Data & Knowledge Engineering xxx (2013) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Identifying streaming frequent items in *ad hoc* time windows

Michele Dallachiesa *, Themis Palpanas

University of Trento, Italy

ARTICLE INFO

Article history:
Received 26 September 2011
Received in revised form 10 May 2013
Accepted 10 May 2013
Available online xxxx

Keywords:
Database management
Data mining
Business intelligence

ABSTRACT

The problem of frequent item discovery in streaming data has attracted a lot of attention, mainly because of its numerous applications in diverse domains, such as network traffic monitoring and e-business transactions analysis.

While the above problem has been studied extensively, and several techniques have been proposed for its solution, these approaches are geared towards the recent values in the stream. Nevertheless, in several situations the users would like to be able to query about the item frequencies in *ad hoc* windows in the stream history, and compare these values among themselves.

In this paper, we address the problem of finding frequent items in *ad hoc* windows in a data stream given a small bounded memory, and present novel algorithms to this direction. We propose basic *sketch*- and *count*-based algorithms that extend the functionality of existing approaches by monitoring item frequencies in the stream. Subsequently, we present an improved version of the algorithm with significantly better performance (in terms of accuracy, at no extra memory cost). Moreover, we propose an efficient non-linear model to better estimate the frequencies within the query windows.

Finally, we conduct an extensive experimental evaluation with synthetic and real datasets, which demonstrates the merits of the proposed solutions and provides guidelines for the practitioners in the field.

1. Introduction

The large growth in produced data volumes and the increase in network bandwidth seen in recent years have made it necessary to revisit conventional problems in the data mining field in the context of these advances. The data being mined are now often in the form of streaming data [3,35,6], and an important problem in this area is that of detecting frequent items in a data stream. The problem of frequent item discovery in streaming data has attracted much attention, because it is relevant to many different applications across various domains [18,20,17].

A naive approach to deal with this problem is to keep a count of each distinct item. However, when dealing with streaming data, the number of distinct items is very large (and in some cases potentially unbounded) and so this approach is infeasible in terms of memory and time requirements. Furthermore, in a streaming data environment, the algorithm can only look at the data once, thus preprocessing or making multiple passes over the data are not possible. 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 [33,14,24,34]. 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 drawback of these approaches is that they operate exclusively on all the values seen in the data stream so far, or otherwise on the values within a sliding window.

* Corresponding author. Tel.: +39 0461283908.
E-mail address: dallachiesa@disi.unitn.it (M. Dallachiesa).

0169-023X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.datak.2013.05.007

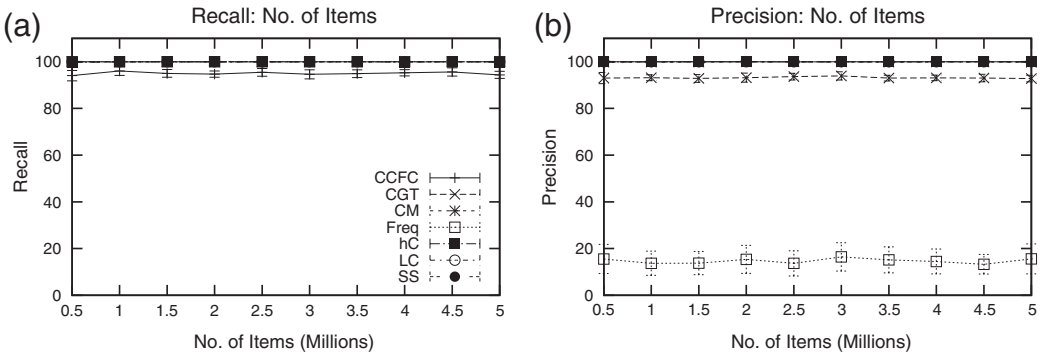

Fig. 1. Effect of number of transactions on recall and precision (items: 500,000–5,000,000; support: 0.001; Zipf: 1.1; runs: 20).

In several situations though, we are more interested in knowing the frequencies with which items have appeared in several different time windows in the past, and in comparing these frequencies among themselves. 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 during different time intervals. This knowledge is crucial for applications that deal with automatic trading, pre-trade analysis, post-trade execution, and market monitoring [43].
- 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 [17], and trace this activity over time. This information is in many cases in 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 in different time periods. The results of this analysis can be used for launching special promotions, performing inventory management, and in other applications [27].

The applications in the above examples require estimates in the item frequencies for *ad hoc* windows, rather than for the entire history of the data stream. As such, they allow for a richer set of data analytics, enabling analysts to capture seasonal trends, trends during particular events, as well as recent trends.

In this paper, we propose for the first time efficient solutions for the discovery of frequent items in *ad hoc* windows in streaming data given a small bounded memory. These solutions are based on existing *sketch* and *counter*-based techniques, which we extend in order to be able to effectively operate on the stream history.

We describe the *sketch*-based *TiTiCount*[1] algorithm that can be used to efficiently answer queries for frequent items in *ad hoc* windows. The algorithm uses a tilted timeframe for the representation of the past, which allows for frequency estimates for any user-specified windows in the past, using a small amount of memory. At the same time, the use of the tilted timeframe leads to estimates that are more accurate for the most recent windows; this accuracy of the estimates diminishes as we go further in the past. The tilted timeframe idea has been used in the past in several different problems, including time-variant data summarization [39,7,38], clustering [4], and storage [11]. In these studies though, the mechanism that was employed for the tilted timeframe had similar characteristics and properties, failing to make full use of the memory dedicated to the relevant data structures. Instead, in our work we describe a novel mechanism for implementing a tilted timeframe. It uses a new algorithm for shifting data between windows, which maintains an increased amount of information.

Furthermore, we propose *TiTiCount +*, an enhanced algorithm for query answering that makes use of the new tilted timeframe model. 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.

Finally, we propose *TiTiCount+h*, an enhanced algorithm that extends *sketches* by adding concise histograms to the tilted timeframe windows. Histograms provide a more detailed representation of frequencies within each window, resulting in better frequency estimates for the query windows. The challenge in this case is to devise a representation that uses minimal space, since the space budget used by the histograms is taken away from the counters of the sketch-based algorithms for frequency estimation. In this work, we describe a technique that uses only a few bits for each histogram. As we discuss later on, this representation is effective enough to lead to noticeable performance improvements.

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

- We describe *sketch*- and *count*-based algorithms that can estimate the frequency counts of hot items in the data stream history. Our approach efficiently supports queries on *ad hoc* windows in the past.
- 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 straightforward approach.

[1] Tilted Timeframe Count.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

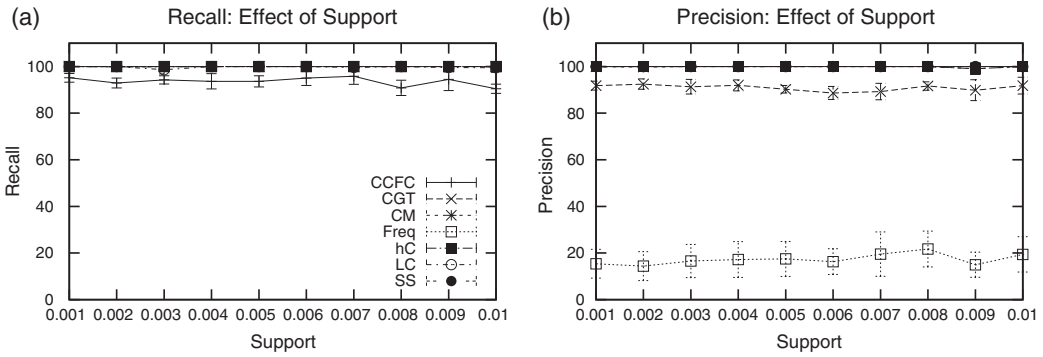

Fig. 2. Effect of support on recall and precision (items: 5,000,000; support: 0.001–0.01; Zipf: 1.1; runs: 20).

- We improve the basic algorithm, leading to a new tilted timeframe model, 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.
- We further extend the tilted timeframe model to support non-linear frequency estimation. Our solution, based on bitwise histograms, does not require additional memory, and leads to significant performance improvements.
- 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 algorithms.

The rest of the paper is organized as follows. We start by discussing the related work (Section 2) and giving some necessary background for the problem of mining data streams for frequent items (Section 3). In Section 4, we formally describe the problem of frequent items in *ad hoc* windows. Section 5 describes the details of our algorithms. Our experimental evaluation is presented in Section 6, and we conclude in Section 7.

This study extends a preliminary version of the work [41] by extending the discussions on the related work background material, providing proofs for the lemmata and more details for the proposed algorithms, describing additional algorithms based on lightweight histograms for more accurately answering queries on skewed distributions, and including a complete set of experimental results on the properties of the proposed algorithms that were also conducted on two additional real datasets.

2. Related work

In the recent years, numerous studies have focused on problems related to streaming data, ranging from practical applications to theoretical questions [22,36]. There is a wealth of work on the problem of identifying frequent items in streaming data. The Frequent (FREQ) [25], the Space Saving (SS) [34] and the Lossy Counting (LC) [33] 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) [15], Count-Min (CM) [14], CCFC [10] and *hCount* (HC) [24] 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. We note that all the above works solve the problem of identifying frequent items in a landmark window of the stream, i.e., in a window that contains all the data values seen up to the current time point. The important difference between these works and our approach is that we want to be more flexible in identifying frequent items, enabling the identification of frequent items in *ad hoc* windows in the data stream history. This is a more complex problem (given that we want a resource-efficient solution), but allows for a much wider set of analysis opportunities of the data.

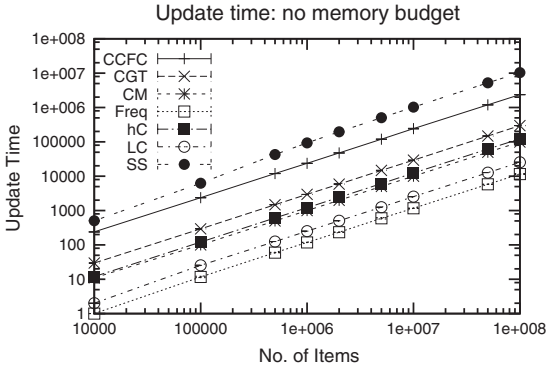

Fig. 3. Update time without memory budget. Note logarithmic scale on both axes.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

In order to achieve our goal, we use time-decaying approximations, an idea that has been also used in other areas, such as online time series summarization [39,7], streaming data clustering [4], and data warehousing [11]. Contrary to previous studies, we describe a different tilted timeframe model, which is based on a novel technique for shifting data between the data structures used by the model. As we discuss in the following sections and experimentally demonstrate in our evaluation, the new tilted timeframe model that was specifically designed for the frequent item identification task, leads to a significantly improved performance.

Other works have studied the problem of efficiently identifying and maintaining frequent itemsets over streaming data [8,13,28,44]. (For an examination of algorithms for frequent itemsets, the interested reader is referred to the survey by Cheng et al. [12].) 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 [31,40], or tilted time windows [19,9,29], in order to focus on the transactions in the recent past of the data stream.

One of the above approaches, the *FP-Stream* [19] uses a logarithmic tilted timeframe window model to identify frequent itemsets in *ad hoc* windows. However, it makes use of the *FP-Tree* structure, which has been specifically designed for itemsets (rather than items), rendering it inefficient for the frequent items problem. In our study, the timeframe window model used by the baseline algorithm *NaiveCount* is equivalent to the model used in *FP-Stream*. The *FP-Tree* structure is substituted by state-of-the-art techniques that have been proposed to identify frequent items. Moreover, as mentioned earlier, in our work we describe novel shifting schemes for the tilted timeframe, which are used by *TiTiCount +* in order to deliver significant performance improvements.

Recent works propose novel techniques to identify frequent itemsets. In [30], frequent itemsets are identified by processing the data stream through a sliding window. In this case, the frequent itemsets refer to the window content (i.e., they are frequent with respect to the contents of the current window). The proposed methods do not support the identification of frequent itemsets in *ad hoc* windows in the data stream history. In [16], frequent itemsets in the recent past are maintained by processing the data stream through a sliding window of varying size. The window can expand and shrink adaptively in order to better model the most recent frequent itemsets. Once again, the proposed algorithms cannot answer queries for arbitrary windows in the stream history. Moreover, our work is different from both the above studies in that it focuses on frequent items (not itemsets), a problem variation that requires specific techniques (i.e., the ones mentioned at the beginning of this section) in order produce accurate results in an efficient manner.

Finally, another recent work studies a problem relevant to frequent items [42]. In this case, the problem is to compute the k most frequent items in a data stream by processing stream items through sliding windows of dynamic length. The support of each item is defined as the maximum of the item frequency over all window lengths. Once again, the frequent items may refer to the entire data stream history and queries on the frequency of items in *ad hoc* windows cannot be supported, which is the focus of our work.

3. 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 E 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].

Due to the streaming nature of the data, any algorithm which deals with finding frequent items needs to take into account the following constraints.

- The algorithm can only see each item once. It is not plausible in terms of space and time to store all the data in the stream and perform multiple passes over it.

Table 1
Symbols used in the paper and their explanations.

Symbol	Description
S	Data stream
E	Largest item that can exist in stream S
T_i	ith item of stream S
T_n	Most recent item in stream S
ϕ	User-defined threshold for frequent items
w	Query window
w_{min}	Last recent point in window w
w_{max}	Most recent point in window w
L	Least recent item that can be part of w
b	Window batch size
$c[i]$	ith counter in tilted timeframe model
M	Number of counters for each *hCount* hash function
H	Number of *hCount* hash functions
q	Number of bits used by a histogram bucket
R	Number of histogram buckets

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

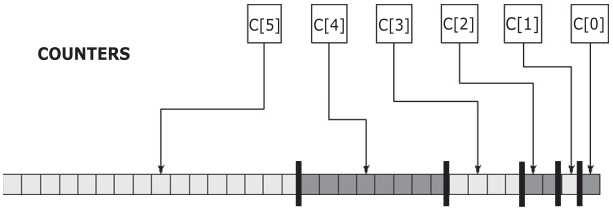

Fig. 4. Tilted-time windows (new items are inserted in the rightmost counter).

- The value of E can be larger than the available main memory. Thus, it may not always be possible to keep counts of all individual items.

The above constraints have led to the development of efficient approximate algorithms that can accurately estimate the true frequency counts (at least for the most frequent items). These algorithms usually make use of a user-defined error parameter which determines the trade-off between accuracy and space/time used.

Past studies can be divided into two groups: *counter*-based, and *sketch*-based. In the first group, a limited number of counters is maintained. The set of monitored items is dynamic, thus it can change over time. Whenever a new item comes in and if its counter exists, it is incremented. Otherwise, multiple heuristics can be used to decide if (1) ignore it or (2) substitute an existing counter with it. Algorithms falling in this category are Frequent (FREQ) [25], Lossy Counting (LC) [33] and Space Saving (SS) [34]. In the second group, the approximate frequency of all the items is maintained and some quality guarantees are usually ensured. An experimental comparison of algorithms following both methods can be found in [32]. In this category we have algorithms Combinatorial Group Testing (CGT) [15], Count-Min (CM) [14], *CCFC* [10] and *hCount* (HC) [24].

3.1. Comparison of existing algorithms

In the following paragraphs we describe experiments that compare the performance of the available algorithms for mining frequent items in streaming data. Our implementation of the *FREQ*, *LC*, *SS*, *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 tests on synthetic data sets, and measured time and space usage for all the above six algorithms. Synthetic data was generated using a stream generator that generated random numbers within a specified range, based on a Zipfian distribution, whose skew can be adjusted using the Zipf parameter.

In order to evaluate the quality of the results obtained, we used the two standard measures of *recall* and *precision*. Recall is defined as the percentage of the true frequent items that are found by the algorithm. Precision is the percentage of items identified by the algorithm, which are truly frequent. We verify the results by keeping the exact count of all items using a brute-force approach, and comparing the results with the algorithm output.

For each experiment, several independent runs were performed. In the graphs, we report the results averaged over all the runs, as well as the 95% confidence intervals. The number of runs is indicated wherever applicable. In the following discussion, we present the results of the experiments using a single dataset. Note though, that similar trends were observed when using other datasets. We omit these results for brevity.

The purpose of the first experiment is to examine the accuracy of the algorithms as a function of the size of the stream. The recall was reasonably high for all algorithms throughout this experiment with the exception of *CCFC*, which fall below 90% (see Fig. 1(a)). The results for precision, depicted in Fig. 1(b) were more varied. *FREQ* had a low precision, which is probably due

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

When new transaction T_n arrives:
1   use hCount to determine the set of counters C related to T_n
2   for each counter in C
3       update the counts of c
4   if (n mod 2b) == 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 C corresponding to T_n
7       call GetFreqEstAtom(C,[t^q_min,t^q_max])
```

Fig. 5. Main skeleton of the proposed algorithms.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

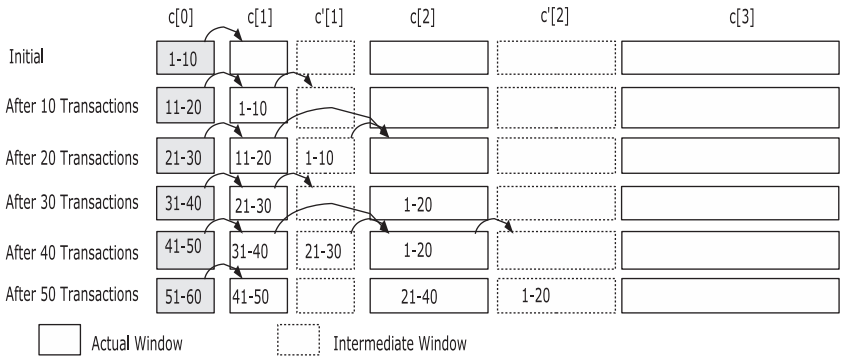

Fig. 6. Shifting with intermediate windows (batch size $b = 10$). Gray boxes denote windows processing new stream values.

to the fact that it is designed to output all top-k items, regardless of the support level. The rest of the algorithms performed excellently, with the exception of *CGT*, which performed slightly worse. Note that the number of transactions does not seem to affect the performance of any of the algorithms.

In the following experiment, we evaluate the performance of the algorithms when the support threshold varies. Recall for all the algorithms (shown in Fig. 2(a)) was reasonably high throughout the support range that we tested, except for *CCFC*, which performed a little bit worse. Regarding precision, *FREQ* is performing very low (averaging a precision of a little bit less than 20%), *CGT* is performing slightly above 90/ (see Fig. 2(b)).

We also measured the scalability and time requirements of the algorithms by running experiments with 10 to 100million transactions. Our results show that all algorithms scale linearly in time with respect to the number of transactions (see Fig. 3). *CCFC* requires the longest time, whereas *LC* and *FREQ* are the most time-efficient.

We selected the *hCount* and *SS* algorithms as representatives to be used in the frequency estimation component of our approach, because they have several desirable characteristics. Namely, they exhibit a consistently good performance across various conditions, they have low time complexity, and are relatively easy to implement. Note that this choice is not restrictive in any way, and in our techniques the *hCount* and *SS* algorithms could be replaced with any other suitable frequency estimation algorithm. We present our model considering *hCount* by default and then a variant substituting it with *SS*.

4. Frequent items in *ad hoc* windows

In this section, we formally define the problem of frequent item discovery in user-defined windows in the data stream history, and we give a brief overview of our approach. In Table 1, we summarize the most important symbols used in the rest of the paper.

Table 2
Internal *hCount* data structures for buckets C[0], C[1] and C[2], where m_i and h_j refer respectively to the i-th counter for the j-th hashing function.

	m_1	m_2	m_3	m_4	m_5
C[0]					
h_1	5	0	3	1	1
h_2	5	0	2	1	2
h_3	3	2	5	0	0
h_4	5	2	1	1	1
C[1]					
h_1	4	3	1	1	1
h_2	2	2	2	2	2
h_3	10	0	0	0	0
h_4	0	3	1	6	0
C[2]					
h_1	9	3	4	2	2
h_2	7	2	4	3	4
h_3	13	2	5	0	0
h_4	5	5	2	7	1

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

```
Let K :=number of windows
    c[i] := counter windows, 0 ≤ i ≤ K
    c'[i] := intermediate windows, 1 ≤ i ≤ K − 1

1 procedure PerformShift()
2    temp := 0
3    for i = 0 to K-1
4        exchange content of c[i] and temp
5        if c'[i] is empty
6            c'[i] = temp
7            exit loop
8        else
9            temp := temp + c'[i]
10           reset c'[i]

11 procedure GetFreqEst(relevant counters C, window interval [t^q_min,t^q_max])
12  for all counters c[] ∈ C do:
13       sum := 0 /* frequency estimate */
14       wc := 0 /* number of transactions considered so far in result */
15       i := 0 /* window interval index */
16       started :=FALSE
17       while (wc < t^q_max)
18           if (started == FALSE) and (wc > t^q_min)
19                sum := sum + c[i]
20                wc := wc + capacity of c[i]
21                started := TRUE
22           else if (started == TRUE) and (wc < t^q_max)
23                sum := sum + c[i]
24                wc := wc + capacity of c[i]
25           else
26                sum := sum + c[i]
27                exit loop
28           i:= i+1
29  return(min(sum) across estimates from all counters c[] ∈ C) /* according to hCount */
```

Fig. 7. Core functionality of the *NaiveCount* algorithm.

4.1. Problem definition

Let the data stream S be represented by $\{T_1,T_2, \ldots,T_n\}$, where T_i denotes the *ith* 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 (e.g., $[w_5, w_1 2]$ refers to a window that includes all the items in the stream from T_5 to T_{12}). The length, or size (in terms of number of items), 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 frequent items in *ad hoc* windows problem for a stream S.

Problem 1. Frequent items in *ad hoc* windows (FIAW) 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 history of the stream. The parameter L determines how far in the past the query window can refer to. Essentially,

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

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

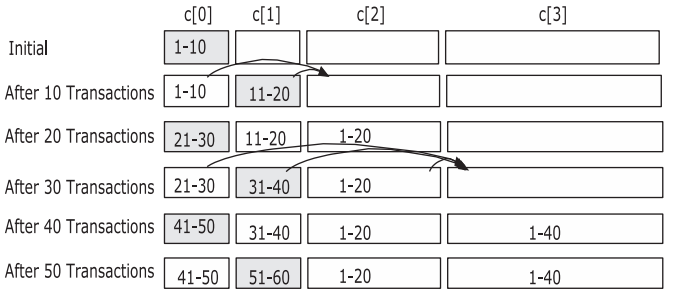

Fig. 8. Shifting without intermediate windows (batch size $b = 10$). Gray boxes denote windows processing new stream values.

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.

4.2. Proposed approach

Previous works have studied the problem of identifying frequent items in the entire history of a data stream [25,33,15,14,10,24,34]. What is fundamentally different in our case is that we wish to identify frequent items in arbitrary window intervals of the stream. In

```
Let n :=current transaction number
    b := batch size
    K := number of windows
    c[i] := counter windows, 0 ≤ i ≤ K

1 procedure PerformShift()
2       batch number bn := n/b
3       for i = K-2 to 0
4           if (bn mod 2^(i-1)) == 0
5               move contents from counters {c[0],...,c[i]} to c[i + 1]
6               exit loop
7       if bn is odd
8           reset counter c[1]
9       else
10          reset counter c[0]

11 procedure GetFreqEst(relevant counters C, window interval [t^q_min,t^q_max])
12      for all counters c[] ∈ C do:
13          sum := 0 /* frequency estimate */
14          seen := 0 /* number of transactions seen so far */
15          started := FALSE
16          for i = K-2 to 0
17              if (bn mod 2^(i+1)) > 2^i
18                  seen := seen + width of window w_{i+1}
19                  if (started == FALSE) and (seen ≥ t^q_min)
20                      started := TRUE
21                      sum := sum + weighted fraction of c[i + 1]
22                      if seen ≥ t^q_max
23                          exit loop
24                  else if (started == TRUE)
25                      if seen ≤ t^q_max
26                          sum := sum + c[i + 1]
27                      else
28                          sum := sum + weighted fraction of c[i + 1]
29                          exit loop
30      return(min(sum) across estimates from all counters c[] ∈ C) /* according to hCount */
```

Fig. 9. Core functionality of the *TiTiCount* algorithm.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

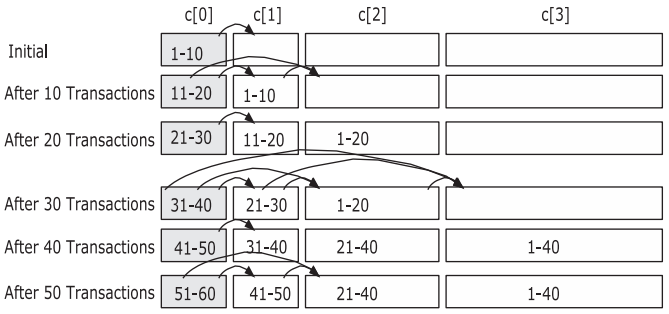

Fig. 10. Value added shifting (batch size $b = 10$). Gray boxes denote windows processing new stream values.

order to solve the *FIAW* 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 history of the stream, that is, the last L transactions, in fixed-size intervals, and estimate the 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 *FIAW* problem, we propose the use of *tilted-time* window intervals. (Similar approaches have been studied in other applications as well [11,4,39,7]. 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 *FIAW* 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.

5. Algorithms for frequent items in *ad hoc* windows

In this section, we present algorithms for the *FIAW* 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 start using the *hCount* sketch in order to estimate the frequency counts within a given window interval. The *hCount* algorithm (for details see [24]) maintains a sketch of size $M \times H$, which can be thought of as a hash-table with M rows and H columns, containing $M \times H$ counters. The M and H parameters are determined by the data characteristics and the allowed error. More specifically, M is set to $\frac{e}{\epsilon}$, where e is an element of the stream and ϵ is the user-defined tolerance on the approximation error to the true item frequencies; h is set to $log\left(-\frac{E}{log(1-\delta)}\right)$, where E is the value of the largest item that can exist in the stream and δ is the probability that the algorithm misclassifies an item as frequent, when it is not, or not frequent, when it actually is.

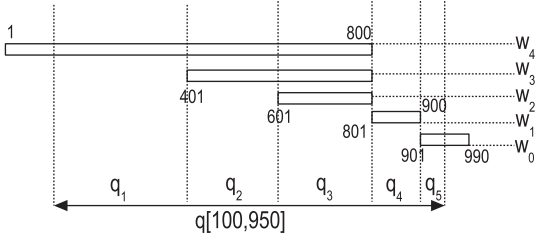

Fig. 11. Example of query answering for *TiTiCount +*.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

```
Let n :=current transaction number
    b := batch size
    K := number of windows
    c[i] := counter windows, 0 ≤ i ≤ K

1 procedure PerformShift()
2     for i = K-2 to 0
3         if (n/b mod 2^(i-1)) == 0
4             move contents from counters {c[0],...,c[i − 1]} to c[i]



5 procedure GetFreqEst(relevant counters C, window interval [t^q_min,t^q_max])
6     for all counters c[] ∈ C do:
7         sum := 0 /* frequency estimate */
8         find ws := the counter c[] corresponding to the smallest window that contains t^q_max
9         find boundaries of window to which counter ws maps:[t^ws_min, t^ws_max]
10        if t^ws_min ≤ t^q_min
11            find im_small := the counter c[] corresponding to the immediately
                               smaller window overlapping with ws
12            if im_small exists
13                sum := weighted fraction of (ws - im_small)
14            else
15                sum := weighted fraction of (ws)
16            exit loop
17        else
18            sum := (GetFreqEst([t^q_min,t^ws_min − 1])+GetFreqEst([t^ws_min,t^q_max]))
19    return(min(sum) across estimates from all counters c[] ∈ C) /* according to hCount */
```

Fig. 12. Core functionality of the *TiTiCount+* algorithm.

The algorithm uses a set of H hash functions to map each item of the dataset to H different counters, one in each column of the table. The hash functions are of the form:

$$\mathcal{H}_i(k) = (a_i \cdot k + b_i) \bmod P) \bmod m, 1 \le i \le h$$

where a_i and b_i are two random numbers, and P is a large prime number. Thus, each data item has a set of H associated counters, which are all incremented when this item appears in the stream. Then, the estimated frequency of an item is simply the minimum of the values of all its associated counters.

In our case, instead of a single window interval, the algorithm has to operate with K intervals. These windows follow a tilt timeframe as follows. The first window, w_0, covers the b most recent stream values, that is, transactions T_{n-b+1}, ..., 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 $w_i = 2^{i-1}b$, $1 < i < K$.

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 $0 \le m < M$, $0 \le h < H$.[3] These arrays of counters correspond to the K windows, as shown in Fig. 4. The first element (in some cases also the second, as we will 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 Fig. 5). 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.

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

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

5.1. Basic algorithm

The straightforward approach to implement the shifting operation is to use intermediate windows (and corresponding counters). The intermediate windows are needed in order to facilitate the process of shifting the contents of a window to the successor window, which is always double the size (since we are using logarithmic tilted time-windows). As shown in Fig. 6, 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. We will refer to each window by the number it corresponds to, the first being window 0 (and the last in Fig. 6, window 3). The intermediate counters are denoted as $c'[]$ in the figure, and correspondingly, we will refer to the intermediate windows by the number and a prime. For example, the two intermediate windows in Fig. 6 are windows 1′ and 2′.

The shifting of data between windows is performed in the following way (refer to Fig. 6 from top to bottom). Initially, when the first value of the stream arrives, it is only window 0 that is active. When window 0 has processed b values it is full, and it shifts its contents to window 1. Window 1 at this point stores a counter for values T_1, …, T_{10}, while window 0 continues to process the new values from the stream. When 0 gets full again, it shifts its contents (T_{11}, …, T_{20}) to window 1, which in order to receive them has to shift its previous contents to the intermediate window 1′. We then have windows 1 and 1′ each storing counters for 10 values, which is exactly their size, and window 0 continues to process the new values of the stream. When 0 gets full again, we have a new cascade of shifts, leading this time both windows 1 and 1′ to shift their contents to window 2, which is of double the size of either 1 or 1′. Fig. 6 shows how this process continues until T_{60} appears in the stream.

In the following example, we demonstrate how this shifting process works for the counters $C[]$. Table 2 shows the contents of the *hCount* internal data structures for counters $C[0]$, $C[1]$ and $C[2]$ after the first 30 stream values (refer also to Fig. 6). The values seen so far in the stream are hashed according to the hash functions used in the counters $C[]$ (remember that each counter implements *hCount*). Note that in our examples for counters $C[0]$ and $C[1]$, for all hashing functions h_i, $1 \leq i \leq 4$, the sum over all counters is 10, since the batch size b is 10. When the 31st stream value arrives, the contents of counters $C[0]$ and $C[1]$ are summed up and moved to counter $C[2]$ (see Table 2 right). The contents of counter $C[2]$ are simply the aggregate (sum) of the corresponding contents of counters $C[0]$ and $C[1]$. Therefore, the sum over all counters for any hashing function h_i in $C[2]$ is 20. Whenever a shifting operation occurs, the same process repeats (this is also true for the other shifting schemes described in the following sections).

In Fig. 7, we show the outline of the algorithm, which we call *NaiveCount*. Procedure *PerformShift()* (line 1) implements the shifting of windows as described above (see Fig. 6).

Answering item frequency questions in this model is simple, and performed in the following way. When a query for the frequency of an item in a specific window interval w_q comes in, we identify the counters that store information on time intervals overlapping with w_q, and we sum the estimates from these counters (lines 17–28). 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 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.

5.2. Reducing the memory requirements

We observe that we can reduce the memory requirements of the algorithm by discarding the intermediate windows. Note that in the shifting scheme implemented by *NaiveCount*, there is a strict order in which stream values are stored in each window: each successive window is storing values that are older than its predecessors. We now explore a scheme that relaxes this assumption.

Under the new shifting scheme (see Fig. 8), we have to keep track of which counters correspond to which windows, since the values are no longer stored in order of increasing age as before. The benefit is that this scheme allows us to directly move counter contents to the next window, without the need of intermediate windows. We also employ *lazy* shifting, by allowing also the second window to process new stream values, thus, only shifting contents when absolutely necessary and saving some shift operations.

As depicted in Fig. 8, initially it is window 0 processing new values. When it gets full, it does not shift its contents. Instead, now it is window 1 that processes the new values from the stream. When 1 gets full, too, we have two windows of the same size that have become full, and it is time to shift their contents: the contents of both are shifted to window 2, and window 0 starts processing the new stream values once again. Fig. 8 shows how this operation evolves until item T_{60} has appeared in the stream.

Using this shifting scheme, 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.

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 Fig. 11. The edges of w_q are not aligned

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

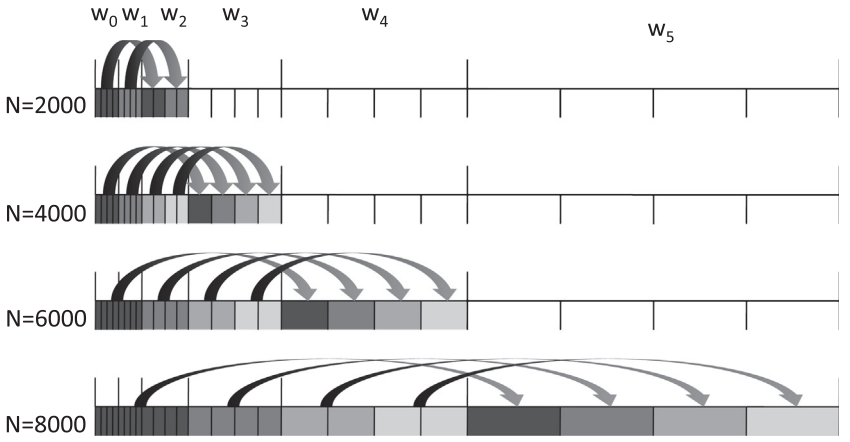

Fig. 13. Maintaining histograms across shifts.

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. The above operation is reflected in lines 21 and line 28 of the *TiTiCount* algorithm, depicted in Fig. 9.

The simplest way of computing the weighted fraction is to adopt a linear model (i.e., the fraction is directly computed as the amount of overlap). As shown by the experiments, this simple idea improves the quality of the results substantially. More

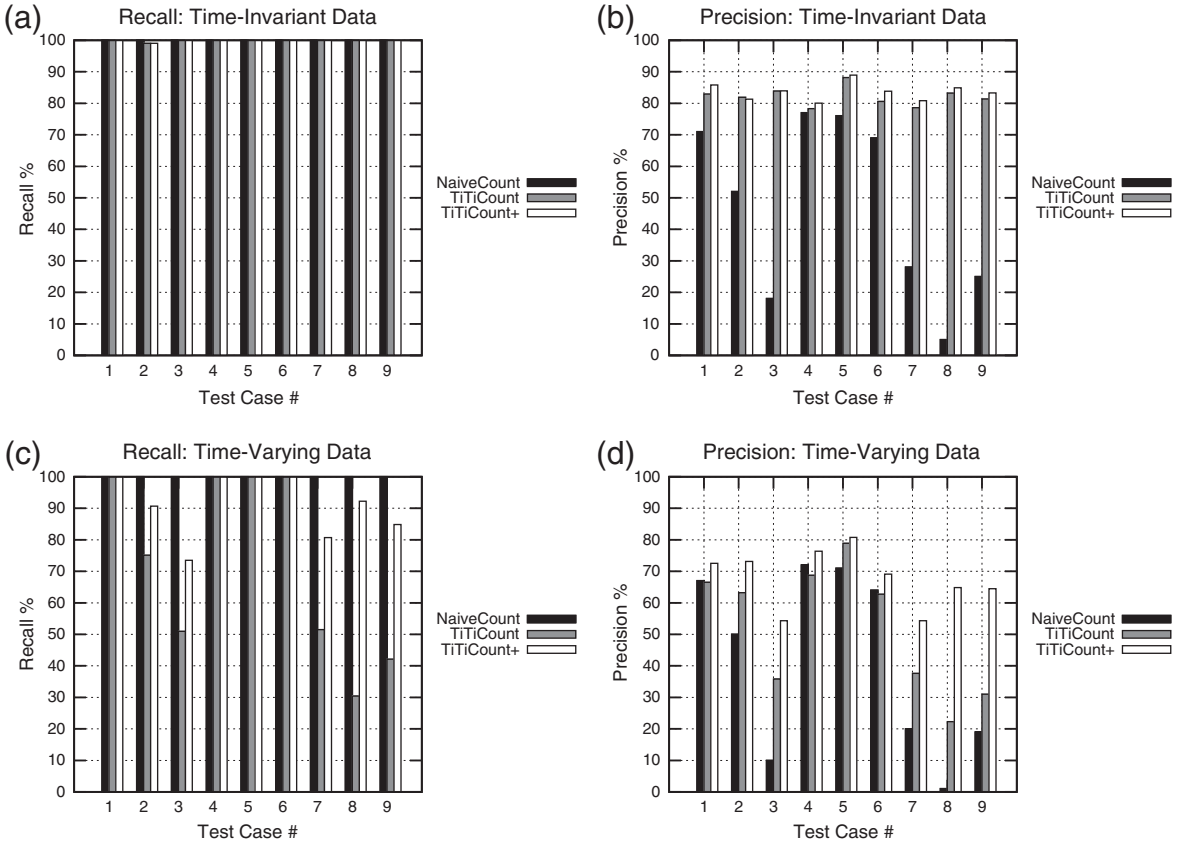

Fig. 14. Performance on time varying and non-varying data distributions in terms of precision and accuracy.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

sophisticated techniques can be applied. For example, even limited knowledge on the distribution of the frequencies within a window could lead to a more accurate non-linear model). In the next sections, we show how compact histogram structures can be introduced to extend *TiTiCount*. However, as we show in the experimental evaluation of the algorithms, introducing non-linear models does not always pay off in terms of accuracy.

5.3. 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 Fig. 8 (example of shifting for *TiTiCount*), we observe that information regarding transactions 1–20 is stored in both $c[2]$ 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, window 0, process all the new data stream transactions. Then, window i receives the contents of all preceding windows, 0, 1, …, $i-1$ every time that $(n/b \ mod \ 2^{i-1})$ equals to zero.

Fig. 10 demonstrates how this shifting scheme operates in practice. Initially window 0 processes the new values. When it gets full after having processed $n = 10$ items, it shifts its contents to window 1, since $(10/10 \ mod \ 2^0) = 0$. After 30 items have arrived in the stream, we are at the position depicted in the fourth row of Fig. 10, with windows 1 and 2 full, and window 0 processing items T_{31}, …, T_{40}. When the 40th item has been processed, window 3 received the contents of all previous windows (since $(40/10 \ mod \ 2^2) = 0$), and the same is true for window 2 (since $(40/10 \ mod \ 2^1) = 0$) and window 1 (since $(40/10 \ mod \ 2^0) = 0$). When the 60th item has been processed, it is only windows 1 and 2 that get updated, and not window 3 (since $(60/10 \ mod \ 2^2) = 2$).

This new window shifting scheme 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 negligible. The memory requirements are the same as before, namely, KS.

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.

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.*

Proof. In the algorithm, the window configuration is changed only when the window content is shifted. Whenever the criterion for shifting the windows is satisfied, the following holds true: every time a shift of preceding windows to $c[i]$ takes place, it will always be followed by a shifting of preceding windows to every $c[j]$, where $j < i$. This follows from the shifting criterion: if $(n/b \bmod 2^{i-1})$ is 0, then $(n/b \bmod 2^j)$ for all $j < i-1$ will always be zero as well.

When this happens, each newly filled window $c[i]$ will be formed from all preceding windows $c[0]$ to $c[i-1]$. Similarly, the next window $c[i-1]$ will be formed from the same preceding windows $c[0]$ to $c[i-2]$. So $c[i]$ and $c[i-1]$ will be overlapping with their end points coinciding, these end points being the most recent points as dictated by $c[0]$. Correspondingly, the start point of $c[i-1]$ will be in the middle of $c[i]$ (remember that each window has a size double the size of its preceding window). This proves Lemma 2. This is true whenever any shifting takes place. The end points will coincide and the start point of each subsequent window be in the middle of the preceding window. This implies that whenever this is an overlap of windows, the smaller window is always completely contained by the larger window. This proves Lemma 1. □

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 $w_{sq_1}, \ldots, w_{sq_j}, \ldots, w_{sq_l}$ 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. They also lead to accurate results, since they always use the information of the finest granularity possible. The following example explains how this algorithm works. Assume we have five window intervals, w_0, …, w_4, and that the current transaction number is 990, as shown in Fig. 11. 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,

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

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 (since w_1 is the only counter interval that can answer q_4, and w_2 is the *smallest* counter interval corresponding to query q_3). 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 applies the weighted fraction model, since q_1 is not interested in the first 100 transactions of w_4.

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 changes over time.

The outline of *TiTiCount +* is shown in Fig. 12.

5.4. Employing counter-based frequency estimation

So far we have used the *TiTiCount* family of algorithms by adopting *hCount* [24] for frequent item discovery. *hCount* is a sketch-based technique that maintains the approximate frequency of all the items. In contrast, counter-based techniques maintain a limited number of counters. The *Space Saving (SS)* [34] algorithm is among the best performers [32], with certain advantages over the sketch-based methods. The algorithm monitors the last *C* distinct items, and reallocates the counter with the lowest frequency whenever an item that is not monitored appears in the stream. Substituting *hCount* with *SS* in *TiTiCount* and *TiTiCount +*, we obtain respectively two novel variants *TiTiCount-s* and *TiTiCount-s +*. Similarly to *hCount*, *SS* is extended to an array of $C \times K$ counters where *C* is the number of counters maintained by *SS* and *K* is the number of windows in the tilted timeframe model.

Nevertheless, there is also a caveat in using *SS*. Recall that *TiTiCount +* exploits redundant information by (1) preferring the smallest window intervals and via (2) subtraction operations. The benefit of the subtraction mechanism is less effective when using *SS*, because there is no guarantee that adjacent windows maintain counters for exactly the same items, thus not allowing subtractions for some of the counters. However, as the experiments show, the subtraction mechanism still proves beneficial in several cases.

5.5. Non-linear frequency estimation with sketches

The *TiTiCount +* algorithm estimates frequencies within each window by computing the linear weighted fraction of the involved counters. A more accurate estimate can be provided by non-linear models. In this section we present the *t TiTiCount + h* algorithm, a *TiTiCount +* extension that adopts histograms to provide fine-grained frequency estimates. In this work, we focus on a very concise (in terms of memory cost) histogram structure. However, other more sophisticated histogram techniques [23,21] can also be applied.

Each counter $c[i]$ of *TiTiCount +* is paired with an equi-width histogram $h[i]$, where the frequency of each bucket is represented by one of a predefined set of frequency levels. Without loss of generality, we assume that the number of frequency levels is a power of two: $f = 2^q$. Then, each histogram of *R* buckets consumes $R \cdot q$ bits of memory.

We start by describing how we construct these histograms, and then how we use them in the context of the tilted timeframe model. Given basis (counter) frequencies $c_1, \ldots, c_R$, we want to construct their histogram. Bucket B_j, $1 \le j \le R$ is represented by frequency level f_j, computed from frequency c_j. Frequency levels depend on frequency thresholds, defined as follows:

$$F_j = \frac{j}{f} \cdot c_{max} \tag{1}$$

for $j \in \{1,..,f\}$, where $c_{max} = max(c_1, \ldots, c_R)$. The intuition here is that the highest frequency level represents the upper limit c_{max}, while the lowest represents zero. Determining the frequency level f_j of bucket B_j is then straightforward:

$$f_j = argmin_k\left(c_j \le F_k\right) - 1 \tag{2}$$

where $k \in \{1, \ldots, f\}$.

We now describe how these histograms are constructed and maintained over time in the context of the tilted timeframe model. Recall that, for each histogram $h[i]$, we need the basis frequencies $c_1, \ldots, c_B$. This procedure is triggered whenever counters $c[0], \ldots, c[i-1]$ get shifted to $c[i]$. The basis frequencies used to construct histogram $h[2]$ rely solely on the first two counters, $c[i]$ and $c[2]$. For all the subsequent histograms $h[i]$ with $i > 2$, we linearly partition the time interval between windows $w_i - 1$ and w_0 into *R* equi-width intervals $\{T_1, \ldots, T_R\}$. Counters falling within interval T_j are added up to c_j. For $i \ge 1$, the width of window w_i is $b2^{i-1}$ and it is associated to counter $c[i]$. The width of the buckets in histogram $h[i]$ is $\frac{b2^{i-1}}{4} = b2^{i-3}$.

Fig. 13 shows an example width four buckets and four frequency levels, and batch size equal to $b = 1000$. When the stream length reaches $n = 2000$, the first two windows get shifted to window w_2. The histogram $h[2]$ (attached to window w_2) is constructed such

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

that its first two and last two buckets share the same frequency level. When the stream length reaches $n = 4000$, windows w_0, w_1 and w_2 get shifted to window w_3. The histogram $h[3]$ (attached to window w_3) is constructed by estimating the basis frequencies $c_1, \ldots, c_4$, taking advantage of the previous counters and histograms. The next shift occurs at $n = 6000$, where the first four windows get shifted to window w_4, and then the first five windows get shifted to window w_5 at $n = 8000$. In this case we have two shifts as dictated by the tilted timeframe model of *TiTiCount +*. Histograms are updated at every shift to reflect the content of each counter. The same applies to all the subsequent shifts.

Last, we discuss how to exploit the extra information provided by histograms. Recall that the *TiTiCount +* algorithm answers user-supplied queries by combining frequency estimates along multiple windows of the tilted timeframe model. Given that window w_i representing interval [*wmin*,*wmax*] overlaps with the user-supplied query $q = [tqmin,tqmax]$, the involved counter and histogram are respectively $c[i]$ and $h[i]$, and the frequency estimate is $freq(q)$. For ease of exposition, we introduce the function $overlap(T1, T2)$ that returns the fraction of time interval $T1 = [T1_{begin}, T1_{end}]$ overlapping with time interval $T2 = [T2_{begin}, T2_{end}]$:

$$overlap(T1, T2) = \frac{max(max(T1_{min}, T2_{min}) - min(T1_{max}, T2_{max}), 0)}{T2_{max} - T2_{min}} \tag{3}$$

We start by formalizing the returned frequency estimate under the linear model assumption adopted by *TiTiCount +*:

$$freq(q, i) = c[i] \cdot overlap(q, w_i), \tag{4}$$

where q is the user-supplied query window, and i is the index of a window contributing to the result as identified by the *TiTiCount +* algorithm. Introducing histograms enables us to make more accurate estimates by applying the linear model assumption at a finer level (moving the focus from counters to buckets), treating the histogram as a discrete probability density function to be normalized:

$$freq(q, i)_h = c[i] \cdot \frac{\sum_{j=1}^{B} \left(f_j \cdot overlap\left(q, w_i^j\right)\right)}{\sum_{j=1}^{B} f_j}, \tag{5}$$

where w_i^j is the time interval of the jth bucket of histogram $h[i]$.

Example 2. Using the Example 10, assume that all the items in interval [1, 200] have value x, all the items in interval [201, 400] have value y and all the items in interval [401, 800] have value x. Suppose, a query comes that asks for the frequency of value x in interval [1, 200]. In this case, *TiTiCount +* will return 100 (the weighted fraction of 200), while *TiTiCount + h* will return 200, which is the correct value. The advantage of *TiTiCount +h* is magnified when the distribution of the values in the data stream changes over time and the query is not perfectly aligned with the window boundaries.

6. 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 sets of integers. It has about 6 million individual items.

Table 3
Query window intervals used as test cases.

n = 50,000			n = 60,000			n = 70,000		
No.	tqmin	tqmax	No.	tqmin	tqmax	No.	tqmin	tqmax
1	5000	45,000	4	5000	55,000	7	20,000	45,000
2	35,000	45,000	5	35,000	55,000	8	40,000	55,000
3	25,000	40,000	6	5000	50,000	9	40,000	65,000

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

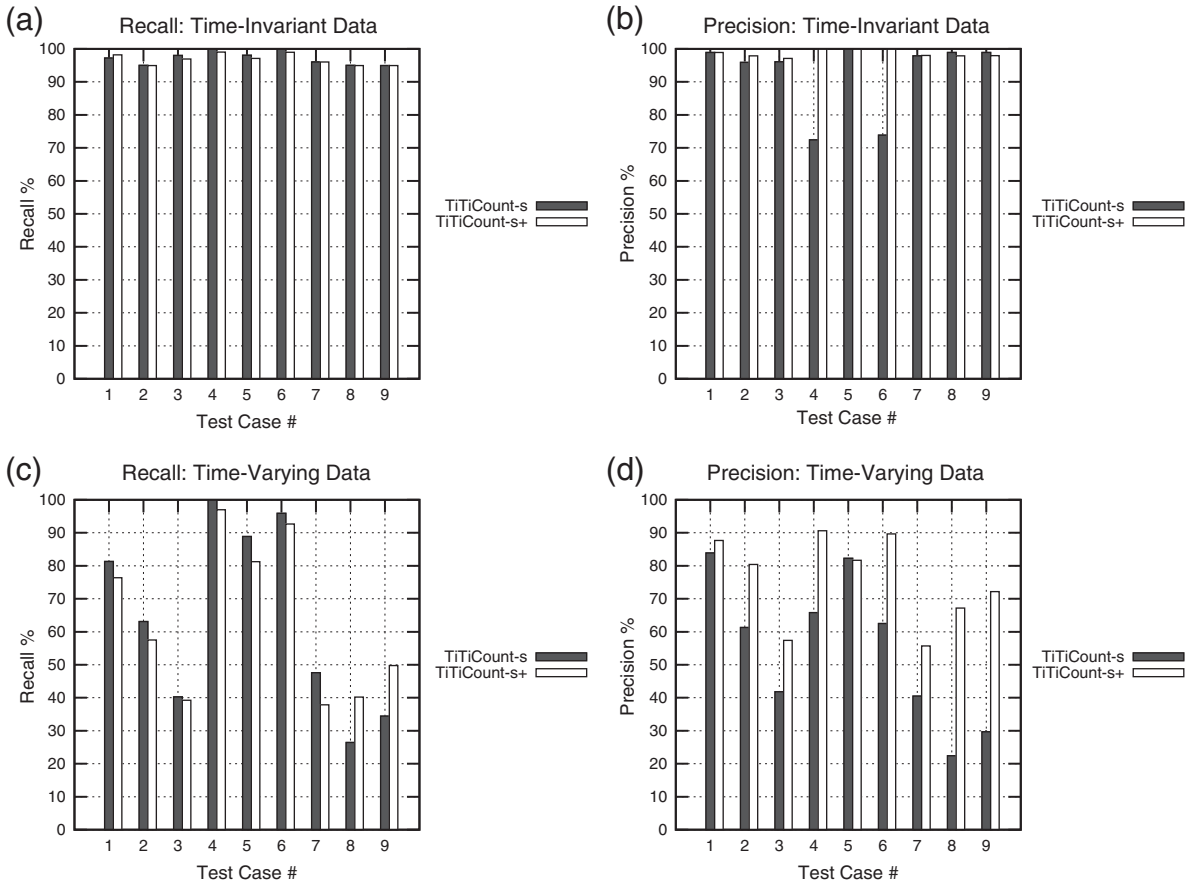

Fig. 15. Performance on time-varying and non-varying data distributions for *counters* based algorithms in terms of precision and recall.

- Retail [5]: It contains retail market basket data from an anonymous Belgian store. This dataset has about 0.9 million individual items.
- NASA [37]: It consists of the Field Magnitude (F1) and Field Modulus (F2) attributes derived from the Voyager 2 spacecraft Hourly Average Interplanetary Magnetic Field Data. This dataset has about 0.3 million individual items.
- Q148 [26]: It contains the values of the attribute Request Processing Time Sum (attribute number 148) from the clicks dataset. This dataset has about 0.2 million individual items.

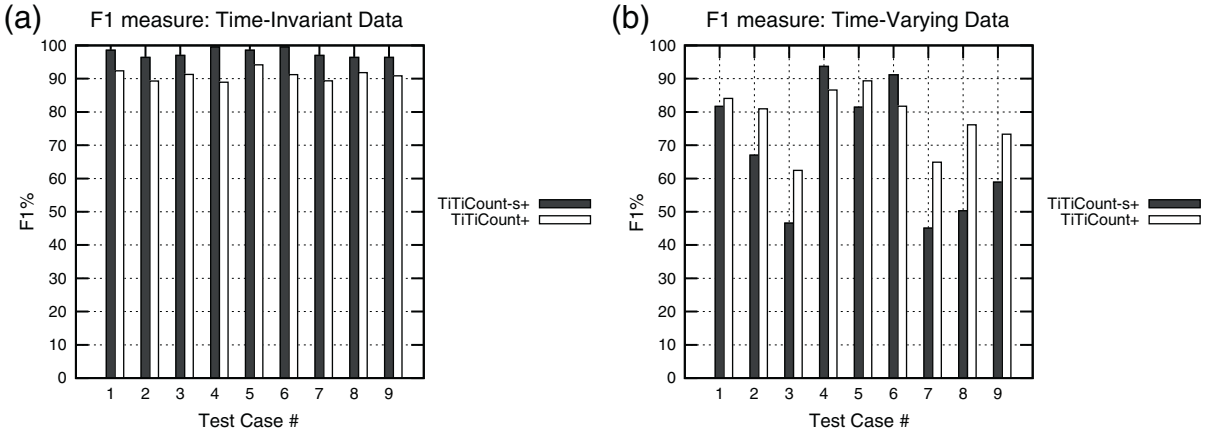

Fig. 16. Comparison of *TiTiCount-s +* and *TiTiCount +* algorithms in terms of F_1 score.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

All techniques were implemented in C/C++, and the experiments were run on a PC with a 2.13 GHz CPU equipped with 4 GB of RAM.

6.1. 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 Fig. 14, 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 3. All experiments used a batch size $b = 1000$, they were repeated 15 times, and results were averaged.

Fig. 14(a) and (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, especially for *NaiveCount*. *TiTiCount* and *TiTiCount +* average precision is close to 82%, with *TiTiCount +* performing slightly better. The performance of *NaiveCount* is notably worse, averaging a mere 46%.

In Fig. 14(c) and (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. Even 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 (refer to Fig. 14(d)).

For the remainder of the discussion, we do not consider the *NaiveCount* algorithm. We rather focus our attention to *TiTiCount* and *TiTiCount +*, which exhibit a consistently better performance.

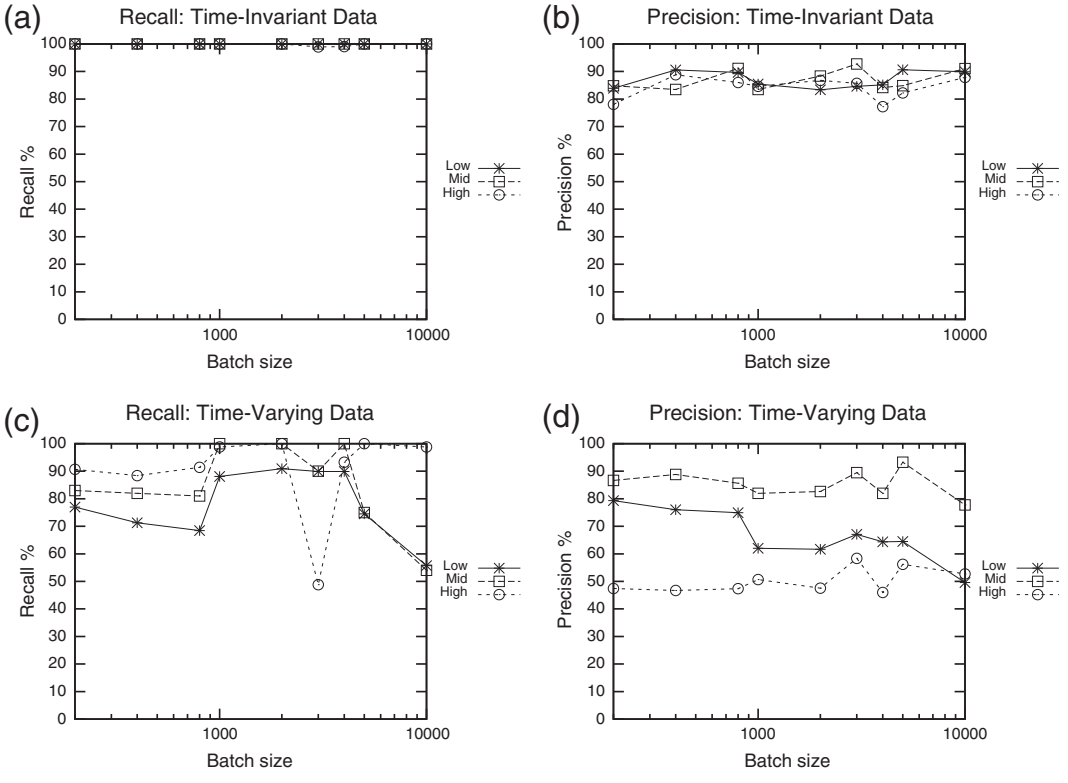

Fig. 17. Performance on time-varying and non-varying data distributions varying the batch size (*TiTiCount +* algorithm).

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

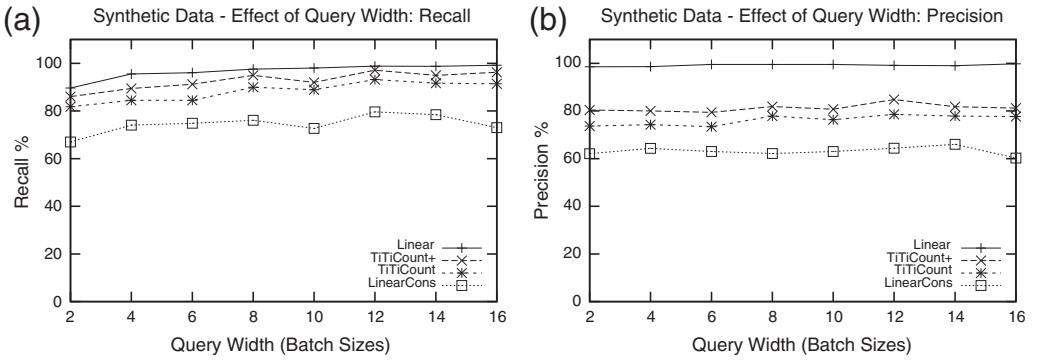

Fig. 18. Performance with respect to query width (dataset: synthetic, batch size $b = 1000$).

We continue by repeating the same experiment and comparing the *TiTiCount-s* and *TiTiCount-s +* algorithms. Fig. 15(a) and (b) show the recall and precision for the two algorithms, when run over a dataset with a stationary distribution. We observe that both algorithms have recall rates that average above 97%. Precision for *TiTiCount-s +* is consistently above 98%, while *TiTiCount-s* exhibits a worse and more variable performance.

In Fig. 15(c) and (d), we show the results of the same experiment, when run over a dataset with time-variant distribution. These experiments represent a more challenging setting for our algorithms, and the results demonstrate their qualitative difference. In terms of recall, *TiTiCount-s* performs slightly better for some of the test cases. However, *TiTiCount-s +* results in a considerably better precision.

The reason *TiTiCount-s +* produces more accurate results than *TiTiCount-s* for the time-varying dataset is because the *TiTiCount-s* algorithm relies solely on the weighted fraction mechanism to arrive at frequency estimates. In contrast, *TiTiCount-s +* uses the subtraction mechanism, which can effectively alleviate this problem and produces better estimates.

We now compare the *TiTiCount-s +* and *TiTiCount +* algorithms, the best performing among the *sketch* and *counter*-based variants. We report the F_1 scores, defined as a combination of precision and recall rates:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{6}$$

In Fig. 16(a) we show F_1 scores when run over a dataset with stationary distribution. *TiTiCount-s +* performs consistently better than *TiTiCount +*, with an average F_1 score above 98%. The trend is inverted in presence of time-variant distributions, as reported in Fig. 16(b): *TiTiCount +* outperforms *TiTiCount-s +* in most of the cases. We observe that the benefit of the subtraction mechanism is less effective using *SS* in conjunction with time-variant distributions, because adjacent windows do not maintain all counters involved in the subtractions. Remind that *tect SS* is a *counter-based* technique, thus it maintains counters only for a small fractions of the items believed to be the most frequent. In contrast, *hCount* maintains the estimated frequencies for all items. On average, subtractions in *TiTiCount-s +* are not feasible for 38% of the cases.

In the next experiment, we focus on the batch size, *b*, and how this parameter influences performance. We report recall and precision for test cases 2, 5, 8 of Table 3 (we observed similar results with the rest of the test cases, too). We refer to these test cases as *Low*, *Mid* and *High*, respectively, to underline the fact that they come from streams of varying history size. Fig. 17(a) and (b) show averaged F_1 scores when run over a dataset with a stationary distribution, while varying the batch size *b* in the range [200, 10,000]. Fig. 17(c) and (d) depict the results of the same experiment when run with a time-variant distribution. As

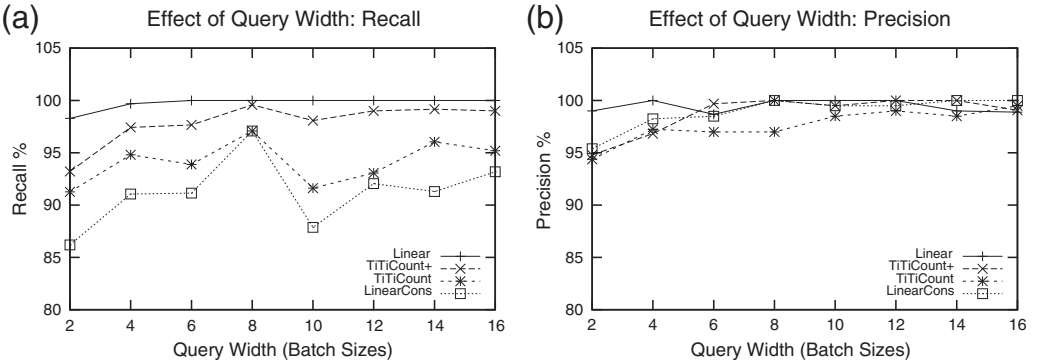

Fig. 19. Performance with respect to query width (dataset: kosarak, batch size $b = 1000$).

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

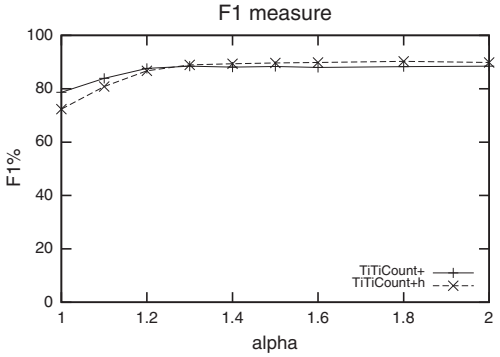

Fig. 20. Comparison of *TiTiCount +* and *TiTiCount+h* (2 buckets per histogram) algorithms varying the underlying Zipf distribution.

expected, when the distribution changes the overall accuracy gets lower because the problem becomes harder. Interestingly, accuracy is not strongly influenced by the batch size b. This is due to the characteristics of the tilted timeframe model: there is a tradeoff between the accuracy we gain by reducing the batch size and the accuracy cost due to the more frequent shifts towards larger (and less accurate) windows.

In the following experiment, we test 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 $b = 1000$, and number of windows $K = 11$. This means that our algorithms can answer queries 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 is the *Linear* algorithm that is given the same amount of space as our algorithms (resulting in a window size of 100,000).

Fig. 18 depicts the results of the experiment using a synthetic dataset and 120 randomly generated queries following a Gaussian distribution (mean $9n/10$, std dev $n/8$). The graphs show that *TiTiCount +* outperforms *TiTiCount* across the entire range of query sizes. As expected, *LinearConst* performs the worst (its somewhat high precision numbers are explained by the low performances 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.

We also run the same experiment on the *kosarak* dataset (refer to Fig. 19). Once again, the performance of *TiTiCount +* is superior to *TiTiCount*, and very close to the performance of *Linear*. (Similar results were obtained with the *retail* dataset, as well.)

We continue comparing the *TiTiCount +* and *TiTiCount+h* algorithms in terms of F_1 score, varying the Zipf α parameter in the interval [1,2]. In Figs. 20, 21 and 22, we show the average F_1 score over all test cases listed in Table 3 using respectively 2, 4 and 8 buckets per histogram. We observe that the highest accuracy is obtained with 4 buckets (and 4 frequency levels) and *TiTiCount+h* performs consistently better than *TiTiCount +* for values of α larger than 1.2.

We move now to the comparison of the *TiTiCount-s +* and *TiTiCount +* algorithms on four real datasets. Results are reported in Fig. 23. For datasets *NASA* and *Q148*, there is nearly no difference between *TiTiCount-s +* and *TiTiCount +* and the average F_1 score of both algorithms is above 95%. For datasets *retail* and *kosarak*, *TiTiCount-s +* performs consistently better. On average, the *F1* score of *TiTiCount +* is 81% and the *F1* score of *TiTiCount-s +* is 97%.

Finally, we compare the *TiTiCount +* and *TiTiCount+h* algorithms. While for most of the datasets *TiTiCount +* performs better, *TiTiCount+h* proves to be superior on the *NASA* dataset (Fig. 24).

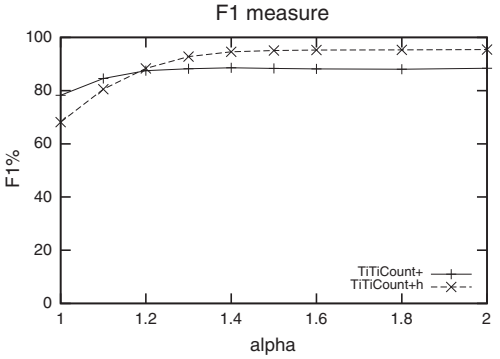

Fig. 21. Comparison of *TiTiCount +* and *TiTiCount+h* (4 buckets per histogram) algorithms varying the underlying Zipf distribution.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

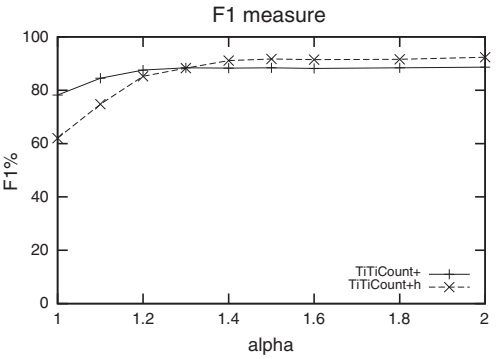

Fig. 22. Comparison of *TiTiCount +* and *TiTiCount+h* (8 buckets per histogram) algorithms varying the underlying Zipf distribution.

In Fig. 25 we report the averaged F_1 scores for $\alpha \in \{1.0,\ 1.2,\ 1.4,\ 1.6\}$ for time-varying distributions, where the distribution changes every βb items (recall that b is the batch size). We observe that, for α higher than 1.2, the *TiTiCount+h* algorithm performs consistently better than *TiTiCount +*. Thus, β has little impact on the conclusions drawn from Fig. 21.

In summary, the above results show that for very skewed frequency distributions, as in the case of Zipf distributions with a high α parameter value, the problem of identifying frequent items becomes easier, because the frequent items are more emphasized. Moreover, when using the *hCount* sketches collisions occur less often. Over a certain threshold ($\alpha = 1.2$ in our configuration), the benefit of using all the available memory to maintain *hCount* sketches is smaller than using part of it to maintain the histograms, and the algorithm of choice is *TiTiCount+h*.

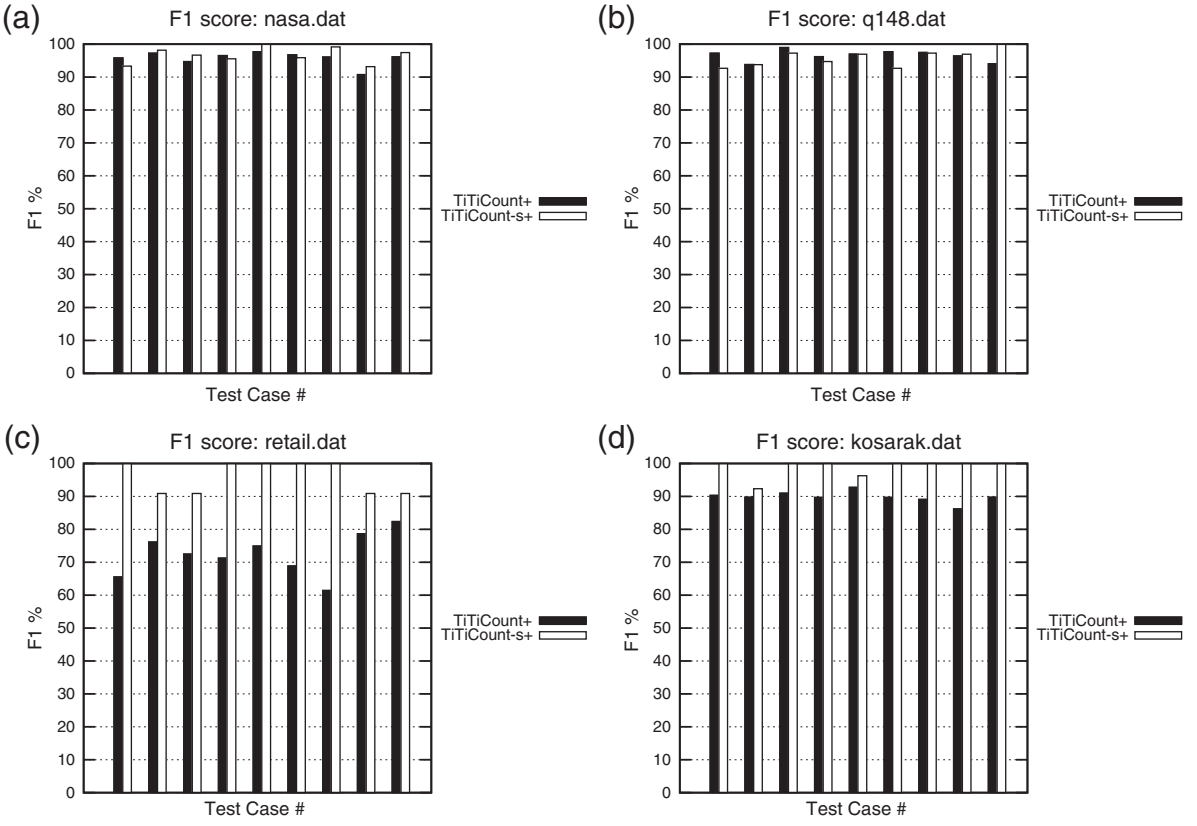

Fig. 23. Comparison of *TiTiCount +* and *TiTiCount-s +* algorithms in terms of F_1 score, real datasets.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

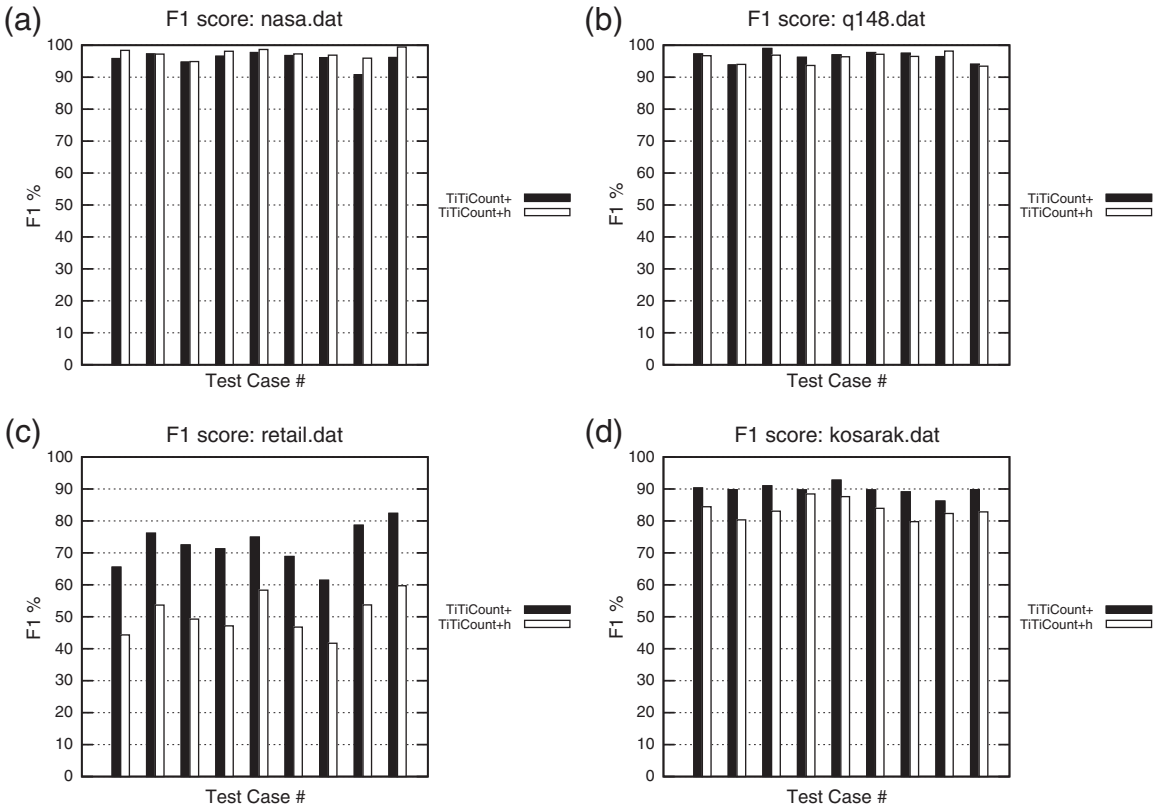

Fig. 24. Comparison of *TiTiCount +*, *TiTiCount +h* algorithms in terms of F_1 score, real datasets.

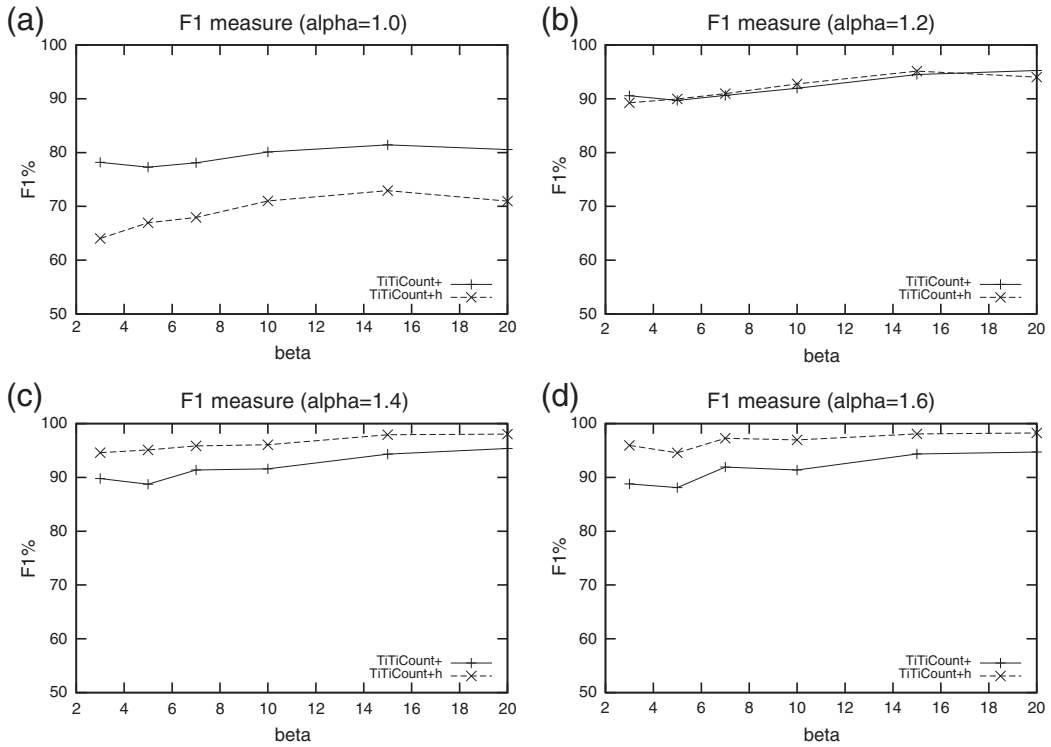

Fig. 25. Comparison of *TiTiCount +* and *TiTiCount +h* algorithms in terms of F_1 score for time-varying distributions, and varying the underlying Zipf distribution.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

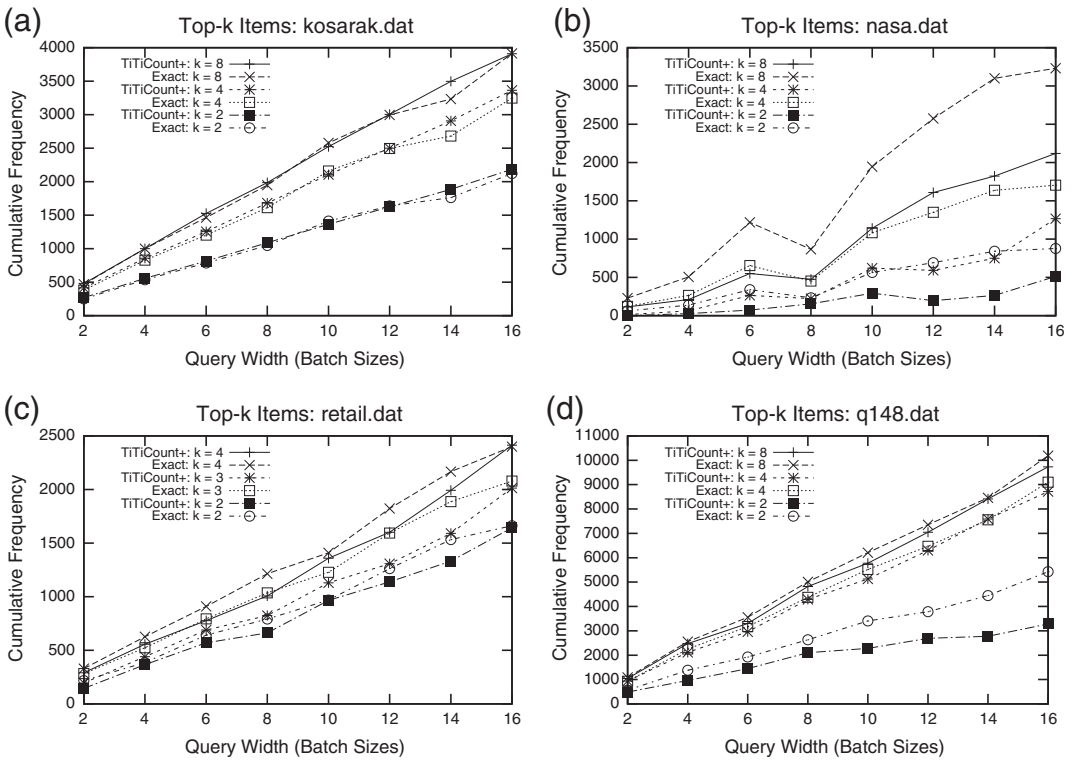

Fig. 26. Top-*k* items: Estimated and actual cumulative frequencies for *TiTiCount*+.

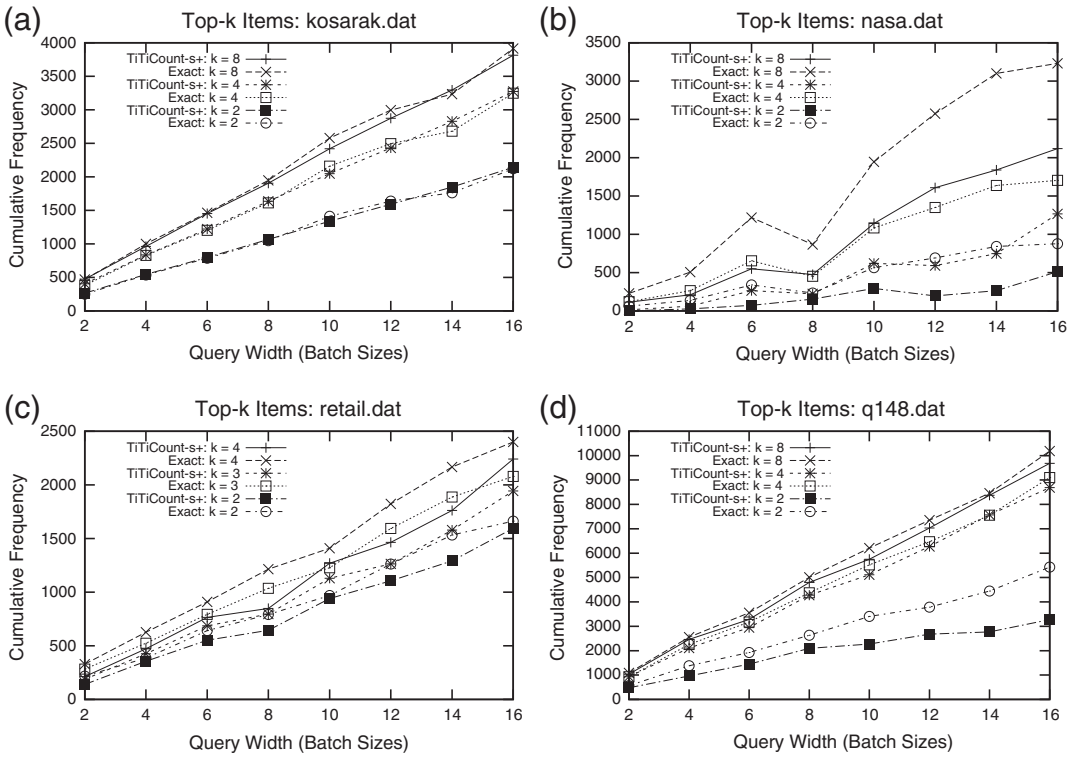

Fig. 27. Top-*k* items: Estimated and actual cumulative frequencies for *TiTiCount*-s+.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

6.2. 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 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 Fig. 26 for all 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 for *retail* and *kosarak* datasets (less than 0.05% error), while performing worse on *Q148* and *NASA* datasets.

The results for *TiTiCount-s +* and *TiTiCount+h* are very similar, and are shown respectively in Figs. 27 and 28.

We observe that *TiTiCount+h* is more accurate than *TiTiCount +* and *TiTiCount-s +* for the *retail* dataset.

6.3. Scalability

In order to evaluate the scalability of the proposed algorithms, we ran experiments to measure the update times of *TiTiCount*, *TiTiCount +*, *TiTiCount-s*, *TiTiCount-s +* and *TiTiCount+h*. 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 Fig. 29. The reported times are averages over five independent runs.

The results show that all algorithms scale linearly with the number of transactions, with *counter*-based variants being slightly less efficient, because of the higher cost for maintaining the *SS* internal structures. Similarly, *TiTiCount +* is slightly less efficient than *TiTiCount*, because of the higher cost of the shift operation that it implements, and the same is true for *TiTiCount+h* due to the increased cost of updating the histogram structures. Nevertheless, this small time-performance penalty is outweighed by their increased performance in terms of accuracy.

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 *ad hoc* windows in the stream history, rather than over the entire history or the recent past. For example, for a supermarket transaction stream, the set of frequent seasonal items would be a much more accurate description of the trends in consumer

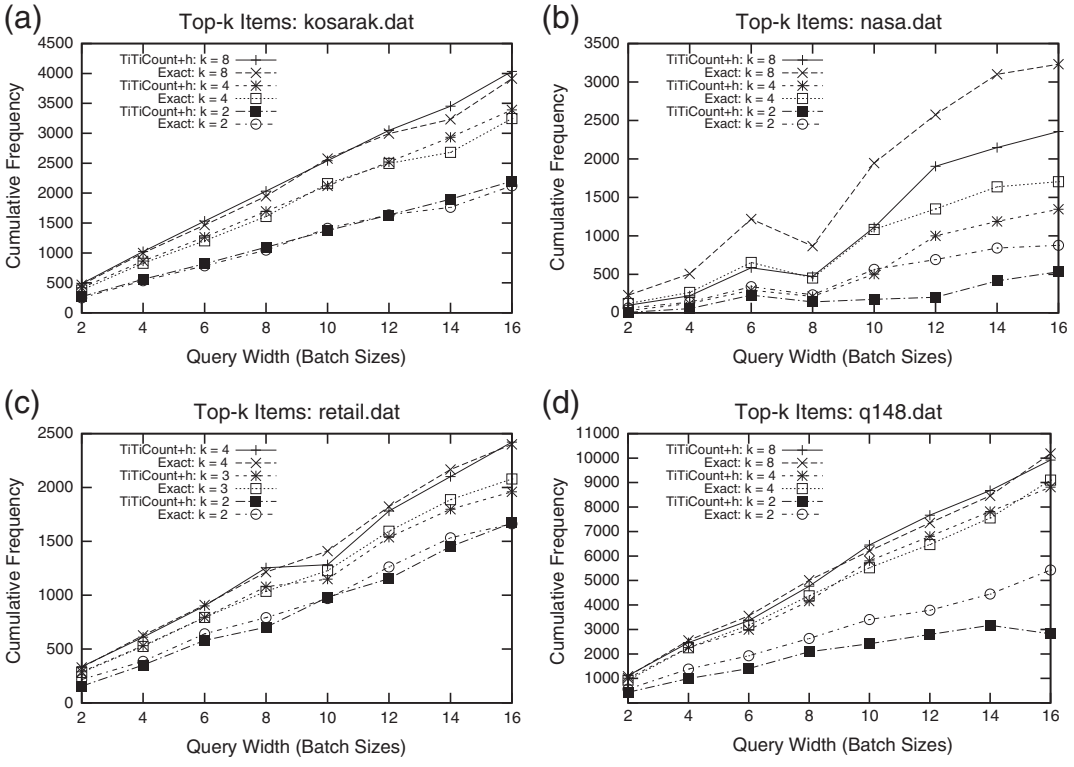

Fig. 28. Top-k items: Estimated and actual cumulative frequencies for *TiTiCount+*h.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

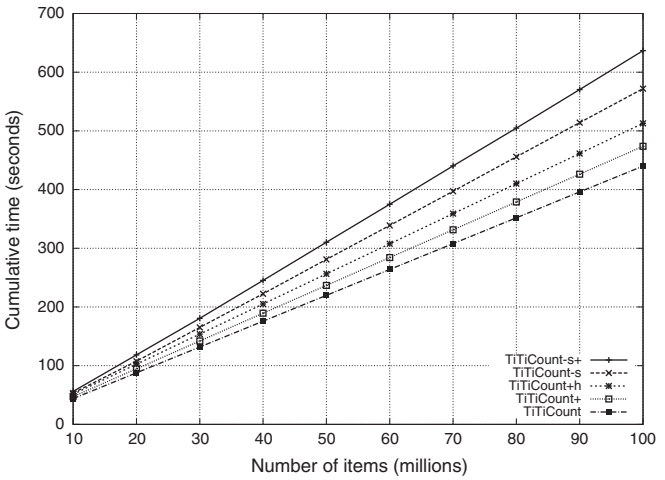

Fig. 29. Scalability: Variation in update time with increasing number of data items.

preferences. For an Internet click-stream, frequent items in an *ad hoc* windows would indicate the most interesting topics of interest on a website during a particular event.

In this paper, we propose novel algorithms for the discovery of frequent items in *ad hoc* windows in a data stream. The proposed algorithms are based on *sketching* and *counter*-based techniques, and are very flexible in that they are designed to answer queries for frequent items in *ad hoc* window intervals in the data stream history. This functionality makes possible the *post mortem* analysis of the streaming data at various time periods in the past. Based on our observations, we 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. Furthermore, we extend the timeframe windows adding concise, bitwise histograms to support non-linear frequency estimation.

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. Based on the experimental results, *counter*-based techniques provide higher accuracy in presence of stationary data distribution. When the data distribution is skewed, the *TiTiCount+h* algorithm leads to a significant improvement of the accuracy (for the same memory cost).

As future work we plan to investigate when it is beneficial to use histograms, and how to store them more efficiently. A more compact representation of the histograms would free some memory budget, to be exploited by other data structures used by the overall algorithm. We will also consider possible extensions of the proposed approach to itemsets and items with uncertainty.

References

[1] Frequent Itemset Mining Dataset Repository, University of Helsinki 2008. (http://fimi.cs.helsinki.fi/data/).
[2] Massive Data Analysis Lab, Rutgers University 2008. (http://www.cs.rutgers.edu/muthu/massdal.html).
[3] C.C. Aggarwal, Data Streams: Models and Algorithms, vol. 31, Springer Science + Business Media2007.
[4] C.C. Aggarwal, J. Han, J. Wang, P.S. Yu, A Framework for Clustering Evolving Data Streams, VLDB, 2003, pp. 81–92.
[5] T. Brijs, G. Swinnen, K. Vanhoof, G. Wets, Using association rules for product assortment decisions: a case study, Knowledge Discovery and Data Mining, 1999, pp. 254–260.
[6] F. Buccafurri, G. Lax, Approximating sliding windows by cyclic tree-like histograms for efficient range queries, Data & Knowledge Engineering 69 (9) (2010) 979–997.
[7] A. Bulut, A.K. Singh, SWAT: hierarchical stream summarization in large networks, ICDE, 2003, pp. 303–314.
[8] C.-H. Chang, S.-H. Yang, Enhancing SWF for incremental association mining by itemset maintenance, PAKDD, 2003, pp. 301–312.
[9] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, KDD '03: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, New York, NY, USA, 2003, pp. 487–492.
[10] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Springer-Verlag, London, UK, 2002, pp. 693–703.
[11] Y. Chen, G. Dong, J. Han, B.W. Wah, J. Wang, Multi-dimensional regression analysis of time-series data streams, VLDB, 2002, pp. 323–334.
[12] J. Cheng, Y. Ke, W. Ng, A survey on algorithms for mining frequent itemsets over data streams, Knowledge and Information Systems 16 (1) (2008) 1–27.
[13] D.W.-L. Cheung, J. Han, V.T.Y. Ng, C.Y. Wong, Maintenance of discovered association rules in large databases: an incremental updating technique, ICDE, 1996, pp. 106–114.
[14] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, Journal of Algorithms 55 (1) (2005) 58–75.
[15] G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, ACM Transactions on Database Systems 30 (1) (2005) 249–278.
[16] M. Deypir, M. Sadreddini, S. Hashemi, Towards a variable size sliding window model for frequent itemset mining over data streams, Computers and Industrial Engineering 63 (1) (2012) 161–172.
[17] C. Estan, G. Varghese, New directions in traffic measurement and accounting, SIGCOMM, 2002, pp. 323–336.
[18] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J.D. Ullman, Computing iceberg queries efficiently, VLDB, 1998, pp. 299–310.
[19] C. Giannella, J. Han, J. Pei, X. Yan, P. Yu, Mining frequent patterns in data streams at multiple time granularities, NSF Workshop on Next Generation Data Mining, 2003.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007

[20] P.B. Gibbons, Y. Matias, Synopsis data structures for massive data sets, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1999.
[21] A.C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, Fast, small-space algorithms for approximate histogram maintenance, Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing, STOC, ACM, New York, NY, USA, 2002, pp. 389–398.
[22] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, VLDB, 2001, pp. 79–88.
[23] H.V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K.C. Sevcik, T. Suel, Optimal histograms with quality guarantees, in: A. Gupta, O. Shmueli, J. Widom (Eds.), VLDB, 1998, pp. 275–286.
[24] C. Jin, W. Qian, C. Sha, J.X. Yu, A. Zhou, Dynamically maintaining frequent items over a data stream, CIKM '03: Proceedings of the Twelfth International Conference on Information and Knowledge Management, ACM Press, New York, NY, USA, 2003, pp. 287–294.
[25] R.M. Karp, S. Shenker, C.H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags, ACM Transactions on Database Systems 28 (1) (2003) 51–55.
[26] R. Kohavi, C. Brodley, B. Frasca, L. Mason, Z. Zheng, KDD-Cup 2000 organizers' report: peeling the onion, ACM SIGKDD Explorations Newsletter 2 (2) (2000) 86–93.
[27] R. Kohavi, F.J. Provost, Applications of data mining to electronic commerce, Data Mining and Knowledge Discovery 5 (1/2) (2001) 5–10.
[28] C.-H. Lee, C.-R. Lin, M.-S. Chen, Sliding window filtering: an efficient method for incremental mining on a time-variant database, Information Systems 30 (3) (2005) 227–244.
[29] D. Lee, W. Lee, Finding maximal frequent itemsets over online data streams adaptively, ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining, IEEE Computer Society, Washington, DC, USA, 2005, pp. 266–273.
[30] H. Li, S. Lee, Mining frequent itemsets over data streams using efficient window sliding techniques, Expert Systems with Applications 36 (2) (2009) 1466–1477.
[31] C.-H. Lin, D.-Y. Chiu, Y.-H. Wu, A.L.P. Chen, Mining frequent itemsets from data streams with a time-sensitive sliding window, SDM, 2005.
[32] N. Manerikar, T. Palpanas, Frequent items in streaming data: an experimental evaluation of the state-of-the-art, Data & Knowledge Engineering 68 (4) (2009) 415–430.
[33] G.S. Manku, R. Motwani, Approximate Frequency Counts over Data Streams, 2002.
[34] A. Metwally, D. Agrawal, A.E. Abbadi, An integrated efficient solution for computing frequent and top-elements in data streams, ACM Transactions on Database Systems 31 (3) (2006) 1095–1133.
[35] K. Mirylenka, T. Palpanas, G. Cormode, D. Srivastava, Finding interesting correlations with conditional heavy hitters, ICDE, 2013.
[36] S. Muthukrishnan, Data streams: algorithms and applications, Foundations and Trends in Theoretical Computer Science 1 (2) (2005).
[37] D. Ness, NASA Voyager 2 Hourly Average Interplanetary Magnetic Field Data, 2001.
[38] T. Palpanas, M. Vlachos, E.J. Keogh, D. Gunopulos, Streaming time series summarization using user-defined amnesic functions, IEEE Transactions on Knowledge and Data Engineering 20 (7) (2008) 992–1006.
[39] T. Palpanas, M. Vlachos, E.J. Keogh, D. Gunopulos, W. Truppel, Online amnesic approximation of streaming time series, ICDE, 2004, pp. 338–349.
[40] O. Papapetrou, M. Garofalakis, A. Deligiannakis, Sketch-based querying of distributed sliding-window data streams, Proceedings of the VLDB Endowment 5 (10) (2012) 992–1003.
[41] F.I. Tantono, N. Manerikar, T. Palpanas, Efficiently discovering recent frequent items in data streams, SSDBM, 2008, pp. 222–239.
[42] H. Thanh Lam, T. Calders, Mining top-k frequent items in a data stream with flexible sliding windows, Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2010, pp. 283–292.
[43] A.T. Whitney, D. Shasha, Lots o' ticks: real-time high performance time series queries on billions of trades and quotes, SIGMOD Conference, 2001, p. 617.
[44] J.X. Yu, Z. Chong, H. Lu, A. Zhou, False positive or false negative: mining frequent itemsets from high speed transactional data streams, VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases, 2004, pp. 204–215.

Michele Dallachiesa is a PhD candidate at the University of Trento, Italy. He received the BS and MSc degrees in computer science from the University of Trento. He was also a Research Fellow at the machine Learning and Intelligent OptimizatioN (LION) Group and then Co-Founder of Reactive Search Srl. He visited as intern student the IBM T.J. Watson Research Center and the Qatar Computing Research Institute. He is currently a member of the Database and Information Management Group (dbTrento) at the University of Trento. He served as reviewer for prestigious international conferences including ICDE, VLDB, SIGMOD, DEBS, MDM, and for the Information Systems (IS) journal. His research interests are in processing and analyzing streaming data in real time, sketching of time series, data cleaning, modeling and processing of time series with uncertainty, and mining and visualization of network graphs.

Themis Palpanas is a professor of computer science at the University of Trento, Italy. He received the BS degree from the National Technical University of Athens, Greece, and the MSc and PhD degrees from the University of Toronto, Canada. Before joining the University of Trento, he worked at the IBM T.J. Watson Research Center. He has also been a Visiting Professor at the National University of Singapore, worked for the University of California, Riverside, and visited Microsoft Research and the IBM Almaden Research Center. His interests include data management, data analysis, streaming algorithms, and business process management. His research solutions have been implemented in worldleading commercial data management products and he is the author of five US patents, three of which are part of commercial products in multi-billion dollar markets. He is the recipient of two Best Paper awards (ICDE 2010 and ADAPTIVE 2009). He is a founding member of the Event Processing Technical Society, and is serving on the Editorial Advisory Board of the Information Systems Journal and as an Associate Editor in the Journal of Intelligent Data Analysis. He is a General Co-Chair for VLDB 2013, has served on the program committees of several top database and data mining conferences, including SIGMOD, VLDB, ICDE, KDD, and ICDM, and has been a member of the IBM Academy of Technology Study on Event Processing.

Please cite this article as: M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in *ad hoc* time windows, Data & Knowledge Engineering (2013), http://dx.doi.org/10.1016/j.datak.2013.05.007