of the simple counter is much different after the occurrence of the second impulse than the first one, while both the sliding window (moving average) and the low-pass filter give stable results. Both impulses are of same value and length. The restart period of the counter, the size of the sliding window and the time constant of the filter were set to 1.0 in this experiment.

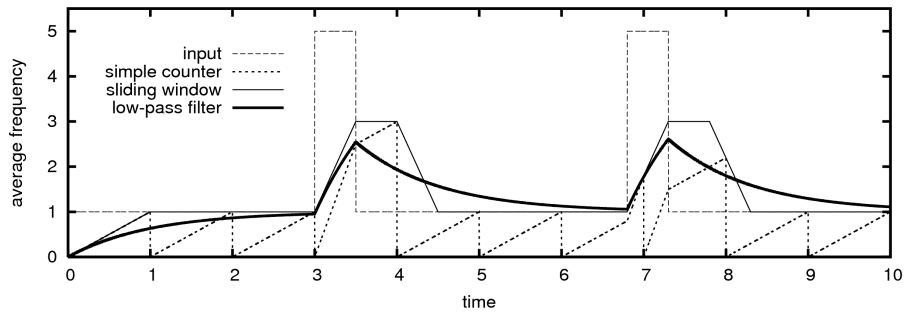


Fig. 1: Various techniques of average frequency measurement

The low-pass filter does not solve the stated problem exactly. The "time-window" is somehow "fuzzy". The most recent items are indeed more important than the previous ones, but the shape of the window is not rectangular but exponential. However, as seen in the figure 1, this would not be a large problem

in practical applications. The results obtained by using a rectangular and an exponential windows are quite similar.

4.2 What Is Frequent and What Is Not

The simplest idea would be to associate an exactly one low-pass filter with a one item category and choose only the k categories with the highest frequency. Unfortunately this cannot be done, because the number of the categories N may be huge and the frequency meters would take up too much memory. Thus only the most frequent categories are associated with filters, one category with one filter. The only problem is how to tell which categories should be in this set. There must be some rules for adding new frequent categories to this set and removing the infrequent ones. Somehow the frequencies of the infrequent items should be estimated, too. Note that this estimation need not to be very accurate, as we only want to decide if the item is worth being in the frequent set or not. This estimation is done thanks to filter sharing implemented by hashing.

Consider m filters numbered from 1 to m . Each item category is hashed to a one filter from this set. We will call this data structure a *sketch*. Because $M \ll N$, more categories can hash to the same filter. Each filter in the sketch will measure a sum of the frequencies of categories that are hashed to it. By the *frequency threshold* f_{thr} we will understand the frequency above which a category is considered frequent and by the *maximum frequency level* f_{max} we will understand the frequency that could be reached by the category if no other categories were present in the stream. In certain applications f_{max} can denote a network connection capacity or a maximum system throughput. We will call a filter that reports frequency above the f_{thr} a *hot filter*. A *cold filter* is any filter that is not a hot filter. If $M \geq k$ and $f_{\text{thr}} \geq f_{\text{max}}/k$, only k reported frequencies will have a chance to reach the high frequency level f_{thr} , for any frequency distribution. Event frequency distributions usually follow a Zipf-like distribution in real world applications. In these cases the number of categories exceeding f_{thr} will be even smaller or the f_{thr} can be set much lower.

The categories that hash to a cold filter cannot be frequent, so they are not added to the frequent set. If they were frequent, they would make the filter hot after some time. Thus false negatives are not possible. Unfortunately false positives are possible because still an infrequent event can hash accidentally to a hot filter and pass to the frequent set. To lower the risk of false positives, the number of cold filters should be high. This can be achieved by setting the total number of filters in the sketch to a value far greater than k . The only problem is, that this solution does not scale well with the increasing number of categories - the m must be proportional to the number of categories N .

The scaling problem has been solved by the parallel multistage filtering, an idea introduced in [7]. Note that the word *filter* in that article does not have anything to do with our *low-pass filters*. We will call it *multistage hashing* to avoid the name clash. Instead of hashing only once and updating the state of only one filter in the sketch, the hashing process is performed in M stages by using M independent hashing functions. States of M filters in the sketch are

updated for each item in the stream. The event is considered to be frequent only if all frequencies reported by each of the M filters are greater than f_{thr} . This significantly lowers the probability for an infrequent event to be classified as a frequent one. If the probability of event hitting a hot filter in each stage is p , then the probability of hitting a hot filter M times and passing to the frequent set is p^M . Of course the size of the sketch must be now M times higher to make this probability remain on the same level as in the one-stage version. Using m only one or two orders of magnitude higher than $s * k$ and having only a few stages can yield very low false positives rates. A thorough theoretical analysis of multistage filters can be found in the original paper [7].

4.3 Managing the Frequent Set

Each entry in the frequent set consists of an item category and its corresponding low-pass filter. The low-pass filter is updated for each occurrence of the item of that category in the input stream. A category is added to the frequent set if any item belonging to that category hashes to hot filters at all stages. A category is removed from the frequent set if its filter becomes cold and so does at least one of the filters being hashed to in the sketch. Otherwise removing such category would be followed by an immediate readding. Because it would be too expensive to update all sketch filters in each iteration, the process of purging the infrequent categories is executed only before adding a new frequent category that would make the frequent set larger than the maximum past size of the frequent set or whenever the user asks for the results.

4.4 Algorithm Improvements

Estan et al. proposed many improvements to the original algorithm. Some of them can be also applied to our modified version of the algorithm.

Serial Filtering In serial filtering the event passes to the next stage only if the values of counters at all previous stages were high enough. The event is blocked by the first low counter and the remaining counters are not checked and not modified. This keeps more counters in a low state (cold). Serial filtering has one disadvantage that makes it less usable for continuous measurements than the parallel filtering: it requires to see M times more items of the same category to detect a new frequent item. This is due to the fact that all the counters/filters in all stages must achieve enough high values and they must do it serially.

Preserving Entries Preserving entries is a technique introduced to improve the accuracy of measurement of frequent items (called large flows in the original paper). Because the original algorithm uses simple, periodically restarted counters, clearing the frequent set after each measurement period caused that frequent items were not counted for some time at the beginning of each period. The optimisation was to leave the frequent items entries in the frequent

set and clear only the counters. The problem does not exist in our algorithm as the filters *are never* restarted and the measurement is continuous. Thus the improvement does not apply.

Shielding The shielding is an improvement consisting in not performing the sketch update on each item that is found in the frequent set. This can be easily applied in our version of the algorithm. The shielding is turned on for all items that have a hot filter in the frequent set. The main purpose of the optimisation is to make some hot filters in the sketch cold and reduce the probability of false positives. The side-effect is that if the frequent item becomes infrequent for some time, it will have to wait longer for being included in the frequent set again, as its filters in the sketch might already become cold.

Conservative Update of Counters Instead of increasing the counters at all the stages about the full amount, only the smallest one is increased about the full amount and the rest is set to the maximum of the old value and the new value of the smallest counter. This prevents counters to increase too fast and reduces the probability of false positives even further. The technique is applicable to the low-pass filters. The value v in (1) is set appropriately at each stage just in the same way as it would be with simple counters.

5 Experiments

To check correctness of the algorithm, we have implemented a simple random item generator that could generate a sequence of random items with a custom defined frequency distribution. Generated sequences of items were given to the input of our frequent event mining algorithm implementation. The process lasted until several millions of items were analysed and at the end the frequent category set was printed out. We also measured the maximum size r_{\max} of the frequent set during the program execution and calculated statistics of filter state values in the sketch.

At first we checked if the algorithm was properly implemented and if it really detected frequent categories. The Zipf distribution with exponents $s \in \{1, 2\}$ were used. The categories were given a rank so that the most frequent had the rank 1, and the last had the rank N . The program printed out only the first few categories from the set, so we concluded it worked. By setting various parameters, we observed how the sizes of the frequent set changed. For some settings, we could notice some false positives - the categories with very high ranks, which could not become frequent by accident. Some results are shown in Table 1. Every repetition of the program run with same settings did not provide same results every time. Though the first items were always the same, various categories were reported as the categories with the lowest frequencies. This was due to the random character of the test. Because the most recent history of the stream has a high importance in our solution, a generally infrequent item could

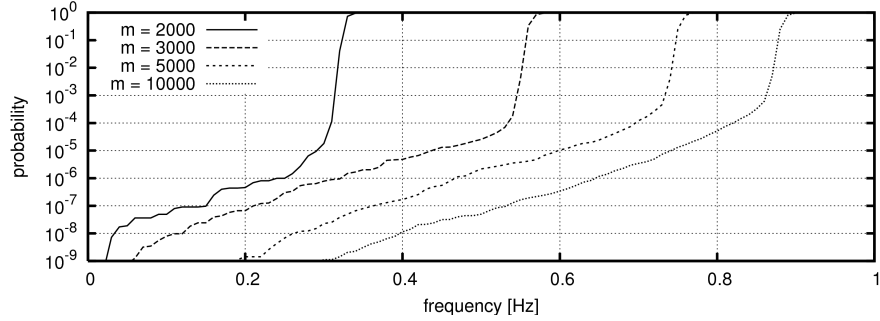
cross the threshold only for some time near the end of the experiment and be reported as a frequent one. This was proved by reporting the times, when the categories were added to the frequent set. The least frequent reported items were usually added just near the end of the experiment, while the most frequent ones – just after the beginning.

Table 1: Sample results of the test program, $n = 5 \times 10^6$, $N = 10^6$, $f_{\max} = 1000$ Hz, $\tau = 100$ s

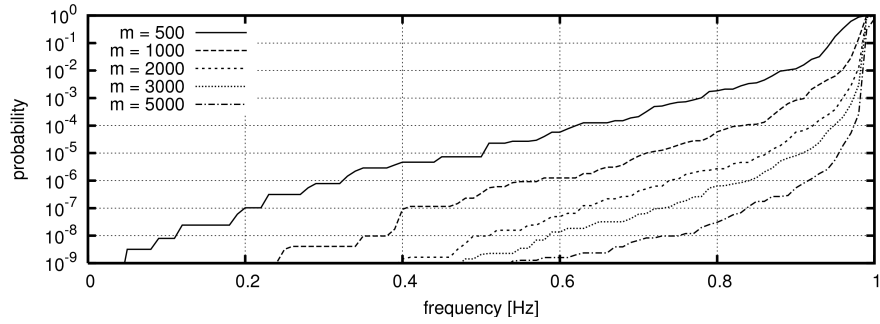
s	$f_{\text{thr}}[\text{Hz}]$	m	M	r_{\max}	result
1	10.0	1000	5	7	(1, 2, 3, 4, 5, 6, 7)
1	5.0	1000	5	15	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
1	2.0	1000	5	37	(1, 2, 3, ..., 24, 26, 25, 27, 28, 30, 29, 32, 31, 33)
1	5.0	250	2	57	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
1	5.0	500	3	14	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)
1	5.0	1500	5	15	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
2	2.0	200	2	19	(1, 2, 3, ..., 9, 10, 11, 12, 13, 14, 15, 16, 17)
2	1.0	200	2	26	(1, 2, 3, ..., 17, 18, 19, 20, 21, 22, 23, 25)

We have also investigated, how the values of certain parameters affect the results. Probabilities of false positives were estimated basing on the final state of the sketch (Fig. 2). As presumed, the probability of adding an infrequent category with a frequency only a little lower than the threshold is very high, but below some level drops very quickly. Thus the algorithm is especially well suited for distributions, where the number of items with very low frequencies can be huge. The more memory is available for the sketch, the less false positives can slip into the result. We observed a very interesting behaviour when changing the number of stages for the fixed sketch size (Fig. 3). The number of stages set too low may cause that, for the large values of N , the expected value of the number of reported infrequent categories may be high. The probability may not fall enough quickly to compensate the increasing number of categories with low frequencies. On the other hand, setting this number too high may cause that the real frequency threshold may be located far below the original, intended by the user. Note that though the probability is very high above this threshold, in general there is no guarantee it will be equal to 1.

In the end we checked how the result set adapts to the frequency distribution changes over time. The test program significantly lowered the probability of items of one of the frequent categories after a half of iterations. It caused that category to become infrequent. The program correctly removed that category from the frequent set after some time, usually slightly exceeding τ . The same experiment was made with adding a new frequent item. In this case a new item was detected faster.



(a) $s = 1$



(b) $s = 2$

Fig. 2: Probability distribution of false positives obtained for a Zipf distribution of categories, for different sizes of the sketch, $n = 5 \times 10^6$, $N = 10^6$, $M = 5$, $f_{\max} = 1000$ Hz, $f_{\text{thr}} = 1$ Hz, $\tau = 100$ s

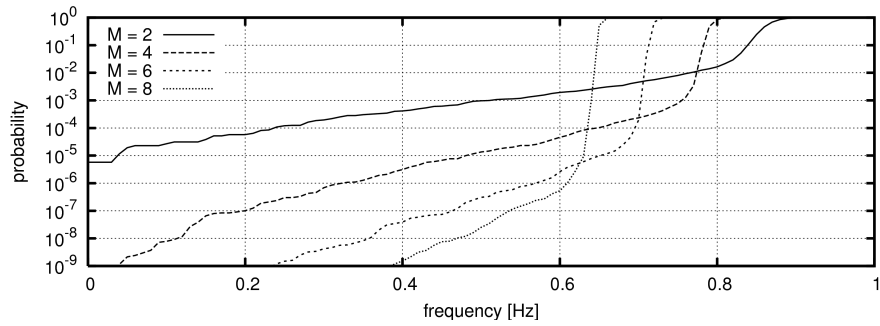


Fig. 3: Probability distribution of false positives obtained for a Zipf distribution of categories, for different number of stages, $s = 1$, $n = 5 \times 10^6$, $N = 10^6$, $m = 5000$, $f_{\max} = 1000$ Hz, $f_{\text{thr}} = 1$ Hz, $\tau = 100$ s

6 Conclusions

The experiments proved our improved version of the multistage filtering algorithm to work correctly. The algorithm gained ability to give stable results at any time while still using very little memory just as in the original algorithm described by Estan et al. [7]. Though it does not implement a standard, sharp-edged sliding window behaviour, the fuzzy, exponential window seems to be useful wherever there is a need to continuously monitor the frequent item set.

We think the modified algorithm is capable of being implemented in hardware, with small static associative memory, and can be employed to analyze network traffic at very high data rates, without a need for data packet sampling [11]. The exponential function in the low-pass filter would not be a problem, while good fast hardware implementations exist [12]. Unfortunately, we did not have a possibility to implement our ideas in hardware, so this remains an open research problem.

References

1. Lan, K., Heidemann, J.: A measurement study of correlations of internet flow characteristics. *Comput. Networks* **50**(1) (2006) 46–62
2. Boyer, R.S., Moore, J.S.: MJRTY: A fast majority vote algorithm. Technical Report 35, Institute of Computer Science, Texas University (1981)
3. Misra, J., Gries, D.: Finding repeated elements. Technical report, Cornell University, Ithaca, NY, USA (1982)
4. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, London, UK, Springer-Verlag (2002) 348–360
5. Manku, G., Motwani, R.: Approximate frequency counts over data streams. In: *Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, August 2002. (2002)
6. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming*. (2002)
7. Estan, C., Varghese, G.: New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.* **21**(3) (2003) 270–313
8. Chang, J.H., Lee, W.S.: estWin: Online data stream mining of recent frequent itemsets by sliding window method. *J. Inf. Sci.* **31**(2) (2005) 76–90
9. Cormode, G., Muthukrishnan, S.: What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.* **30**(1) (2005) 249–278
10. Kołaczkowski, P.: Using low-pass signal filtering for continuous database load estimation. (2007) submitted to BDAS'07 conference, Ustroń, Poland.
11. Gibbons, P.B., Matias, Y.: New sampling-based summary statistics for improving approximate query answers. In: *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, New York, NY, USA, ACM Press (1998) 331–342
12. Kantabutra, V.: On hardware for computing exponential and trigonometric functions. *IEEE Trans. Comput.* **45**(3) (1996) 328–339