Efficiently Discovering Recent Frequent Items In Data Streams *

Ferry Irawan Tantono[1], Nishad Manerikar[1], and Themis Palpanas[1]

University of Trento

Abstract. The problem of frequent item discovery in streaming data has attracted a lot of attention lately. While the above problem has been studied extensively, and several techniques have been proposed for its solution, these approaches treat all the values of the data stream equally. Nevertheless, not all values are of equal importance. In several situations, we are interested more in the new values that have appeared in the stream, rather than in the older ones.
In this paper, we address the problem of finding *recent* frequent items in a data stream given a small bounded memory, and present novel algorithms to this direction. We propose a basic algorithm that extends the functionality of existing approaches by monitoring item frequencies in recent windows. Subsequently, we present an improved version of the algorithm with significantly improved performance (in terms of accuracy), at no extra memory cost. Finally, we perform an extensive experimental evaluation, and show that the proposed algorithms can efficiently identify the frequent items in ad hoc recent windows of a data stream.

1 Introduction

The problem of frequent item discovery in streaming data has attracted much attention, because it is relevant to many different applications across various domains [13, 15, 12]. A naive approach to deal with this problem is to keep a count of each distinct item. Yet, in general, we assume that our main memory is not large enough to hold counters for all the distinct items. Several techniques that can efficiently solve the problem have been proposed in the literature that also take into account the special characteristics and requirements of streaming data [23, 10, 17]. These techniques are approximate, but they can provide the correct answer with high probability and they have been empirically proven to produce accurate results.

The above approaches treat all the values of the data stream equally. Note though, that not all the values that have appeared in the data stream are of equal importance. In several situations, we are more interested in the values that have appeared in the stream in the recent past, rather than in the distant past.

* This work was partially supported by the FP7 EU Large-scale Integrating Project **OKKAM – Enabling a Web of Entities** (contract no. ICT-215032). For more details, visit http://www.okkam.org.

Similar observations have also been made in other works, where the problems of time-variant data summarization [25, 5], clustering [3], and storage [8] have been studied.

The same is true for the problem of frequent item identification in data streams. A few indicative examples are described below.

- In the financial domain, we are interested in finding stocks that are traded the most in a stock exchange system. This knowledge is crucial for applications that deal with automatic trading, pre-trade analysis, post-trade execution, and market monitoring [26].
- In the communications and network operators industry several applications need to monitor the frequency of occurrence of packets traveling between specific nodes in the network [12]. This information is in many cases at the core of the business of companies in this area.
- Retail shops and online businesses are interested in identifying the products that sell the most. The results of this analysis can be used for launching special promotions, performing inventory management, and in other applications [19].

The applications in the above examples require estimates in the item frequencies for the recent past, rather than for the entire history of the data stream. Moreover, in certain cases the users would like to be able to query about the item frequencies in different windows in the recent past, and compare these values among themselves.

In this paper, we propose solutions for the discovery of *recent* frequent items in streaming data given a small bounded memory. These solutions are based on existing *sketching* techniques, which we extend in order to be able to effectively operate on the recent past. We describe the *TiTiCount*[1] algorithm that can be used to efficiently answer queries for frequent items in ad hoc recent windows. The algorithm uses a tilted timeframe for the representation of the past, which allows the algorithm to provide item frequency estimates for a number of different windows in the past, using a small amount of memory. At the same time, these estimates are more accurate for the most resent windows, and the accuracy of the estimates diminishes as we go further in the past. We also present a query answering method that takes into account the size of the window intervals used by our algorithm, and provides better frequency estimates than the straightforward approach.

Furthermore, we propose *TiTiCount+*, an enhanced algorithm for query answering. In this case, when a query for some item frequency in a particular window comes in, the query answering algorithm makes use of the information stored in the specified window of interest, but also uses the information stored in certain neighboring windows. Based on this extra information, the algorithm is able to refine the item frequency estimates, leading to more accurate results, with minimal additional processing. As we will describe in more detail later on,

[1] Tilted Timeframe Count.

this scheme also leads to superior performance in the case where the distribution of the data stream is non-stationary.

In summary, in this work we make the following contributions.

– We describe algorithms that can estimate the frequency counts of hot items in the recent past of a data stream. Our approach efficiently supports queries on ad hoc recent windows, and can store information about arbitrary points in the past, depending on the user preferences and available memory budget.
– We propose a simple method that accounts for the size of our summary structures, and leads to more accurate item frequency estimates in query answering when compared to the straight-forward approach.
– We extend the above algorithm with a technique that combines the information stored in different parts of our data representation structures in order to improve the accuracy of the results. As we empirically demonstrate, the above technique results in a significant performance improvement at a negligible additional processing cost.
– Finally, we perform an extensive experimental evaluation using synthetic and real data. The results show the behavior of the algorithms in different conditions, and demonstrate the effectiveness of the proposed approach.

The rest of the paper is organized as follows. We start by giving some necessary background for the problem of mining data streams for frequent items in Section 2. In Section 3, we describe the problem of recent frequent items formally. Section 4 describes the development of our algorithm on the basis of two existing algorithms. Our experimental evaluation is presented in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Background

We assume a data stream S that is composed of a stream of integer numbers, where each integer represents the occurrence of a data item in S.

Let N be the current length of the data stream S, i.e., N is the current number of transactions. Further assume that the data stream contains M distinct values. A *frequent item* is an item whose frequency is greater than ϕN, where the *support* parameter ϕ is a user-defined threshold in the interval $[0.0, 1.0]$.

Several algorithms have been proposed for efficiently mining frequent items in data streams. The Frequent (FREQ)[18] and the Lossy Counting (LC)[23] algorithms are based on maintaining approximate frequency counts, while Combinatorial Group Testing (CGT)[11], Count-Min (CM)[10], CCFC [7] and hCount (HC)[17] are based on *sketches*. We have conducted extensive experiments in order to compare the performance of these algorithms. Our implementation of the *FREQ*, *LC*, *CM*, *CGT*, and *CCFC* algorithms was based on the Massive Data Analysis Lab code-base [2]. The *hCount* algorithm was implemented from scratch, using the same optimizations as the other algorithms. We ran experiments on both synthetic and real datasets, and measured time and space usage for all the above six algorithms, averaged over several independent runs. In order to evaluate the quality of the results obtained, we used the two standard measures of *recall* (percentage of the true frequent items that are found by the algorithm) and *precision* (percentage of items identified by the algorithm, which are truly frequent).

In the interest of space, we only briefly summarize our results (details are in the full version of this paper). We ran experiments with varying the support threshold ϕ. The results indicate that the performance of *CCFC* is affected when support is low, but its recall improves when the support level is high. Regarding precision, *hCount* and *CCFC* are consistently the top performers, with the other algorithms improving their performance as the support threshold is increased.

We also measured the scalability and time requirements of the algorithms by running experiments with 10 to 100 million transactions. The results show that all algorithms scale linearly in time with respect to the number of transactions. *CCFC* requires the longest time, whereas *LC* and *FREQ* are the most time-efficient, with *hCount* performing very close to the fastest algorithms.

The qualitative results from all our experiments are summarized in Table 1 (a more detailed discussion of the experiments can be found elsewhere [22]). Based on these experiments (similar results have also appeared elsewhere [17]), we selected the *hCount* algorithm as the frequency estimation component of our approach, because it has several desirable characteristics. Namely, it exhibits a consistently good performance across various conditions, it has low time complexity, and is relatively easy to implement.

Note that this choice is not restrictive in any way, and in our techniques *hCount* could be replaced with any other suitable frequency estimation algorithm.

Table 1. Performance Summary.

Algorithm	Characteristics
FREQ	Fast. Low precision.
CGT	Fastest of the sketch-based. Cannot handle lower support
CM	Less space than CGT but more time, cannot handle lower support
CCFC	Slow, fairly good accuracy
LC	Fast, good recall and precision
HC	Fast, good recall and precision

3 Recent Frequent Items

In this section, we formally define the problem of recent frequent item discovery, and we give a brief overview of our approach.

3.1 Problem Definition

Let the data stream S be represented by $\{T_1, T_2, \ldots, T_n\}$, where T_i denotes the i^{th} item, and T_n is the latest (most recent) item in the stream. In this work, we assume that each item, T_i, is represented by a single integer, and corresponds to a transaction [2].

Let $w = [w_{min}, w_{max}]$ define a window in the history of the stream, where w_{min} refers to the index of the least recent point in the window, and w_{max} to the index of the most recent one. The length, or size (in terms of number of transactions), of window w is $|w| = w_{max} - w_{min}$. Further, assume that ϕ, $0 < \phi \leq 1$ is a user-defined parameter that determines which items are frequent, according to the following definition.

Definition 1. [Frequent Item] *An item is called* frequent *with respect to a window w if it appears in at least $\phi|w|$ transactions within w.*

We can now define the recent frequent item problem for a stream S.

Problem 1. [Recent Frequent Item (RFI)] Given a threshold ϕ, and a window w, where $n - w_{min} \leq L$, we want to identify the frequent items in w, for a predetermined parameter $L >> 1$.

We make two remarks regarding the above definition of the problem. First, both the window w and the threshold ϕ are part of the query, and can be different for each query. Also note that the query window of interest w, is ad hoc, and can refer to any interval in the recent history of the stream. The parameter L determines how far in the past the query window can refer to. Essentially, L defines the least recent transaction that can be part of the query window, and in practice can be very large. That is, for a window $w = [w_{min}, w_{max}]$, $n - L \leq w_{min} < w_{max} \leq n$.

Second, we define the window size in terms of the number of transactions, rather than time, because the data rates of streams are often times variable. Hence, windows defined in terms of number of transactions are more appropriate. Nevertheless, the techniques we propose can in principle work for both cases.

3.2 Proposed Approach

Previous works have studied the problem of identifying frequent items in the entire history of a data stream [18, 23, 11, 10, 7, 17]. What is fundamentally different in our case is that we wish to identify frequent items in arbitrary (recent) window intervals of the stream. In order to solve the *RFI* problem, we have to store information about the item frequencies in various time-points in the past, which will allow us to answer queries for ad hoc windows.

A simple solution to the above problem is to divide the recent history of the stream, that is, the last L transactions, in fixed-size intervals, and estimate the

[2] For the remainder of this paper, we will use the terms *item* and *transaction* interchangeably.

item frequency counts for each one of these windows. This scheme allows us to answer queries even if they are not aligned to the interval boundaries; in this case, we provide an approximate answer.

However, the drawback of this approach is that the memory requirements are rather high. For $L >> 1$, we need to keep information on a large number of intervals. Note that the number of intervals is also directly related to the accuracy of the query answers we can provide. Therefore, reducing the memory requirements comes at the cost of performance.

In order to overcome the above limitation and efficiently solve the *RFI* problem, we propose the use of *tilted-time* window intervals. (Similar approaches have been studied in other applications as well [8, 3, 25, 5]. Though, as we describe later on, we propose novel operation schemes that allow our algorithms to offer significant performance improvements.) Under this scheme, we divide the history of the stream in increasingly larger intervals as we move in the past (resulting in more accurate item frequency estimations for the most recent window intervals, and increasingly less accurate for the window intervals further in the past). Therefore, we can significantly reduce the memory requirements, while still being able to answer queries from different time horizons. In the algorithms we propose, we assume *logarithmic* tilted-time windows, where each subsequent older window interval is twice the size of the previous interval. In this case, we can cover the entire space of L transactions with just $K = \log L$ windows.

In the following section, we describe algorithms that efficiently and effectively solve the *RFI* problem, using the tilted-time windows scheme. We also propose techniques that can significantly improve the accuracy of the algorithms using the same amount of memory. These improvements are more pronounced for queries involving the older, larger window intervals. Thus, we effectively alleviate the disadvantage that the tilted-time windows have on the intervals referring to stream values further in the past.

4 Algorithms for Recent Frequent Items

In this section, we present algorithms for the *RFI* problem. We start by briefly describing the main skeleton of the algorithms, which is the same for all of them. Subsequently, we discuss in more detail specific features of each algorithm, and the benefits it brings along.

As we mentioned earlier, we use the *hCount* sketch in order to estimate the frequency counts within a given window interval. The *hCount* algorithm [17] maintains an array of $M \times H$ counters, where M and H are parameters determined by the data characteristics and the allowed error. The algorithm uses H hash functions that map the occurrence of an item to H of the counters, which are subsequently incremented by one. The estimate of an item's frequency is computed as the minimum value of all the counters to which the item maps.

In our case, instead of a single window interval, the algorithm has to operate with K intervals. These windows follow a tilted time-frame as follows. The first window, w_0, covers the b most recent stream values, that is, transactions

$T_{n-b+1}, \ldots, T_n$. The parameter b defines the size of w_0, and is called *batch size*. The second window, w_1 is also of size b, and covers the next b transactions. Then, the size of each subsequent window is double the size of the previous one. In general, the size of the i-th window is given by the formula $w_i = 2^{i-1}b$, $0 < i < K$.

In order to account for all K window intervals, we extend the *hCount* sketch to an array of $M \times H \times K$ elements by replacing each one of the $M \times H$ counters $c_{m,h}$in the original structure with an array $c_{m,h}[]$ of K counters, for $0 \leq m < M$, $0 \leq h < H$ [3]. These arrays of counters correspond to the K windows, as shown in Figure 1. The first element (in some cases also the second, as we will

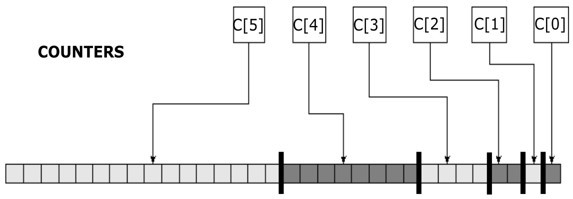

Fig. 1. Tilted-time windows.

explain later) of these arrays, $c[0]$, stores the counts for newly incoming stream values (according to the *hCount* algorithm). The subsequent elements store the historical values of the counts that refer to the corresponding window interval. In essence, they keep track of the history of the item frequencies.

There are two main operations that we need to have in place (outlined in Figure 2). First, the shifting of the counter values $c[]$ so that they correspond to the current window intervals. This operation is triggered every time the window that receives the new stream values gets full, that is, every b transactions. Second, the item frequency estimation mechanism, used to provide the estimate of an item frequency within a given window interval.

In the next sections, we describe in more detail different solutions that we propose for the above two operations.

4.1 Basic Algorithm

The straightforward approach to implement the shifting operation is to use intermediate windows (and corresponding counters). As shown in Figure 3, the counters corresponding to the first window, $c[0]$, are always receiving the new data (depicted in gray), and counter values shift sequentially every b transactions.

Answering item frequency questions in this model is simple as well. When a query for the frequency of an item in a specific window interval w_q comes in, we

[3] For the remainder of the text, we omit the indices m, h when we refer in general to the array of counters $c[]$.

```
Let $n$ :=current transaction number
    $b$ := batch size

When new transaction $T_n$ arrives:
1   use hCount to determine the set of counters $\mathcal{C}$ related to $T_n$
2   for each counter in $\mathcal{C}$
3       update the counts of $c$
4   if ($n$ mod $b$) == 0
5       call PerformShift()

When query for frequency of item $i$ in window interval [$t^q_{min}$,$t^q_{max}$] arrives:
6   use hCount to determine the set of counters $\mathcal{C}$ corresponding to $T_n$
7       call GetFreqEst($\mathcal{C}$,[$t^q_{min}$,$t^q_{max}$])
```

Fig. 2. Main skeleton of the proposed algorithms.

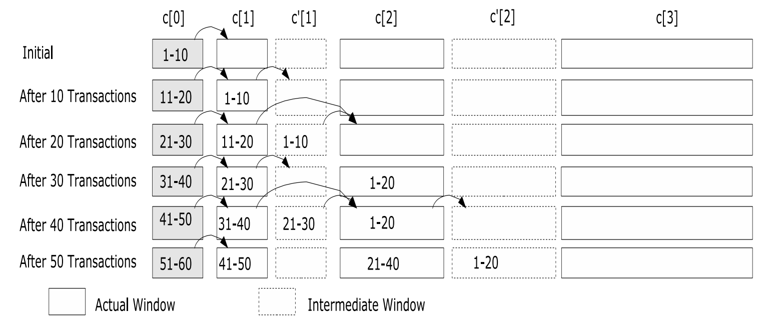

Fig. 3. Shifting with Intermediate Windows (batch size $b = 10$). Gray boxes denote windows receiving new stream values.

identify the counters that store information on time intervals overlapping with w_q, and we sum the estimates from these counters. Note that if the query interval w_q is not aligned with the counter time intervals, then we introduce errors in the estimation, since we are counting frequencies over intervals that do not belong in the w_q (this is true for the two ends of the query interval).

The advantage of this algorithm, which we call *NaiveCount*, is its simplicity. Though, this advantage comes at the expense of memory (for the intermediate windows). The required memory for *NaiveCount* is $(2K - 1)S$, where K is the number of windows and S is the memory required by *hCount* (or any other similar technique that can be used here), and the number of shift operations is in the worst case K. In the following sections, we show how we can reduce the memory requirements, while at the same time improving the accuracy of the results.

4.2 Reducing the Memory Requirements

We observe that we can reduce the memory requirements of the algorithm by discarding the intermediate windows. Under the new shifting scheme (see Figure 4), we keep track of which counters correspond to which window intervals, which allows us to directly move counter contents to the next window. (We also employ *lazy* shifting, by allowing also the second window to process new stream values, thus, only shifting contents when necessary and saving some shift operations.) In this case, the memory requirements are KS (K is the number of windows and S the size of the sketch), while the number of shift operations is in the worst case K.

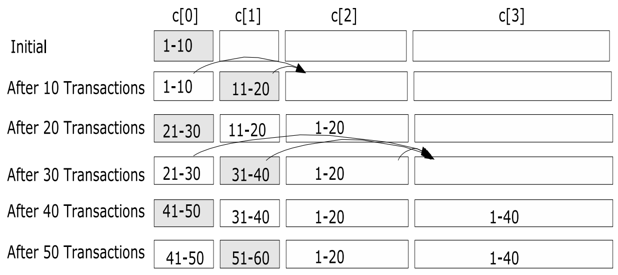

Fig. 4. Shifting without Intermediate Windows (batch size $b = 10$). Gray boxes denote windows receiving new stream values.

Even though this algorithm, *TiTiCount*, needs almost half the amount of memory of *NaiveCount* by not using any intermediate windows, the accuracy of its results is not affected. This is because the intermediate windows are only used to facilitate the shifting operation.

What is more interesting is that with *TiTiCount* we can actually improve the accuracy of the results. In *NaiveCount*, we notice that whenever the edges of the query window w_q are not aligned with the edges of the window intervals corresponding to the counters $c[]$, we introduce an error in the results. Consider query q with $w_q = [100, 950]$ of Figure 6. The edges of w_q are not aligned with the edges of w_4 and w_0. Nevertheless, for the calculation of the result *NaiveCount* will consider the counts corresponding to the entire intervals w_4 and w_0, even though part of them falls outside w_q.

TiTiCount resolves this problem, and only takes into account the portions of the windows that are covered by the query. This is achieved by considering in the result the *weighted fraction* of the estimate provided by the counters $c[]$ that corresponds to the fraction of the counter window overlapping the query window. In this work, we apply a linear model in this computation (i.e., the fraction is directly computed as the amount of overlap), but other, more sophisticated techniques can be applied (e.g., even limited knowledge on the distribution of the frequencies within a window could lead to an even more accurate non-linear model). However, as we show in the experimental evaluation of the algorithms, this simple idea improves the quality of the results substantially.

4.3 Exploiting Redundant Information

Taking a close look at the shifting operation, we observe that during specific time intervals, information pertaining to the same data stream transactions is stored in more than one counters at the same time. For example, referring back to Figure 4 (example of shifting for *TiTiCount*), we observe that information regarding transactions $1-20$ is stored both in $c[2]$ and $c[3]$ (see bottom of the figure). Note that the counters do not store the same information, as $c[3]$ corresponds to a larger time interval than $c[2]$. Nevertheless, there is a certain amount of information redundancy, and in the following paragraphs we explain how the *TiTiCount+* algorithm uses it in order to further improve the accuracy of the results.

In order to exploit the above side-effect of shifting, we modify the shifting operation as follows. We no longer employ the lazy shifting scheme used by *TiTiCount*, but instead have the first window process all the new data stream transactions. This results in an increased number of shift operations, which in the worst case can be as many as $K(K-1)/2$. However, the required shifts are on the average much less, and as we empirically demonstrate, the additional cost in the total running time is very small. The memory requirements are the same as before, namely, KS.

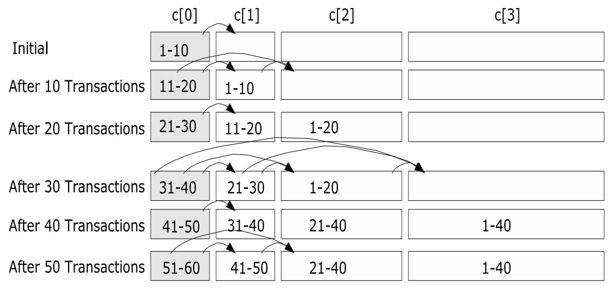

Fig. 5. Value added shifting (batch size $b = 10$). Gray boxes denote windows receiving new stream values.

When we apply the above shifting mechanism, the way the various window intervals are placed with respect to each other is governed by the following properties [4].

Lemma 1. [Window Property 1] *If two window intervals overlap, then the smaller window interval is completely contained in the larger one.*

Lemma 2. [Window Property 2] *All window intervals that overlap have one common boundary, and this common boundary is the most recent edge of these intervals.*

[4] In the interest of space, we omit the proofs, which can be found in the full version of this paper.

The above properties are very important, because they constitute the base of the query answering algorithm. The main idea of the algorithm is to always use the counters corresponding to the *smallest* possible window interval in order to estimate some frequency count. When a query w_q comes in, it is split into subqueries $w_{sq_1}, \ldots, w_{sq_j}, \ldots, w_{sq_J}$ that align with the boundaries of the counter window intervals. Then, results for each subquery are derived as follows.

- *Smallest Interval:* If w_{sq_j} can be answered using multiple counters then use the counter that corresponds to the smallest window interval to compute the result.
- *Subtraction Operation:* If the window interval, w_l, of the counter that is to be used to answer w_{sq_j} overlaps with a smaller window interval, w_s, then subtract the values of the w_s counter from the w_l counter, and subsequently compute the result.

The above steps lead to correct results, because the properties stated in Lemmata 1 and 2 ensure the window intervals are aligned in such a way that the subtraction operation is feasible. The following example explains how this algorithm works.

Example 1. Assume we have five window intervals, $w_0, \ldots, w_4$, and that the current transaction number is 990, as shown in Figure 6. A query q comes in, asking for frequent items in interval $[100, 950]$. The algorithm splits q in five subqueries, according to the boundaries of the window intervals with which it overlaps. Then, the algorithm computes frequency estimates for each subquery as follows. The estimate for q_5 is derived from the w_0 counter by applying the weighted fraction model (i.e., the estimate will be $(950-901+1)/(990-901+1)$ times the result returned by the counter). Frequency estimates for q_4 and q_3 are derived directly from the counters of intervals w_1 and w_2, respectively. For q_2, the estimate is directly computed after subtracting the values of the w_2 counter from the w_3 counter. Finally, for q_1 the algorithm first subtracts the values of the w_3 counter from the w_4 counter, and then applied the weighted fraction model, since q_1 is not interested in the first 100 transactions of w_4.

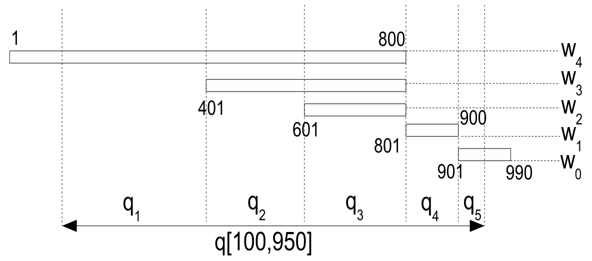

Fig. 6. Example of query answering for *TiTiCount+*.

We can now demonstrate the advantage that the subtraction operation provides to *TiTiCount+* for producing estimates with significantly improved accuracy, when compared to *TiTiCount*. Using the same example as above, assume that all the items in the interval $[1, 400]$ have the value x, and all the items in the interval $[401, 990]$ have the value y. Suppose, a query comes that asks for the frequency of value y in interval $[1, 400]$. In this case, *TiTiCount* will use just the w_4 counter with the weighted fraction model, returning an answer of 200. On the other hand, *TiTiCount+* will subtract the contents of the w_3 counter from those of the w_4 counter, and correctly return 0 as an answer. Evidently, this advantage of *TiTiCount+* is magnified when the distribution of the values in the data stream change over time.

5 Experimental Evaluation

We implemented our proposal and conducted a series of experiments to evaluate the efficiency of our techniques in a variety of settings. Apart from the three algorithms we describe in this paper, we also implemented algorithm *Linear* to compare against our approach. *Linear* is similar to *TiTiCount*, except that instead of tilted time window intervals, it uses window intervals of fixed size.

In our experiments we used both synthetic and real datasets. The synthetic datasets we used were generated according to a Zipfian distribution with Zipf parameter 1.1, unless noted otherwise. We generated datasets with up to 100 million items, with both stationary and non-stationary distributions. The real datasets we used were as follows.

- kosarak [1]: It consists of anonymized click-stream data of a Hungarian online news portal, expressed as a sets of integers. It has about 8 million individual items.
- retail [4]: It contains retail market basket data from an anonymous Belgian store. This dataset has about 0.9 million individual items.

We implemented all our algorithms in C using the gcc compiler under Linux Fedora Core 5. The experiments were run on a dual Intel Xeon 2.8Ghz machine.

5.1 Evaluating the Accuracy

In the first experiment, we compare the algorithms *NaiveCount*, *TiTiCount*, and *TiTiCount+* in terms of the accuracy of the results they provide. We measure recall, defined as the percentage of the true frequent items that are found by the algorithm, and precision, defined as the percentage of items identified by the algorithm that are truly frequent. We ran experiments using several query window intervals, where in each interval we were looking for the frequent items ($\phi = 0.005$). In Figure 7, we report the results for nine of these queries (the results for the rest of the queries we tried were similar). The queries we used as test cases are listed in Table 2 (we report the boundaries of the query window

Table 2. Query window intervals used as test cases.

n=50000			n=60000			n=70000		
No.	t^q_{min}	t^q_{max}	No.	t^q_{min}	t^q_{max}	No.	t^q_{min}	t^q_{max}
1	5000	45000	4	5000	55000	7	20000	45000
2	35000	45000	5	35000	55000	8	40000	55000
3	25000	40000	6	5000	50000	9	40000	65000

intervals). All experiments used a batch size $b = 1,000$, they were repeated 15 times, and results were averaged.

Figures 7(a) and 7(b) show the recall and precision for the three algorithms, when run over a dataset with a stationary distribution. We observe that all three algorithms have virtually perfect recall rates. However, precision varies. *TiTiCount* and *TiTiCount+* average precision rates close to 90%, with *TiTiCount+* performing slightly better. The performance of *NaiveCount* is notably worse, averaging a mere 45%.

In Figures 7(c) and 7(d), we show the results of the same experiment, when run over a dataset with time-variant distribution. In this case, the stream was generated by concatenating several small datasets. These datasets were all generated by sampling a Zipfian distribution, but each one of them had a different set of frequent items.

These experiments represent a more challenging setting for our algorithms, and the results demonstrate the qualitative difference among them. *TiTiCount+* is consistently the best performer among the three, with significantly better performance than *TiTiCount* in several cases. The *NaiveCount* algorithm performs very poorly in terms of precision, which explains its high recall rates.

The reason *TiTiCount+* produces even more accurate results than *TiTiCount* for the time-varying dataset is because the *TiTiCount* algorithm relies solely on the weighted fraction mechanism to arrive at frequency estimates. Evn though this is an improvement over the *NaiveCount* algorithm, this mechanism works well only for stationary distributions, where the item frequencies remain relatively stable across different window intervals. In contrast, *TiTiCount+* using the subtraction mechanism can effectively alleviate this problem and produce better estimates. This explains the large difference in performance observed in test cases $7-9$.

We also performed tests by varying the skew parameter of the Zipfian distribution. The trends in these experiments are similar, and we omit them for brevity. For the remainder of the discussion, we do not consider the *NaiveCount* algorithm.

In the following experiment, we tested the performance of the algorithms as a function of the size of the query window interval, and we also compare them to *Linear*. We use *Linear* only as an indication of how good the performance of our algorithms would be if they had enough memory to use fixed- instead of tilted-time window intervals. For our experiment, batch size $b = 1,000$, and number of windows $K = 11$. This means that our algorithms can answer queries

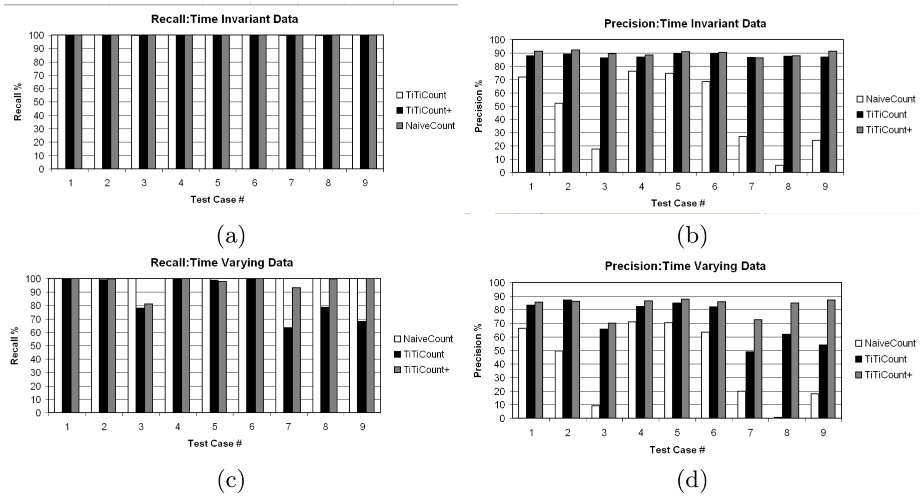

Fig. 7. Performance on time varying and non-varying data distributions.

about item frequencies for the past $1,000,000$ transactions. In order for *Linear* to be able to answer the same class of queries, we have to use 1000 windows (for window size equal to b), which requires two orders of magnitude more space than our algorithms. We also compared against *LinearConst*, which the *Linear* algorithm that is given the same amount of space as our algorithms (resulting in a window size of $100,000$).

The experiment was run on the *kosarak* dataset, using 120 randomly generated queries following a Gaussian distribution (mean $9N/10$, stddev $N/8$). Figures 8(a) and 8(b) depict the results of the experiment for recall and precision, respectively. The graphs show that *TiTiCount+* outperforms *TiTiCount* across the entire range of query sizes. It is interesting to note that while *TiTiCount* exhibits a steady recall rate across the experiment, *TiTiCount+* improves its performance as the size of the queries increase. This happens because larger queries are more effectively managed by the subtraction mechanism of *TiTiCount+*.

As expected, *LinearConst* performs the worst (its somewhat high precision numbers are explained by the low performance in recall), and *Linear* is almost always the winner in both metrics. Note though, that the performance of *TiTiCount+* is very close to *Linear*, which demonstrates the effectiveness of the subtraction mechanism.

5.2 Finding Top-k Items

In some situations, it is desirable to know the top-k most frequent items in a stream, or their cumulative frequency. Our algorithms can be adapted to determine those values. In this experiment, we tested *TiTiCount+* for the accuracy

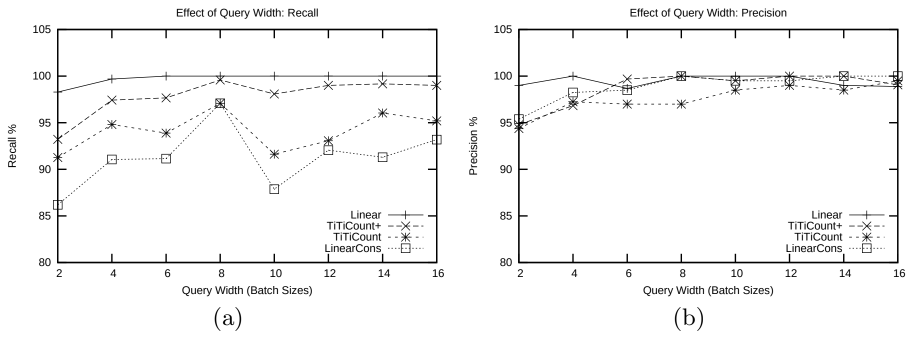

Fig. 8. Performance with respect to query width (dataset: kosarak, batch size $b = 1,000$).

of the estimated frequencies of the top-k items, and compared its results to the exact answers.

Similar to the previous experiment, we ran random queries of different sizes, asking for the cumulative frequencies of the top-k items, for several values of k. The results are illustrated in Figure 9, for both real datasets. The top-k items were correctly identified in all cases. The graphs show that the cumulative frequencies reported by *TiTiCount+* were consistently very accurate (less than 0.05% error for our experiments).

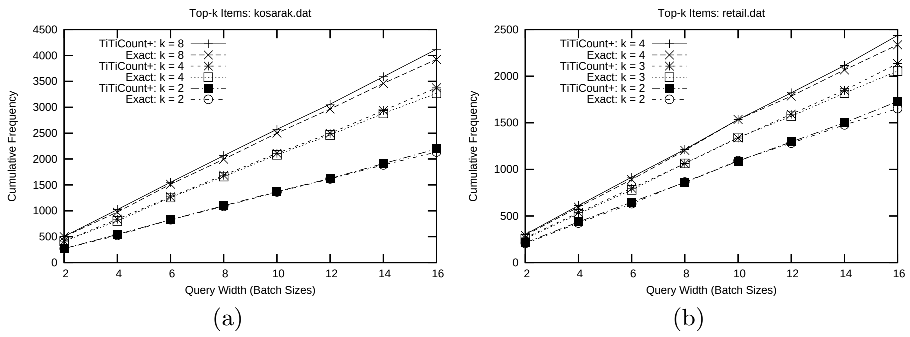

Fig. 9. Top-k items: Estimated and actual cumulative frequencies.

5.3 Scalability

In order to evaluate the scalability of the proposed algorithms, we ran experiments to measure the update times of *TiTiCount* and *TiTiCount+*. The update time is the time required to update the internal data structures every time a new transaction arrives, including shifting operations We tested the algorithms with data streams of 100 million transactions, and we report the cumulative update time in Figure 10. The reported times are averages over five independent runs. The results show that both algorithms scale linearly with the number of transactions, with *TiTiCount+* being slightly less efficient, because of the higher worst case cost of the shift operation that it implements.

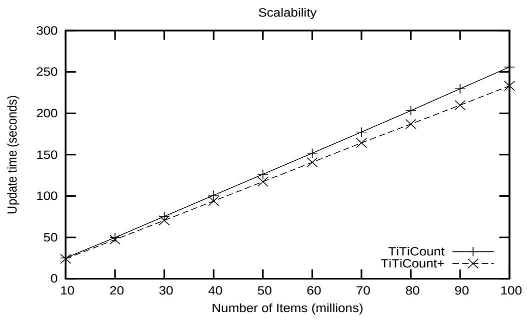

Fig. 10. Scalability: Variation in update time with increasing number of data items.

6 Related Work

In the recent years, numerous studies have focused on problems related to streaming data, ranging from practical applications to theoretical questions [16, 24]. There is a wealth of work on the problem of identifying frequent items in streaming data. The Frequent (FREQ)[18] and the Lossy Counting (LC)[23] algorithms maintain a number of counts, which are pruned as new items arrive in the data stream. Other algorithms, such as Combinatorial Group Testing (CGT) [11], Count-Min (CM)[10], CCFC [7] and hCount (HC) [17] are based on *sketches*. The sketches are designed so that they provide accurate results for the frequent item discovery problem, while requiring limited memory resources.

The important difference between these works and our approach is that we want to be more flexible in identifying frequent items, placing more importance on recent frequent items. For mining recent frequent items, an intuitive approach is to use time-decaying approximations. This technique has been used in several diverse areas, such as online time series summarization [25, 5], streaming data clustering [3], and data warehousing [8].

Other works have studied the problem of efficiently identifying and maintaining frequent itemsets over streaming data [6, 9, 20]. In this case, we are interested in *sets* of items that appear frequently together. Specialized techniques and algorithms have been developed for the solution of this problem. Some of these works are also based on sliding windows [21], or tilted time windows [14], in order to focus on the transactions in the recent past of the data stream. The *FP-Stream* approach [14] uses a tilted timeframe similar to our work. However it makes use of the *FP-Tree* structure, which has been specifically designed for itemsets, rather than items. An efficient implementation of the above approach for the problem we solve in this study is not straightforward. Moreover, in our work we describe novel shifting schemes for the tilted timeframe, which are used by *TiTiCount+* in order to deliver significant performance improvements.

7 Conclusions

The problem of frequent item identification has attracted lots of attention in the past years, and has found many interesting applications across diverse domains. This work is motivated by the need of many real-world applications to identify frequent items in the *recent* past of a data stream, rather than over the entire history.

In this paper, we propose novel algorithms for the discovery of *recent* frequent items in a data stream. The proposed algorithms are based on the *sketching* technique, and are very flexible in that they are designed to answer queries for frequent items in ad hoc window intervals in the recent past of the data stream. Based on our observations, we also describe extensions of the basic algorithm that can significantly improve the accuracy of the query results, while maintaining the same memory usage and at negligible additional processing cost.

We have evaluated the performance of the proposed techniques on real and synthetic data streams. The results show that the algorithms can efficiently operate using few space and time resources, while maintaining a high quality approximation in query answering.

Acknowledgments. The authors would like to thank the anonymous referees for their constructive criticism and comments.

References

1. Frequent itemset mining dataset repository, university of helsinki. http://fimi.cs.helsinki.fi/data/, 2008.
2. Massive data analysis lab, rutgers university. http://www.cs.rutgers.edu/~muthu/massdal.html, 2008.
3. C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.
4. T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.
5. A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *ICDE*, pages 303–314, 2003.
6. C.-H. Chang and S.-H. Yang. Enhancing swf for incremental association mining by itemset maintenance. In *PAKDD*, pages 301–312, 2003.

7. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 693–703, London, UK, 2002. Springer-Verlag.
8. Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, pages 323–334, 2002.
9. D. W.-L. Cheung, J. Han, V. T. Y. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *ICDE*, pages 106–114, 1996.
10. G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
11. G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.
12. C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, pages 323–336, 2002.
13. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1998.
14. C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. Mining frequent patterns in data streams at multiple time granularities. In *NSF Workshop on Next Generation Data Mining*, 2003.
15. P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1999.
16. A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, pages 79–88, 2001.
17. C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 287–294, New York, NY, USA, 2003. ACM Press.
18. R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
19. R. Kohavi and F. J. Provost. Applications of data mining to electronic commerce. *Data Min. Knowl. Discov.*, 5(1/2):5–10, 2001.
20. C.-H. Lee, C.-R. Lin, and M.-S. Chen. Sliding window filtering: an efficient method for incremental mining on a time-variant database. *Inf. Syst.*, 30(3):227–244, 2005.
21. C.-H. Lin, D.-Y. Chiu, Y.-H. Wu, and A. L. P. Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *SDM*, 2005.
22. N. Manerikar and T. Palpanas. Frequent Items in Streaming Data: An Experimental Evaluation of the State-of-the-Art. Technical Report DISI-08-017, University of Trento, Mar. 2008.
23. G. S. Manku and R. Motwani. Approximate frequency counts over data streams, 2002.
24. S. Muthukrishnan. Data streams: algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.
25. T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, pages 338–349, 2004.
26. A. T. Whitney and D. Shasha. Lots o' ticks: Real-time high performance time series queries on billions of trades and quotes. In *SIGMOD Conference*, page 617, 2001.