A New Algorithm for Finding Frequent Items in Streams of Data [*]

Klemen Simonic
Jozef Stefan Institute
klemen.simonic@ijs.si

Janez Brank
Jozef Stefan Institute
janez.brank@ijs.si

Marko Grobelnik
Jozef Stefan Institute
marko.grobelnik@ijs.si

ABSTRACT

Given a stream of queries posed to an Internet search engine, how should we detect the popular queries? Or, which are the frequent transactions in an enormous collection across all branches of a supermarket chain? Or, how quickly can we declare a query or transaction as being frequent?

These kinds of data usually arrive at enormous rates, hundreds of gigabytes per day or even more, but we are given only a small fraction of resources compared to the total quantity of data. Are we able to process the data in real time and provide up-to-minute analysis and statistics on current trends?

We present a new algorithm for finding frequent items in a stream of data that handles the above mentioned problems in a better way than other known algorithms. We evaluated our algorithm on several large real-world data streams and made several experiments in order to optimize the algorithm. We applied our algorithm to the task of detecting frequent n-grams (sequences of n adjacent words) in streams of text. This way, we are able to detect popular phrases on the Internet "as it happens".

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithm and Problem Complexity**]: Nonumerical Algorithms and Problems

General Terms

Algorithms; Experimentation

Keywords

Algorithm; Data Stream; Frequent Items; n-grams

1. INTRODUCTION

Nowadays, the Internet is gaining more and more popularity. Each day, we send billions of requests to websites, pose millions of queries to an Internet search engine, watch videos over the Internet, use social websites etc. We also make millions of transactions every week. For example, we go to a shop and pay the bill, make a money transaction over the Internet, etc.

These data generation processes and many others can be interpreted as data streams. They usually arrive at enormous rates and produce a huge amount of data. These data can be stored inside a database and then analyzed later, but it is also important and often more valuable to provide real-time analysis and statistics. This requires algorithms and techniques to work under additional constraints which are commonly referred to as the *streaming model*. Under this model, an algorithm must make only a single pass over the input stream and its resource requiremens (average amount of time and memory used per item processed) must be low (sublinear in the stream length; ideally they will be polylogarithmic or even constant). Often, algorithms that produce approximations of the correct results are also considered acceptable in this setting. The output may be produced at the end of the stream or continuously during the processing. The study of problems which can be solved under these constraints and the algorithms used for that purpose is still attracting much research interest.

In the *frequent items problem* (also known as the *heavy hitters* problem), we are given a sequence of items and we want to find those items which occur most frequently. This can be formalized in various ways, e.g. to find the a given number of items with the greatest frequency, or to find all the items whose frequency exceeds a given threshold (possibly expressed as a fraction of the total number of items in the stream). The frequent items problem is an important one to understand in order to produce efficient streaming implementations. Since its beginnings in the 1980s, this area is attracting more and more research interest.

Many real-world streams can be analyzed using this variety of settings. For example, the items can represent the queries posed to an Internet search engine or money transactions. Then the frequent items represent the most popular queries or the most frequent transactions, respectively. Having this knowledge can help us optimize which queries the search engine should cache given the current trends, instead of calculating them again. It is important to find algorithms which are capable of processing each new item very quickly, without blocking and while using a small working space.

Obtaining frequent items is usually one of the first steps in a more complex computational process. In the next steps, for example, we can try to learn something from the data or find hidden patterns, relations between the items. Abstracting knowledge from the frequent items and then trying to use that knowledge on the other items can be very helpful.

Our main problem was finding frequent n-grams (sequences of n adjacent words), of length up to $n = 8$, over the Internet and providing up-to-minute analysis about which n-grams are popular or just became popular. Finding frequent n-grams on the Internet would give us another look on what is popular on the Internet in a text manner. An n-gram can represent words, phrases or even short sentences.

[*](Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

The main contributions of this paper are:

- We exploit properties of the stream to develop FRESHSAVING, a new algorithm for finding frequent items in streams of data. FRESHSAVING solves the frequent items problem almost perfectly with only a few percent of memory compared to total quantity of data. It achieves drastically better results than SPACESAVING, which has been found to be one of the best counter-based algorithms in a recent survey [6].

- We evaluated both algorithms extensively on the problem of finding frequent n-grams in several large real-world streams of textual data.

- We developed a framework for finding frequent n-grams over the Internet which will be able to provide real-time information about which n-grams are popular or just became frequent.

- We calculated n-gram distributions and investigated how Heaps' law behaves if we observe a stream of n-grams instead of individual words.

- We explored and experimented several techniques how to efficiently represent n-grams in memory and developed a specialized hash table with several memory optimizations.

2. FREQUENT ITEMS

2.1 Problem statement

For the purposes of our discussion, a *stream* is simply a sequence of items or elements, $S = (a_1, \ldots, a_\ell)$, where the individual items a_i are drawn from an alphabet Σ which is potentially very large, $|\Sigma| \gg \ell$. The length of the stream, ℓ, can also be denoted $|S|$. The significance of S being a stream instead of arbitrary sequence lies in the fact that the algorithm should read its elements sequentally, make only one pass through the sequence, and that the average amount of time and memory used to process each item should be small. The *frequency* of an item x is simply its number of occurrences: $\phi_x = |\{i : a_i = x, 1 \leq i \leq \ell\}|$.

Several problems related to item frequency and frequent items have been proposed and studied. The *exact ϑ-frequent items problem* is often formulated as follows: given a stream S and a threshold θ, output exactly those x for which $\phi_x \geq \vartheta|S|$. In other words, the threshold applies to the relative frequency of items, rather than the absolute one. This problem turns out to be difficult to solve exactly, even for a high value of θ, which can be demonstrated using the following well-known information-theoretic argument: suppose that we are given a set A and an algorithm that solves the exact ϑ-frequent item problem; we set $\vartheta = 1/2$ and feed all the elements of A into the algorithm, in any order; at this point the algorithm is able to decide, for any item x, whether it is a member of A or not. This can be done by feeding $n-1$ occurrences of x into the algorithm; after this we ask it which are the θ-frequent items. If it reports x as a ϑ-frequent item, we know that x was an element of A; otherwise we know that it wasn't. Since this can be done for an arbitrary x, it means that the algorithm, after processing all the elements of A, must have held information equivalent to the entire membership of A, which requires at least $O(|A|)$ space.

Since in practice a linear space requirement is unacceptable when processing large streams, research has mostly focused on approximation algorithms. One can define the *ε-approximate ϑ-frequent items problem*: given a stream S, a threshold ϑ and an $\varepsilon \in (0, 1)$, output all those x for which $\phi_x \geq \vartheta|S|$, but additionaly we may also output some items x for which $\phi_x \geq (\vartheta - \varepsilon)|S|$.

A related problem is the *eps-approximate frequency estimation problem*: given a stream S and a value $\varepsilon \in (0, 1)$, we want the algorithm to process the stream so as to be able to return approximate frequencies: given an item x, it should give an estimated frequency $\hat{\phi}_x$ such that $|\hat{\phi}_x - \phi_x| < \varepsilon|S|$.

For our approach, we consider a problem setting which differs from the above formulations in the following details. First of all, we consider the amount of available memory to be strictly limited and not allowed to grow as a function of the number of items processed so far. Therefore, what we want from the algorithm is not so much to find all items whose frequency (or relative frequency) is above some threshold, but to find as many of the most frequent items as will fit into the available memory. Thirdly, in addition to outputting the frequent items themselves, the algorithm should be able to output estimates of their frequencies.

We believe this formulation makes sense from a practical point of view, where the amount of memory available for processing is usually fixed and known in advance and the user cannot allow the algorithm's memory consumption to grow indefinitely. Given this constraint it is also impractical to formulate the problem by using a minimum frequency threshold, because the number of distinct items that climb above that threshold may be too large for the available memory. This is especially true if the threshold refers to absolute frequency ϕ_x instead of relative frequency $\phi_x/|S|$; in this case the number of frequent items may practically grow indefinitely as the stream gets longer and longer and it's hard for the user to set a reasonable threshold in advance without knowing quite a bit about the item frequency distribution in the stream. Thus we assume that it is the most frequent items that are the most interesting to the user, and the goal of the algorithm should be to make the most use of the available memory and find as many of the most frequent items as possible.

2.2 Properties of the stream

Many real-world streams that are of interest in practical applications have specific characteristics that a frequent-item finding algorithm might be able to make use of to achieve better performance.

One such characteristic is that the vast majority of items are relatively infrequent; the number of frequent items is relatively small, therefore an algorithm can hope to achieve good results at finding frequent items while using an amount of memory that is small relative to the amount of data in the stream being processed.

Another characteristic is that the occurrences of any particular item are usually not spread uniformly around the stream. Instead, they might occur in bursts or clusters; there might be (relatively short) periods of time when an item occurs fairly frequently, interspersed by long periods when it occurs much more rarely.

3. PROPOSED ALGORITHM

3.1 Idea of the algorithm

Our algorithm belongs to the family of counter-based algorithms. That is, it maintains a store of items in main memory, along with a counter for each of these items. The purpose of the counter is to contain information about the frequency of the item in the part of the stream that has been processed so far. If the amount of available memory was unlimited, the task would be trivial: every time an item is read from the stream, we would check if it is already in the store; if yes, we would increment its counter by 1, otherwise we would add it to the store and set its counter to 1. But since the amount of available memory is limited (and is typically rather small relative to the length of the stream), the algorithm must be able to decide which items to actually keep in the store and how to update their counters.

In a typical real-world stream, items might vary greatly in their frequency. There would typically be a small number of very frequent items that occur all the time and won't be difficult for a counter-based algorithm to identify. There would be a large number of infrequent items that won't be of interest for the task at hand. But there would also be a nontrivial number of items that are moderately frequent; they do not appear all the time but are frequent enough that we would probably want to output them. The challenge for the algorithm is how to identify such moderately frequent items and deal with the deluge of infrequent ones.

Now suppose that we are processing a stream and we have encountered an item which is not currently present in our store. We would like to add it into the store; the question is what to do if the store is full and we cannot increase it because this would exceed the amount of memory available to us. In this case we have to delete an item from the store before adding the new item into it. A naive approach to the deletion of items might be to delete the item with the smallest frequency, but it can be easily seen that this would unfairly favour items which first appeared early in the stream, before the store was full. After that, recently added items, having a small value of the frequency counter, would be the first candidates for removal and even if they were later added back, their counter would start at 1 again; thus they would have a difficult time climbing in frequency and getting themselves established in the store.

The intuition behind our algorithm is that in many real-world data streams, interesting moderately frequent items do not occur uniformly over the entire stream. Instead, such an item might occur in "bursts" caused by some underlying event or change in the process that generates the stream. For example, let's say that we have a stream of words from news articles; the word "earthquake" is not very frequent overall, but there will come periods of time when an earthquake occurred and the word "earthquake" was mentioned frequently for a while.

Thus, when a new item occurs and is added to the store, it might be beneficial to protect it, for a period of time, from being removed from the store. If additional occurrences of this item appear in the near future, its frequency counter in the store will grow and subsequently this frequency itself will be enough to prevent it from being deleted from the store (unless it turns out that, in the long term, it really isn't frequent enough after all).

Therefore we will maintain, in addition to the store, a sliding window containing a certain number of the most recently processed items from the stream. When a new item is read from the stream, it is added to the end of the sliding window, and (if the window is full) the oldest item is removed from it. The store will actually maintain two counters for each item, one for its overall frequency and one for its frequency in the sliding window only. With this additional counter, which we will refer to as *status*, it is always easy to determine if an item has occurred recently (i.e. is present in the sliding window) or not. We will say that the items present in the sliding window are *fresh items*, those that aren't are *unfresh items*, and we will refer to the sliding window itself as the *fresh buffer*. The main idea of our algorithm then is that, when an item has to be deleted from the store to make space for a new item, the item to be deleted is chosen from among unfresh items. Fresh items are guaranteed to remain in the store at least as long as they are fresh; hence, we call our algorithm FRESHSAVING.

Thus an item that recurs in the stream within a period that is shorter than the length of the fresh buffer will not get deleted from the store. There might also be some reasonably frequent items whose reoccurrence period exceeds the length of the fresh buffer; for this kind of items, we will see later that by adding a few more restrictions on item deletion, we can catch most of the frequent items.

The following pseudocode shows a high-level description of our algorithm:

```
algorithm FRESHSAVING:
Input: a stream $S$, store capacity $M_h$,
       fresh buffer capacity $M_f$.
Output: a set of frequent items and their frequencies.
1  let $H$ be the store, initially empty, and let
   FB be the fresh buffer, initially an empty sequence;
2  for each item $x$ from the stream $S$:
3      if $|FB| \geq M_f$:
4          let $y$ be the first element of FB;
5          delete the first element of FB;
6          in $H$, decrease the status of $y$ by 1;
7      append $x$ at the end of FB;
8      if $x \notin S$:
9          if $|H| \leq M_h$:
10             let $y$ be some unfresh element of $H$
               (i.e. one whose status is 0);
11             delete $y$ from $H$;
12         add $x$ to $H$, with $status = frequency = 0$;
13     in $H$, increase the status and freqency of $x$ by 1;
14 output the items from $H$, along with their frequencies.
```

The main open questions at this point are how to efficiently implement the store H and the fresh buffer *FB*, and how to select items for deletion in step 10. This will be dealt with in the following sections.

Note that the frequency associated with an item x in the store H is not necessarily equal to the total number of occurrences of x in the part of the stream that has been processed so far. Instead, the frequency in H counts just the number of occurrences of x since the last time when it was added into the store. If an item x is added to the store, then deleted, then added again at some later point, its frequency as stored in H will always be strictly less than the actual number of occurrences of x in the stream. Thus, from the point of view of frequency estimation, our algorithm is biased in a conservative direction; it tends to underestimate the frequency of items, but it never overestimates them.

3.2 Data structures

As we have seen in the previous section, our algorithm requires two data structures: the store and the fresh buffer. The fresh buffer, whose capacity is limited to M_f items, can be implemented efficiently as a circular buffer (a.k.a. ring buffer) in an array of M_f elements, with two integer indices pointing at the oldest and newest elements in the fresh buffer. Each element of the array contains simply an item of the input sequence, with no additional data.

The store is slightly more complex. It has to be able to maintain a set of items, along with the frequency and status of each item; and it has to efficiently support the addition and removal of items, as well as accessing and modifying the frequency and status of a given item. Thus, a hash table is a natural choice of data structure to implement the store. However, since we want our algorithm to be able to deal with large datasets under a strict memory limit, it is worth discussing some details regarding the hash table implementation, with a view to speed and efficient use of memory.

For reasons both of space and time efficiency, we decided not to use a closed hash table; there, a substantial proportion of table elements would have to be kept empty to avoid having to traverse long chains of elements during lookups. Thus, either the available memory would be poorly utilized, or lookups would be slow. A closed hash table is also less convenient for situations where items frequently have to be deleted. Therefore we decided to use an open hash table instead.

Conceptually, an open hash table is a sequence of M_m lists, where the i'th list contains exactly those elements whose hash code is i. A traditional implementation would use linked lists, in which each node contains a key, its corresponding satellite data, and a pointer to the next node in the list; in addition there would be an array of M_m pointers to the first elements of the lists. Our implementation differs from this in two details.

(1) Instead of having an array of M_m pointers to the first elements of the lists, the array will contain the first element of each list. We will refer to this array as the *main table*. Using the main table saves some space (for M_m pointers). It can also lead to some wasted space if some of the lists are empty (because no items have hashed into that particular hash code so far), since the main table will still contain (unused) entries for them; however, as we process more and more items from the input stream, eventually practically all the hash codes get used up and the amount of wasted space in the main table tends towards 0.

(2) The remaining elements (i.e. all non-first elements of all the lists) are not allocated individually on the heap and linked by pointers, as would be the case in a traditional linked list. Instead, they are all stored in a second large array, which we call the *overflow table*. In each entry we now have the index of the next element in the list, instead of a pointer to it. (Similarly, the main table will also contain indices into the overflow table, rather than pointers to its entries.) The unused entries in the overflow table are connected into a linked list of their own; to allocate a new entry, we remove (and use) the entry from the start of this list, and similarly when an entry is deallocated we add it to the beginning of this list. This memory management scheme for the overflow table is simple and efficient, and it ensures that there will be no hidden overheads that might be introduced by relying on an external heap manager. Additionally, there may be situations where linking the entries using indices instead of pointers can lead to a smaller memory consumption. For example, in a 64-bit application the pointers would occupy 64 bits each, but if the capacity of our overflow table does not exceed 2^{32}, we can use 32-bit indices and thus save some space.

Although the distinction between the main table and the overflow table is conceptually useful, the structure of the entries is exactly the same in both tables, so they could in principle be allocated as one large array. Each entry contains the following four fields: *key* (one of the items from the input stream), *frequency* (the number of occurrences of this item in the stream since the last time it was added into the hash table), *status* (the number of occurrences of this item in the fresh buffer) and *next* (the index of the next item with the same hash code as this item). In the *next* field, two index values are reserved for special purposes: we use an index of 0 to indicate that there is no next element in the linked list (i.e. the current item is the last item with this hash code), and an index of 1 to indicate that the current entry is not used at all (i.e. it does not contain a valid key and other fields; this is only used in the main table, not in the overflow table). We keep the first two entries in the overflow table unused, because their indices have been employed for these special cases.

As we saw in the previous section, we assume that the maximum capacity of our hash table, M_h, is specified by the user in advance. This memory is divided between the main table and the overflow table: the main table has M_m entries and the overflow table has $M_h - M_m$ entries. We will assume that the ratio M_m/M_h is also fixed and specified by the user, just like the total capacity of the hash table itself. Changing this ratio dynamically during the operation of the algorithm would require a rehash of all the items, a time-consuming operation which would be hard to in-place and which would be of dubious utility anyway.

The choice of M_m/M_h is subject to a tradeoff: since wasted entries can only occur in the main table, but not in the overflow table, having a smaller main table leads to a more efficient use of the available memory. On the other hand, if we have approximately M_h elements in the hash table, divided into at most M_m linked lists, the average length of each list is approx. M_h/M_m, so having a smaller main table means having to traverse longer lists, which slows down all the hash table operations. We will discuss an experimental investigation of the role of M_m/M_h in section 4.4.2.

3.3 Keeping count approximately

An approach that can potentially lead to a further reduction in the amount of memory used for each hash table entry is to maintain approximate instead of accurate values in the *frequency* and/or *status* fields. The idea is to map the original range of possible values of these fields into a smaller range, so that after this mapping the values can be represented by a smaller number of bits. On the downside, this mapping cannot be injective and the distinction between some values will be lost after the mapping; that's why the representation of frequencies or statuses is now only approximate, not accurate.

Let $u(f)$ be a suitably chosen slowly increasing function of f, defined for $f \geq 0$, with $u(0) = 0$ and $0 < u'(f) \leq 1$ for all f. We want our approximate counter c to store the value $u(f)$ instead of the correct frequency f. When a new occurrence of the item appears, f grows to $f + 1$ and the counter should grow from $u(f)$ to $u(f + 1)$. But since the counter is an integer variable, we cannot increase it by $u(f + 1) - u(f)$. Instead, we will roll a random number and increase c by 1 with probability $u(f + 1) - u(f)$, and leave it unchanged with probability $1 - (u(f + 1) - u(f))$. Thus, the expected increase of c will be $u(f + 1) - u(f)$, which is just what we want. The above-mentioned requirement that $u'(f) \leq 1$ ensures that $u(f + 1) - u(f)$ won't be greater than 1.

The following pseudocode describes the operations on an approximate counter.

```
Initialization: set $c \leftarrow 0$;
Increment:
  let $\hat{f} \leftarrow u^{-1}(c)$;
  with probability $u(\hat{f} + 1) - u(\hat{f})$, increase $c$ by 1;
Decrement:
  let $\hat{f} \leftarrow u^{-1}(c)$;
  with probability $u(\hat{f} + 1) - u(\hat{f})$, decrease $c$ by 1;
```

If we start the counter at 0 and call the increment function f times, the value of c after all these calls will be approximately $u(f)$. By examining the value of the counter, c, we can infer that the correct frequency is approximately $u^{-1}(c)$.

For u one might choose a function that grows logarithmically; for instance, to map frequencies from the range $[0, f_{max}]$ into $[0, u_{max}]$, we could use $u(f) = \alpha \ln(f + 1)$ where $\alpha = u_{max}/\ln(f_{max} + 1)$. However, the derivative $u'(f) = \alpha/(f + 1)$ is > 1 for $f < \alpha - 1$, meaning that, for low values of f, the mapping function is wasting space in the target range and our increment/decrement functions wouldn't work there because they always modify the counter by at most 1. Thus a more suitable function u would consist of a linear part for low frequencies and a logarithmic part for higher frequencies:

$$u(f) = \begin{cases} f & \text{if } f < \alpha \\ \alpha + \beta(\ln f - \ln \alpha) & \text{if } f \geq \alpha \end{cases}$$

where the scaling factor β is defined as $\beta = \frac{u_{max} - \alpha}{\ln f_{max} - \ln \alpha}$. This ensures that u is continuous at $f = \alpha$ and that $u(f_{max}) = u_{max}$. We must choose α so that the derivative u' will be ≤ 1: $u'(f) = \beta/f$ for the logarithmic part, and the requirement $u'(f) \leq 1$ will be met (for all $f \geq \alpha$) if we choose α such that $\beta < \alpha$. For example, if we want to map 64-bit frequencies into 16-bit values, we have $f_{max} = 2^{64} - 1$ and $u_{max} = 2^{16} - 1$; it turns out that the smallest suitable integer α is $\alpha = 4228$. In effect this means that we'd be using a 16-bit integer counter to store our frequencies, with frequencies up to 4228 being stored accurately and larger ones being stored approximately.

3.4 Removal of items from the store

As we saw in section 3.1, when we read an item x from the stream we want to add it to the hash table if it isn't there yet. But the capacity of the hash table is limited, so it might happen that some unfresh item will have to be deleted from the hash table first. Which item should we delete to make space for the new one?

A natural first idea would be to always remove the unfresh item with the lowest frequency. However, this idea has several drawbacks. To find such an item efficiently, a hash table is not enough; we would also have to maintain a heap with the unfresh items. There are usually many such items (the fresh buffer would typically be small relative to the hash table), so having a heap like that would significantly increase our memory consumption. In addition, in some of our preliminary experiments this aggressive approach to deleting items with the lowest frequency led to poor results. Instead, we decided to use a randomized approach. We randomly select r unfresh items and delete the one with the lowest frequency. A smaller r means faster removal of items but potentially worse results; on the other hand an r that is too large would slow the algorithm down and possibly also lead to worse results. We use $r = 4$ as a good compromise between performance and speed.

There are a few other details related to the removal of items that one needs to be aware of, given our current implementation of the hash table. First of all, the need for deletion arises if overflow table is full and the new item x maps to a hash code $h(x)$ for which the corresponding entry in the main table is occupied. (There might at that time still be some empty entries in the main table, but they are of no use for storing x; this is a drawback of our implementation of the hash table but, as we said earlier, over time the number of such entries tends towards 0.) This means that we have to free up an entry in the overflow table. This can be done either by deleting an item that is already in the overflow table, or by deleting an item that is in the main table but whose chain continues into the overflow table; in the latter case, the next item in the chain can be moved from the overflow table into the main table, thereby freeing an entry in the overflow table. Thus, items that are in the main table but whose *next* is null are *not* candidates for deletion, because deleting them would not free up an entry in the overflow table and would thus not help us store the new item x.

Due to these additional constraints, simply choosing a random entry from the hash table will not necessarily yield an item suitable for deletion. In principle we'd like to keep trying until we've found r suitable items. However, theoretically it could happen that there are very few (or even none) suitable items. Thus, as a safety measure, our deletion algorithm is actually as follows: we make at most T random probes; if we find r suitable items we stop immediately; we delete the least frequent of the suitable items examined; if we found no suitable items, we don't delete anything and we won't be adding our new item x into the hash table at all.

```
1   $t \leftarrow 0$; $i \leftarrow 0$; $f^\star \leftarrow \infty$;
2   while $t < T$ and $i < r$ do:
3       let $Y$ be a random entry from the hash table;
4       $t \leftarrow t + 1$;
5       if $Y$ is in the main table and is either an unused
        entry or its *next* is null, go to step 2;
6       let $y$ be the item stored in entry $Y$,
        and let $f$ be its frequency;
7       if $y$ is unfresh, go to step 2;
8       $i \leftarrow i + 1$;
9       if $i = 1$ or $f < f^\star$ then $y^\star \leftarrow y$, $f^\star \leftarrow f$;
10  if $i > 0$ then delete $y^\star$ from the hash table;
```

The problems against which this deletion algorithm tries to guard — the non-existence or extreme scarcity of items suitable for deletion — generally mean that the parameters M_f, M_m and M_h have been chosen unwisely: the fresh buffer is probably too large and the overflow table too small. The best solution in that case is to modify these parameters and run the algorithm again. In any case, these problems did not occur in the experiments presented in this paper and with the current version of the algorithm.After some preliminary experiments, we set r to 3 and left T practically unlimited by setting it to 1000.

4. EXPERIMENTS

4.1 Frequent *n*-grams

Although our algorithm can in principle work on any stream, the underlying motivation for our work was finding frequent n-grams (sequences of n adjacent words) in a stream of text. For example, finding frequent n-grams on the Internet might give us an insight into what is popular on the Internet in a textual manner. We investigated n-grams of length up to $N = 8$ words.

Given a chronological sequence of documents, we start by applying several preprocessing steps: we extract text from the documents (discarding XML/HTML tags etc.), convert it into lowercase and split it into words. We did not perform any stemming, lemmatization or stopword removal. This results in a sequence of words, $w_1, w_2, \ldots, w_\ell$. We will denote by w_{in} the n-gram $(w_{i-n+1}, \ldots, w_i)$. Of the n-grams that end with w_i, the following N are of interest to us: $w_{i1}, w_{i2}, \ldots, w_{iN}$. If we concatenate these sequences together, for all i from N to ℓ, we obtain a sequence of n-grams (of various lengths) with $(\ell - N + 1)N$ elements. For each of these n-grams, we take its string representation and compute a 64-bit hash code from it, using the SDBM hash function. This results in a sequence of $(\ell - N + 1)N$ integers, which is the input into our algorithm.

This sequence will be adequate for our main goal in these experiments, which is to evaluate our algorithm from the point of view of finding frequent items in a stream. However, from the point of view of a user interested in frequent n-grams it would be insufficient because it deals only with 64-bit hash codes of n-grams rather than with their string representation. Various extensions could be considered to make the algorithm display the string representation of frequent n-grams. One possibility is to store, in main memory, the string representation of every n-gram that is currently present in the hash table H. However, this would consume a prohibitively large amount of memory.

An alternative is to keep working with 64-bit hash codes, as we described above, but to ask the user to select a threshold ϑ; then, when we process a n-gram x, if we notice that its frequency counter in the hash table has increased from $\vartheta - 1$ to ϑ, we output its string representation (which we still have because we have just read it from the stream). The advantage of this approach is that it outputs many frequent n-grams at an early point and without consuming any extra memory; the downsides are that the output does not include the final frequencies of those n-grams and that a n-gram may potentially be output multiple times (if it falls out of the hash table at some point and later re-enters it, climbing to a high enough frequency again).

In any case we consider the problem of how to provide the user with string representations of frequent n-grams, rather than just with their hash codes, to be outside the scope of this paper.

4.2 Experimental setup

We used the well-known RCV1 corpus [18] for our experiments. This is a collection of approx. 800 000 Reuters news articles from 1996 and 1997. They can be ordered chronologically, which is important for our purposes because it enables us to think of them as a stream of textual data. The total length of this corpus is about 1.3 GB. After preprocessing this resulted in approx. $\ell = 175 \cdot 10^6$ words. Generating all n-grams of lengths up to $n = 8$ (as described in sec. 4.1) gave us a sequence of approx. $1.5 \cdot 10^9$ items, with each item being a 64-bit hash code of an n-gram. Thus the total length of our input sequence was approx. 10.4 GB; this will be the reference point for controlling the memory consumption in our experiments.

We ran our experiments on a computer server with 64-bit architecture, 24 CPUs and 128 GB of RAM. The algorithm and the framework were implemented in C++ and compiled using Microsoft Visual Studio.

In addition to our algorithm, we implemented SpaceSaving [14], which has been found to be the most successful counter-based algorithm in a recent survey [6]. SpaceSaving uses a hash table to store item frequencies, but in addition to that it also stores all these items in a heap, where the items with smaller frequencies are higher up in the heap. When it is necessary to delete an item (to make space for a new one), the one from the root of the heap (i.e. with the smallest frequency) is deleted; but when the new item is then added into the hash table (and the heap), its frequency doesn't start at 1, but at $1 + f$, where f is the frequency of the item that has just been deleted from the store. This enables the algorithm to provide a theoretical guarantee on its results: the frequencies it has stored alongside the items differ from the correct frequencies by at most ℓ/M, where M is the memory capacity (maximum number of items stored in the heap) and ℓ is the length of the input stream. Unlike our algorithm, which can underestimate the correct frequency of an item but never overestimate it, SpaceSaving can either under- or overestimate the frequencies.

To give SpaceSaving a fair chance for good performance, we used the following not entirely obvious optimization to save space and fit more items into the available memory: the hash table contained, for each item, its frequency counter and its index in the heap, *but not the item itself*; the heap, on the other hand, contained only items and nothing else. Hash table lookups are now slightly more time-consuming because the items are not available directly in the hash table and an extra lookup into the heap is necessary at each step while walking through the hash table entries; but by not having to store the items twice (in the hash table and in the heap), we can fit a lot more items into the available memory.

We used several evaluation measures to evaluate the output of the two algorithms. One way is to assume that the user has defined a threshold ϑ; items that occur at least ϑ are considered frequent, the rest are infrequent, and the user wants to get a list of all the frequent items (and no infrequent ones). In this formulation the problem becomes similar to retrieval or binary classification (frequent = positive class, infrequent = negative class) and we can use well-known evaluation measures from information retrieval and machine learning, such as the area under the ROC curve (AUC) [10], precision, recall, and F_1 (harmonic mean of precision and recall).

We also used the following measure, which does not rely on a user's threshold. Let $\{x_1, \ldots, x_k\}$ be the set of items stored by the algorithm in memory at the end of the stream. Since the aim of the algorithm has been to identify the most frequent items, we may consider the sum of the frequencies of the items it has actually found as a measure of its sucess. For normalization, we divide it by the sum of the frequencies of the k most frequent items in the stream. We will refer to this measure as the *frequency sum ratio* or *FSR*. Like the measures from the previous paragraph, it lies in the range $[0, 1]$ with higher values indicating better performance.

The chief independent variable in our experiments has been the amount of main memory available to the algorithm for processing. We defined this memory constraint, m, as relative to the size of the dataset, and used the following values: $m = 0.05$ (i.e. 5 %, or 534 MB), 0.01 (i.e. 1 %, or 107 MB) and 0.001 (i.e. 0.1 %, or 10.7 MB). For the evaluation measures which depend on a threshold (AUC and F_1), we used the thresholds $\vartheta \in \{5, 10, 20, 40\}$.

4.3 *n*-gram distribution

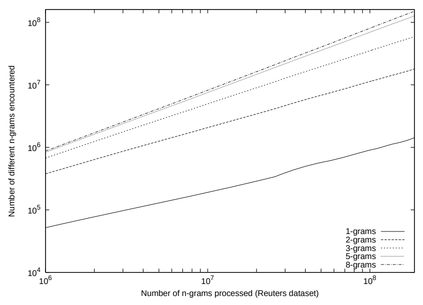
To get a better understanding of the data we're dealing with, we started with some analyses of n-gram distributions in our stream. Let $f_n(k)$ be the number of distinct n-grams among the first k n-grams in our stream. For $n = 1$, i.e. when dealing with a sequence of individual words, it has been observed (*Heaps' law*) that $f_1(k) \propto k^{\beta_1}$, where the value of β_1 has usually been empirically estimated to be in the 0.4–0.6 range for English text. An interesting question is whether the same observation holds (with a different exponent, β_n) for $n > 1$. In other words, how does the number of distinct n-grams grow as a function of the amount of text processed? This is also a question of practical interest, e.g. when trying to plan the memory consumption of an algorithm.

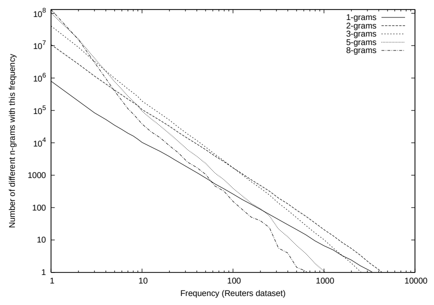
The left chart on figure 1 shows the functions $f_n(k)$ for $n = 1, \ldots, 8$. As can be seen, the log-log plot of these functions yields straight lines, indicating that Heaps' law applies to them as well. However, the exponents β_n quickly approach 1 as n grows towards 8. This confirm the well-known empirical observation that the number of distinct n-grams for larger values of n tends to be linearly proportional to the length of the corpus processed. The empirically estimated values of β_n are shown in the table in fig. 1.

Another interesting question is the distribution of n-gram frequencies. Let $g_n(k)$ be the number of n-grams that occur k times in our corpus. There are a few frequent n-grams and many infrequent ones; as with many other phenomena, this distribution can be described by a power law: $g_n(k) \propto k^{-\gamma_n}$. This observation is also of practical interest, as many algorithms rely on it to a greater or lesser degree [5]; after all, our algorithm also works on the assumption that the number of frequent n-grams is relatively small compared to the total number of distinct n-grams. Figure 1 shows the functions $g_n(k)$ on a log-log plot. However, because the actual frequencies are discrete values, the graph of $g_n(k)$ can get messy for large values of k where the power law would predict a small value between 0 and 1, whereas the actual value of $g_n(k)$ oscillates wildly between 0 and 1. Therefore the chart actually shows the functions smoothed using exponential binning: $\hat{g}_n(k) \triangleq \text{average}\{g_n(k') : k/q \leq k' \leq qk\}$ for $q = 10^{0.05}$. From these smoothed functions, we estimated the exponents γ_n; they are shown in the table in fig. 1. As we look at larger values of n, we can see that there are more and more infrequent n-grams, but fewer and fewer frequent ones.

4.4 FreshSaving results

As we saw in sec. 3.1, our algorithm relies on the user's parameters to define the capacity of the fresh buffer, main table, and overflow table. Since the only externally imposed constraint is the total amount of available memory, M, this leaves open the question of how to divide this memory among the various data structures. We will control this via two parameters: the size of the fresh buffer relative to the hash table, M_f/M_h; and the size of the main table relative to the entire hash table, M_m/M_h. We varied both of these parameters in the range $[0.3, 0.7]$, in increments of 0.1.





	Heaps'	Freq.
n	β_k	γ_k
1	0.666	1.761
2	0.734	2.074
3	0.847	2.347
4	0.923	2.488
5	0.962	2.542
6	0.979	2.583
7	0.986	2.621
8	0.990	2.682

Figure 1: *Left:* the number of distinct n-grams as a function of the total number of n-grams processed. *Center:* the number of distinct n-grams with a particular frequency, as a function of that frequency. *Right:* Heaps' and frequency distribution exponents for n-gram sequences.

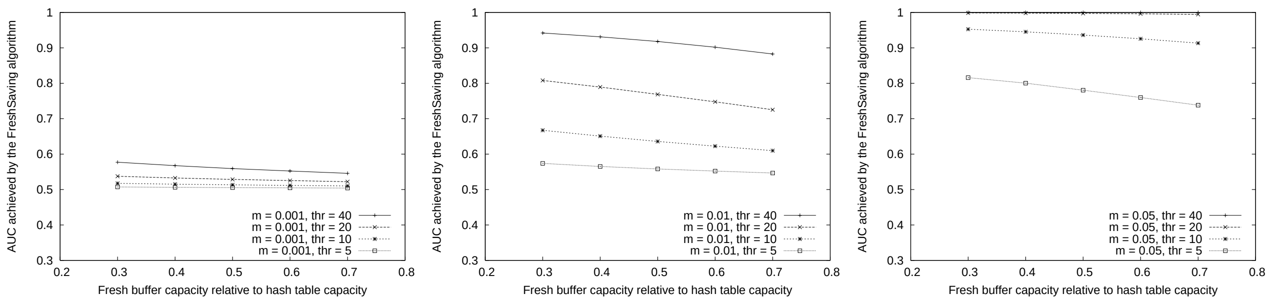


Figure 2: Influence of the fresh buffer capacity on the performance of the FreshSaving algorithm.

4.4.1 *Fresh buffer vs. hash table*

We'll start with an investigation of the role of the relative size of the fresh buffer and the hash table. Intuitively one would expect that this parameter is subject to a trade-off. Having a smaller fresh buffer means that more memory is available to the hash table, which can therefore store more items, leading to better performance. On the other hand having a smaller fresh buffer also means that some items whose reocurrence period exceeds the length of the fresh buffer might be deleted from the hash table even if their overall number of occurrences might be high enough to make them of interest for the user. However, it turned out that in our experiments this tradeoff did not occur, and the smallest fresh buffer capacity ($M_f/M_h = 0.3$) always led to the best performance.

The results of these experiments are shown on figure 2. The left chart shows results for the memory constraint $m = 0.001$, the middle chart for $m = 0.01$ and the right chart for $m = 0.05$. Given a value of m, a threshold ϑ and a value of M_f/M_h, the charts report the best performance that could be achieved in those condition by optimally choosing the M_m/M_h ratio. The charts show performance as measured by AUC, but the results for F_1 and for the threshold-independent FSR measure all point to the same conclusion: the smallest fresh buffer, $M_f/M_h = 0.3$, always gave the best results.

4.4.2 *Main vs. overflow table*

Based on the results of the previous section, we now fixed the fresh buffer capacity at $M_f/M_h = 0.3$. The remaining memory is occupied by the hash table and must be divided between the main table and the overflow table. This again opens up the opportunity for a kind of tradeoff. A smaller main table (and hence larger overflow table) decreases the chances of problems with the removal of items (see sec. 3.4), but it also means longer chains and hence slower operations on the hash table. A larger main table (and smaller overflow table) means shorter chains and thus faster hash table lookups, but there may be fewer items suitable for deletion, requiring more random probes before an item gets deleted and hence slowing down the operation of the algorithm as a whole.

The results of these experiments are shown on figure 3. The left chart shows results for the memory constraint $m = 0.001$, the middle chart for $m = 0.01$ and the right chart for $m = 0.05$. Generally, if the total amount of available memory was scarce relative to the number of frequent items (thus when $m = 0.001$, or when $m = 0.01$ but the threshold ϑ was low, e.g. 5 or 10), a balanced hash table ($M_m/M_h = 0.5$) gave the best results; when memory was more plentiful, a smaller main table ($M_m/M_h = 0.3$) worked better. However, the differences in performance due to varying values of M_m/M_h were generally quite small. The charts on figure 3 show performance as measured by AUC, but looking at F_1 or FSR leads to the same conclusions.

4.5 SpaceSaving results

The SpaceSaving algorithm does not have a tunable parameter analogous to our M_f/M_h, because it requires all items that are present in the hash table to also be stored in the heap (unlike in the case of our fresh buffer). Thus, the total memory limit m by itself already determines the maximum number of items that we can store. We can, however, still tune the M_m/M_h parameter, the size of the main table relative to the entire hash table.

The results of these experiments are shown on figure 4. The left chart shows results for the memory constraint $m = 0.001$, the middle chart for $m = 0.01$ and the right chart for $m = 0.05$. The best performance was consistently achieved when the main table was the smallest ($M_m/M_h = 0.3$), both in terms of AUC, F_1 and FSR (the only exception being FSR for $m = 0.05$, where larger sizes of the main table did better). At the same time, having a smaller main table also consistently slowed down the algorithm;

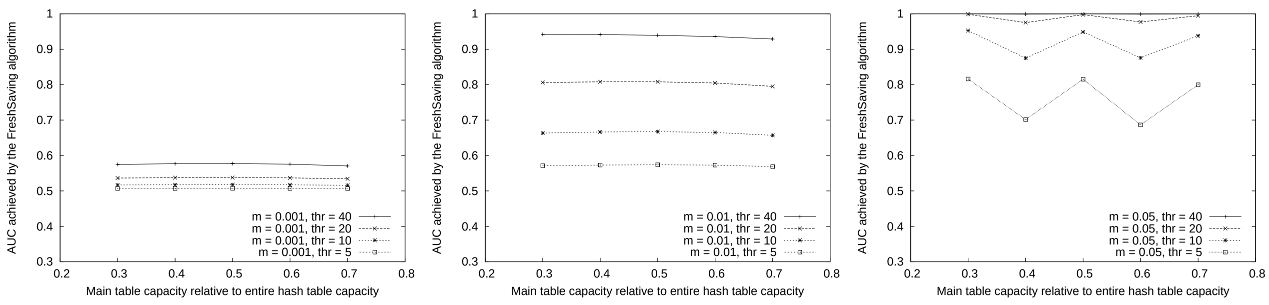


Figure 3: Influence of the main table vs. overflow table capacity ratio on the performance of the FreshSaving algorithm. The fresh buffer capacity has been fixed to 30 % of the total hash table capacity.

Table 1: Comparison of the performance (FSR) and execution time of the FreshSaving (FS) and SpaceSaving (SS) algorithms.

Memory constraint	Performance (FSR measure)		Processing time (sec.)	
m	FS	SS	FS	SS
0.001	0.8592	0.7220	13029	3123
0.01	0.8841	0.7711	15227	3558
0.05	0.8936	0.7890	20838	3891

the processing time at $M_m/M_h = 0.3$ was about 20 % longer than at 0.5 and about 35–40 % longer than at 0.7.

4.6 Comparison

Fig. 5 shows a comparison of the performance of SPACESAVING (SS) and FRESHSAVING (FS), for various values of the memory constraint m and the threshold ϑ. The left chart shows the area under the ROC curve, while the right chart shows the F_1-measure. As we can see, FS never performs worse than SS (according to both measures), and is in fact significantly better than SS most of the time. Given enough memory relative to the number of frequent items (e.g. when $m = 0.05$ and the threshold is high, $\vartheta = 40$), both algorithms can achieve near-perfect results.

Table 1 also shows the performance of both algorithms as evaluated by the FSR (frequency sum ratio) measure, which does not depend on a threshold ϑ. We can see that according to this measure, FRESHSAVING found a set of items that are on the whole considerably more frequent than those found by SPACESAVING. The table, however, also shows an important drawback of FRESHSAVING: it is much slower and requires about 4–5 times as much time as SPACESAVING to process the same amount of data.

5. RELATED WORK

The frequent items problem was first discussed by Boyer and Moore in the early 1980s. [6] is a recent survey of the area. Frequent items algorithms can be broadly categorized into counter-based and sketch-based. Counter-based algorithms maintain a set of items, along with a counter for each item, which contains information about that item's frequency. They differ as to when they add or remove items from memory and how they update the counters. Examples of counter-based algoritms are MAJORITY [17], FREQUENT [15, 9], LOSSY COUNTING [16] and SPACESAVING [14]. Sketch-based algorithms work by maintaining a low-dimensional projection of the vector of item frequencies; they include COUNTSKETCH [3], HCOUNT [13] and COUNTMIN [7]. There is also much work on slightly different but closely related problems, such as frequency moment estimation [2], entropy estimation [4] and maintenance of statistics over sliding windows [8, 11]. Recently some of that work has also been extended to probabilistic streams [12].

6. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed the frequent items problem — how to find frequent or popular items in streams of data. We showed that most of the streams have certain characteristics that can be exploited to achieve better results. We then developed the FRESHSAVING algorithm, which makes use of these characteristics to find frequent items. With this algorithm, we are able to find most of the frequent items and give their almost exact frequency by using an amount of main memory that is only a few percent of the size of the entire input stream. We also developed a framework for finding or detecting popular n-grams over the Internet and providing up to minute analysis of when an n-gram becomes frequent. We extensively evaluated the FRESHSAVING and SPACESAVING algorithms on a large real-world data stream (the RCV1 corpus, consisting of Reuters news stories from 1996–7) and showed that our algorithm outperforms SPACESAVING according to several evaluation measures.

There are a number of interesting directions for possible future work. By using more complex memory allocation techniques in the overflow table, it might be possible to further reduce the memory overhead due to the *next* pointers/indices in the entries that form linked lists. The role of the fresh buffer capacity could also be investigated further, in particular how much it may be shrunk before performance begins to deteriorate, and how this interacts with the characteristics of the input stream. Experiments on permutated versions of real-world sequences could help us understand to what extent our algorithm depends on the locality of item occurrences. Another possibility is to attempt a theoretical analysis of our algorithm and try coming up with bounds on the errors of its frequency estimates.

We are convinced that the methodology and the algorithm introduced in this paper can apply to many other applications, such as detecting frequent queries posed to an Internet search engine, finding frequent transactions across the branches of a supermarket chain or detecting popular text topic or text trends on the Internet. Data streams also arise in domains like highspeed networking, finance, and transaction logs [1]. In the future we intend to apply our algorithm to several other large textual data streams, including the stream of blog posts and news articles from Spinn3r (a large-scale content aggregator) and the stream of Twitter messages.

As mentioned before, our main motivation for developing the FRESHSAVING algorithm was finding popular n-grams over the Internet. Being able to find the frequent n-grams at any moment is just the first step towards the bigger picture. The next step would be to connect the framework to the stream of news, blogs and feeds, in order to get a live stream of what is being written on the Internet.

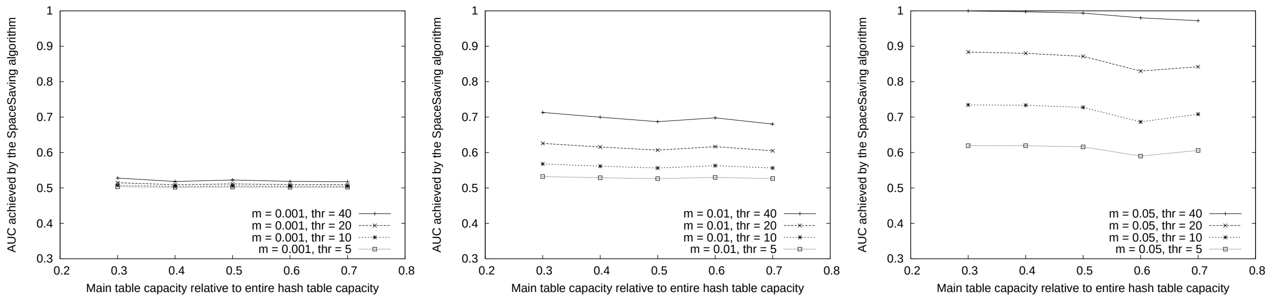


Figure 4: Influence of the main table vs. overflow table capacity ratio on the performance of the SpaceSaving algorithm.

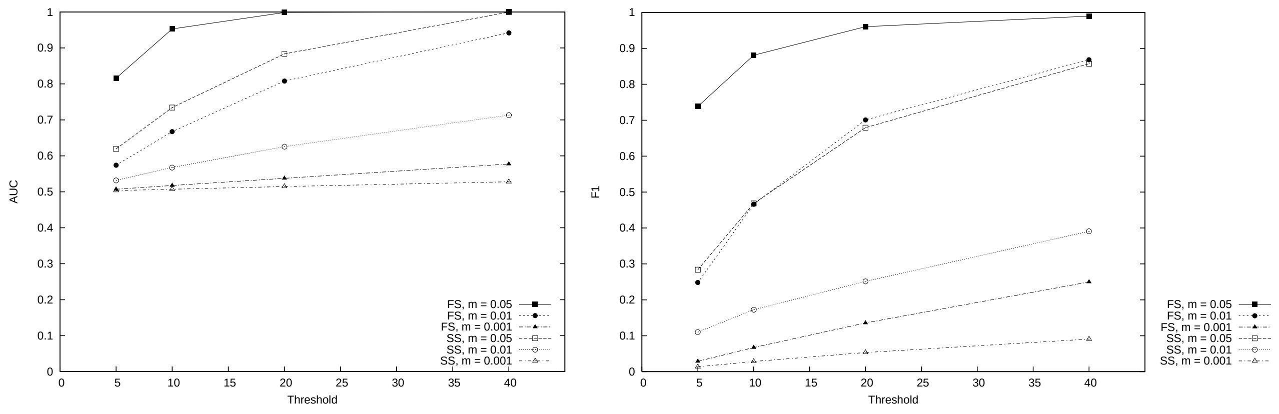


Figure 5: Comparison of the performance of the SpaceSaving and FreshSaving algorithms, for various memory constraints m and minimum frequency thresholds. The left chart shows AUC, the right one shows the F_1-measure.

To each item, we would add two additional features: the source of the content (New York Times, Bloomberg, etc.) and the list of times when we encountered the item. Now, we could plot a graph and see how the frequent n-grams evolve through the time. In this context, n-grams would be more than just sequences of n words, they would be popular phrases or even topics on the Internet. Having a list of sources where each frequent n-grams came from would tell us when did each source start writing about an n-gram or topic and how that topic evolved through time.

7. REFERENCES

[1] A. Arasu, G. S. Manku. Approximate counts and quantiles over sliding windows. *Proc. 23rd PODS*, 2004, pp. 286–296.

[2] N. Alon, Y. Matias, M. Szegedy. The space complexity of approximating the frequency moments. *STOC 1996*, pp. 20–29.

[3] M. Charikar, K. Chen, M. Farach-Colton. Finding frequent items in data streams. *Proc. 29th Int. C. Automata, Languages and Programming*, 2002. LNCS vol. 2380, pp. 693–703.

[4] A. Chakrabarti, G. Cormode, A. McGregor. A near-optimal algorithm for computing the entropy of a stream. *Proc. SDA 2007*, pp. 328–335.

[5] K. Chen, S. Rao. *An improved frequent items algorithm with applications to web caching*. UC Berkeley Tech Report, UCB-CS-05-1383, 2005.

[6] G. Cormode, M. Hadjieleftheriou. Finding the frequent items in streams of data. *CACM*, 52(10):97–105, 2009.

[7] G. Cormode, S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55(1):58–72, April 2005.

[8] M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining stream statistics over sliding windows. *Proc. 13th Symp. on Disc. Alg.*, 2002, pp. 635–644.

[9] E. D. Demaine, A. López-Ortiz, J. I. Munro. Frequency estimation of internet packet streams with limited space. *Proc. Eur. Symp. on Alg.*, 2002, pp. 348–360.

[10] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006.

[11] P. B. Gibbons, S. Tirthapura. Distributed streams algorithms for sliding windows. *Proc. SPAA 2002*, pp. 63–72.

[12] T. S. Jayram, A. McGregor, S. Muthukrishnan, E. Vee:. Estimating statistical aggregates on probabilistic data streams. *Proc. 26th PODS*, 2007, pp. 243–252.

[13] C. Jin, W. Qian, C. Sha, J. X. Yu, A. Zhou. Dynamically maintaining frequent items over a data stream. *Proc. CIKM 2003*, pp. 287–294.

[14] A. Metwally, D. Agrawal, A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. *Proc. ICDT 2005*. LNCS vol. 3363, pp. 398–412.

[15] J. Misra, D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, November 1982.

[16] G. S. Manku, R. Motwani. Approximate frequency counts over data streams. *Proc. 28th VLDB*, 2002, pp. 346–357.

[17] M. J. Fischer, S. L. Salzberg. Finding a majority among n votes: Solution to problem 81-5. *J. Alg.*, 3(4):362–380, 1982.

[18] D. D. Lewis, Y. Yang, T. Rose, F. Li. RCV1: A New Benchmark Collection for Text Categorization Research. *JMLR*, 2004.