Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2007)

Adaptive Frequency Counting over Bursty Data Streams

Bill Lin, Wai-Shing Ho, Ben Kao and Chun-Kit Chui
Department of Computer Science
The University of Hong Kong, Hong Kong.
e-mail: {hlin,wsho,kao,ckchui}@cs.hku.hk

***Abstract*— We investigate the problem of frequent itemset mining over a data stream with bursty traffic. In many modern applications, data arrives at a system as a continuous stream of transactions. In many cases, the arrvial rate of transactions fluctuates wildly. Traditional stream mining algorithms, such as Lossy Counting (LC), were generally designed to handle data streams with steady data arrival rates. We show that LC suffers significant loss of accuracy when the data stream is bursty. We propose the Adaptive Frequency Counting algorithm (AFC) to handle bursty data. AFC has a feedback mechanism that dynamically adjusts the mining speed to cope with the changing data arrival rate. Through extensive experiments, we show that AFC outperforms LC under bursty traffics in terms of the accuracy of the set of freqeunt itemsets.**

I. Introduction

Extracting *frequent itemsets* from transactional datasets is a crucial step in association analysis. It finds applications in many areas such as business decision support and direct marketing. For example, identifying groups of products that are frequently purchased together helps a company to formulate its marketing strategies. Traditional mining algorithms assume a finite dataset over which the algorithms are allowed to scan multiple times. In recent years, however, researchers have shifted their attention to mining data streams. A data stream is an unbounded continuous stream of data records. Example applications include network traffic monitoring system, web logs and click streams, and sensor networks, etc. An important property that distinguishes data stream mining from traditional data mining is the unbounded nature of stream data, which precludes multiple-scan algorithms. Traditional frequent itemsets mining algorithms are thus not applicable.

Besides an unbounded size, there are other properties of a data stream that makes stream mining a challenging task. Let us consider a network monitoring system that identifies Denial-of-Service (DOS) attacks as an example [2]. In such an application, network activities are recorded continuously and records of such activities are sent to a monitoring system as a data stream. We observe the following properties of the system.

- Large volumes of data. There could be hundreds of millions of events that occur in a network each day. The number of network activity records is huge. It is thus expensive to store those records for offline analysis.
- Real-time processing. Standing queries are commons in a stream-processing system. Hence, the system should process the data in real-time. For example, network activities should be tracked and potential attacks should be detected in real-time.
- Bursty traffic. The traffic rate in a network varies significantly over a day. Sharp rises in traffic volume within a short period of time may happen unpredictably.
- Data aging. In some cases, recent data are more important than old ones. Old data should therefore be discounted or given a smaller weight of importance in data analysis.

Therefore, a system that analyzes data stream should: (i) be able to summarize data by a selected set of key statistics as the complete data are usually not available for analysis, (ii) work incrementally and avoid multiple scans of the data, (iii) provide an accuracy guarantee on the results if only an approximate solution is possible due to the unavailability of the complete dataset, (iv) handle data bursts gracefully without a significant performance penalty, and (v) discount the effect of old data. Previous solutions to the problem of mining frequent itemsets from data streams satisfy only some of the above five requirements. In particular, there are relatively few works that address bursty traffic handling. The goal of this paper is to provide a solution to the mining problem with a focus on a bursty data stream environment.

A. Related Work

Lossy Counting [10] (LC) was the first algorithm for finding frequent itemsets from a data stream. LC is a one-pass algorithm that provides an accuracy guarantee on the set of frequent itemsets it reports and their associated support counts. Basically, given a user-specified support threshold ρ_s and an error threshold ϵ, LC guarantees that any itemset that is not reported as frequent by the algorithm cannot has a support greater than $\rho_s - \epsilon$. Also, the error of the estimated support count of a reported frequent itemset cannot exceed ϵN, where N is the total number of transactions processed. LC satisfies the first three requirements we mentioned previously. However, LC has two main drawbacks. First, LC does not handle data bursts well. As we will see later, the choice of the error bound ϵ critically affects LC's performance in terms of processing speed and accuracy. Specifically, a small ϵ gives high accuracy

This research is supported by Hong Kong Research Grants Council Grant HKU 7138/04E.

but low processing speed and high memory usage, while a large ϵ gives low accuracy but fast processing and low memory usage. The value of ϵ, however, has to be preset before the algorithm starts and cannot be changed afterwards. Given a bursty data stream, a system designer would have to set ϵ to a value that is large enough so that the system can cope with the highest traffic. This approach suffers from three disadvantages: (1) the highest data arrival rate is difficult to estimate, (2) low accuracy of the results, and (3) poor system utilization during low-traffic periods. The second problem of LC is that it does not discount the effect of old data. All transactions in the whole history of the data stream are given equal weight. This is undesirable in some applications.

There are other algorithms for mining frequent itemsets from data streams. These include [3], [4], [5], [6], [7], [8], [13]. Many of these works [3], [5], [7], [8] are similar to LC in that they allow *false positives* in the answer set, i.e., some infrequent itemsets could be reported in the result. Others, such as [6], [13], allow *false negatives*, i.e., some frequent itemsets are not reported. Most of these algorithms employ preset error thresholds and thus they suffer from the same problem as LC does in handling bursty data streams.

B. Contribution

In this paper we propose a flexible algorithm called Adaptive Frequency Counting (AFC) for mining frequent itemsets from bursty data streams. AFC is based on Lossy Counting with three improvements. First, the error threshold can be adjusted in the middle of the mining exercise. Second, AFC has a feedback mechanism that dynamically controls the processing speed to cope with a changing data arrival rate. Third, an aging data model is defined in AFC so that old data can be properly discounted. We have conducted extensive experiments evaluating the relative performance of AFC and LC. Our results show that AFC outperforms LC significantly in terms of its ability to provide accurate results under bursty data.

The rest of the paper is organized as follows. We review Lossy Counting, describe a data aging model and formally define the problem in Section II. Section III describes the AFC architecture and the design of a feedback mechanism, which is the core component of the AFC algorithm. Section IV describes the implementation of AFC. Section V reports our experimental results. Finally, Section VI concludes the paper.

II. Lossy Counting, Aging Model and Problem Definition

In this section we review the Lossy Counting algorithm, which inspired our AFC algorithm. We also describe a data aging model that uses a *decay factor* to discount old data in a data stream. Finally, we give a formal definition of our mining problem.

A. Preliminaries

Let I be a set of items and T be a stream of transactions such that each transaction is a subset of I. We consider the transactions in T being segmented into a sequence of buckets $T = T_1, T_2, \ldots$. Each bucket contains a number of transactions. Given an itemset $X \subseteq I$, the support count $\sigma_i(X)$ of X in bucket T_i is the number of transactions in T_i that contain X. Suppose n buckets $T_1, \ldots, T_n$ in T have arrived at the system, the support count of X with respect to T, $\sigma(X)$, is defined as the total number of transactions in T that contain X, i.e., $\sigma(X) = \sum_{i=1}^{n} \sigma_i(X)$. Given a user-specified support threshold ρ_s, X is a *frequent itemset* if $\sigma(X) \geq \rho_s N$ where N is the total number of transactions in the n buckets.

B. Lossy Counting

In [10], Manku and Motwani proposed the Lossy Counting (LC) algorithm, the first one-pass algorithm for counting approximately the set of frequent itemsets over a stream of transactions. Given a user-specified error bound ϵ, LC processes incoming data bucket-by-bucket and updates a summary structure D. D contains a set of entries of the form $\langle X, \hat{\sigma}(X), \delta(X) \rangle$, where X is an itemset, $\hat{\sigma}(X)$ is an approximate support count of X, and $\delta(X)$ is the maximum possible error in $\hat{\sigma}(X)$. The summary structure D has the following two properties:

P1. For each itemset X, if X is not in D, then $\sigma(X) \leq \epsilon N$, where N is the total number of transactions processed.

P2. For each entry $\langle X, \hat{\sigma}(X), \delta(X) \rangle$ in D, $\hat{\sigma}(X) \leq \sigma(X) \leq \hat{\sigma}(X) + \delta(X)$ and $\delta(X) \leq \epsilon N$.

From Property P1, we know that itemsets that are not present in D have very small supports and are thus not frequent. From Property P2, we know that the maximum possible support count of an itemset X cannot exceed $\hat{\sigma}(X) + \delta(X)$. If a user requests a set of frequent itemsets, LC reports all those itemsets X in D whose support upper bounds are not less than the support threshold, i.e., $\hat{\sigma}(X) + \delta(X) \geq \rho_s N$.

LC maintains the structure D by the following procedure. Initially, D is empty. Let us assume that LC has processed buckets $T_1, \ldots, T_{i-1}$ and has summarized the transactions in D. After the transactions of the next bucket T_i are collected, LC starts the process of updating D. Let N_1 denote the number of transactions processed before bucket T_i and N_2 denote the total number of transactions processed including those in T_i. We have $N_2 = N_1 + |T_i|$, where $|T_i|$ denotes the number of transactions in T_i. Lossy Counting enumerates itemsets that are present in T_i and counts their supports. Then for each itemset X, its support count $\sigma_i(X)$ in bucket T_i is determined and D is updated by the following rules:

- If D does not contain an entry for X, we create an entry $\langle X, \sigma_i(X), \epsilon N_1 \rangle$ if $\sigma_i(X) + \epsilon N_1 > \epsilon N_2$.
- Otherwise, X has an entry in D and the approximate support count $\hat{\sigma}(X)$ is incremented by $\sigma_i(X)$.
- If the updated entry satisfies $\hat{\sigma}(X) + \delta(X) \leq \epsilon N_2$, we delete the entry from D.

In this paper we use D_i to represent the data structure D right after bucket T_i is processed. It is proved in [10] that the set of frequent itemsets and their support counts reported by LC ensure the following guarantees:

- All itemsets whose true support counts exceed $\rho_s N$ are reported.
- No itemset whose true support count is less than $(\rho_s - \epsilon)N$ is reported.
- The difference between the reported support count $(\hat{\sigma}(X) + \delta(X))$ of an itemset X and the true support count $(\sigma(X))$ is at most ϵN.

C. Data Aging

In many data stream applications, recent data are more important than older ones [11]. To capture this, we use a decay factor $\alpha \in (0, 1]$ in calculating itemsets' support counts. We call such modified support counts *time-weighted support counts*. Again, transactions are divided into buckets T_i (i starts from 1). When the system has collected a new bucket of transactions, all itemsets' support counts decay by a factor of α. Let $\sigma_i(X)$ denote the support count of X in bucket T_i, and $\sigma_{1..k}(X)$ denote the time-weighted support count of X obtained from the first k buckets of the stream. We have,

$$\sigma_{1..k}(X) = \alpha \times \sigma_{1..(k-1)}(X) + \sigma_k(X) = \sum_{i=1}^{k} \alpha^{k-i}\sigma_i(X).$$

We modify the definition of frequent itemsets accordingly. Specifically, an itemset X is *frequent* with respect to a data stream $T = T_1T_2 \dots T_k$ of k buckets if

$$\sigma_{1..k}(X) \geq \rho_s \sum_{i=1}^{k} \alpha^{k-i}|T_i|.$$

D. Problem Definition

Given a data stream T with bursty transaction arrivals, a user-specified support threshold ρ_s and a decay factor α, the problem is to report all itemsets X and to estimate their support counts such that (i) all frequent itemsets in T are reported, and (ii) a maximum possible error is reported for each estimated support count. Due to the large volume of transactions, we also require that transactions be loaded into memory and processed only once.

III. System Architecture and Feedback Mechanism

As we mentioned in Section I, most frequent itemset mining algorithms are unable to adjust their mining speed and thus do not handle bursty traffic well. In this section we describe the architecture of AFC and explain in details the feedback mechanism AFC employs to cope with a bursty data stream.

A. Architecture

Figure 1 shows the architecture. AFC consists of three modules: the *buffer module*, the *mining module* and the *speed regulator*. The buffer module receives transactions from the data stream and put them into memory buffer. The memory buffer is divided into a number of buffer slots. Each slot can store one bucket[1] of transactions.

[1]The meaning of *bucket* here is different from that in LC [10]. In LC, one bucket contains $\lceil 1/\epsilon \rceil$ transactions. Here, one bucket refers to the transactions to be processed together, which is similar to one *batch* of buckets in LC.

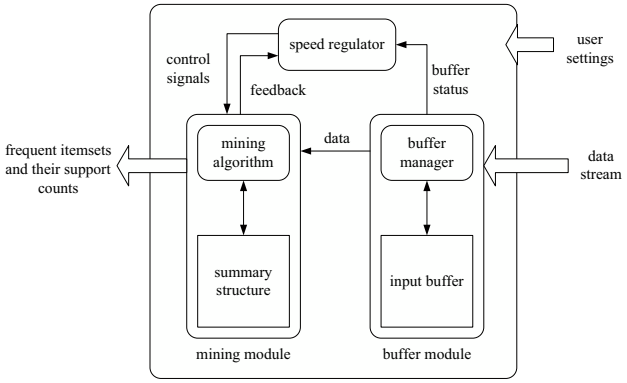

Fig. 1. Architecture of Adaptive Frequency Counting (AFC).

The buffer manager monitors the occupancy of the buffer slots and submits statistical information to the speed regulator to control the mining speed. The mining module processes each bucket of buffered data and summarizes the mining result in its internal summary structure. The mining module submits feedback statistics to the speed regulator after each bucket is processed. In our implementation of AFC, the mining module is based on Lossy Counting except that our implementation allows a flexible error threshold that can be dynamically adjusted. The speed regulator receives statistical information from the buffer manager to estimate the data arrival rate. It also receives feedback information from the mining module to determine the mining speed. Base on this information, the speed regulator determines a *target processing speed* and sends a speed control signal to the mining module.

B. Feedback Mechanism

The core component of AFC is the *feedback mechanism*, which is implemented in the speed regulator. In summary, the speed regulator learns about the changes in the buffer occupancy from the buffer manager to estimate the data arrival rate and the presence of data bursts. It estimates a *target processing time*, denoted by p_i, which is the amount of time within which the mining module should complete the processing of the next bucket of transactions (T_i). The time p_i is set based on the objective of bringing the buffer occupancy to a reasonably low level so that the system can gracefully receive an influx of transactions during data bursts. As we will explain later, the processing speed of the mining module depends on the error threshold ϵ. Unlike Lossy Counting, which employs a fixed ϵ, AFC dynamically adjusts ϵ to control the mining speed. More specifically, AFC assigns an error threshold ϵ_i for processing bucket T_i. Statistics obtained from the mining module is considered by the speed regulator to determine ϵ_i so that the amount of time needed to process T_i

is likely to be close to p_i, the target processing time of T_i. We will explain how AFC estimates p_i and how it determines ϵ_i in the following subsections.

C. Estimating Target Processing Time

We estimate the target processing time p_i of bucket T_i by considering the buffer occupancy and the processing time of the previous bucket T_{i-1}. Let Q be the number of buffer slots in the buffer module. (Each slot can contain one bucket of transactions.) The value of Q is determined by the available memory. Let q_i denote the number of buffer slots that are occupied just before bucket T_i is processed. The objective of the feedback mechanism is to try to maintain the buffer occupancy to a fraction f of the Q buffer slots. For example, if q_i is larger than the target buffer occupancy fQ, AFC should set p_i to a small value so that transactions are mined at a higher rate to bring the occupancy down towards fQ. The value of f is a system parameter. A larger f reduces the empty buffer slots that guard against potential data bursts, while a smaller f may cause buffer underflows. When buffer underflows, the system wait for incoming transactions instead of using the time to improve mining accuracy of buffered buckets. The user can tune f to get the best system performance.

To determine the processing time p_i for maintaining the buffer occupancy, let us consider the processing of bucket T_{i-1}. Let t_{i-1} be the time taken to process T_{i-1}. Within this period of time, the system has consumed one bucket (i.e., T_{i-1}) while the buffer occupancy has changed from q_{i-1} to q_i. Hence, the number of buckets that have arrived during this period is $1 + q_i - q_{i-1}$. Therefore, the bucket arrival rate is $\frac{(1+q_i-q_{i-1})}{t_{i-1}}$ buckets per unit time.

Now, if $q_i > fQ$ (i.e., the buffer occupancy is larger than the target occupancy right before the processing of bucket T_i), we need to increase the mining speed to bring the buffer occupancy down to fQ. Assume that we want to restore the target occupancy within a certain time period, $t_{restore}$, in which no more than k new buckets have arrived from the stream. During this period of time, we have k bucket arrivals, and the occupancy changes from q_i to fQ. Hence, the number of buckets processed is $q_i + k - fQ$. If we assume that buckets arrive at a constant rate during this period, we have

$$\begin{aligned} t_{restore} &= \frac{k}{\textit{bucket arrival rate}}, \\ &= \frac{k \cdot t_{i-1}}{1 + q_i - q_{i-1}}. \end{aligned}$$

Since the system has to process $q_i + k - fQ$ buckets within this period of time, the target processing time of T_i should be

$$\begin{aligned} p_i &= \frac{t_{restore}}{(q_i + k - fQ)}, \\ &= \frac{k \cdot t_{i-1}}{(1 + q_i - q_{i-1})(q_i + k - fQ)}. \end{aligned} \quad (1)$$

The value of p_i given by 1 is an estimation based on the current buffer status. This estimation is redone after each bucket is processed. The value of k is a system parameter. It controls the responsiveness of the system in maintaining the target buffer occupancy. A smaller value of k leads to a smaller restoration time ($t_{restore}$) and thus a higher processing speed is required at the mining module. As we will see later, this translates into a larger error threshold and thus less accurate result. In our implementation, k is a function of the buffer space available. A smaller number of empty buffers implies a more critical state that calls for a faster reaction in order to avoid buffer overflow. Hence, a smaller k is used. Figure 2 shows two possible functions for determining k given q_i.

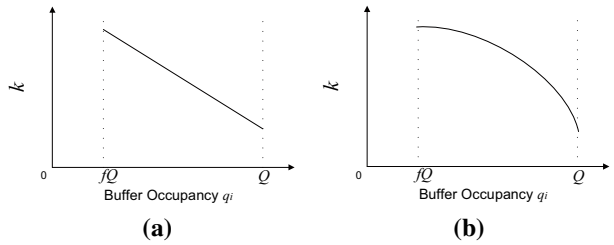

Fig. 2. Two sample functions connecting k and q_i.

D. Speed Control

In order to control the processing speed of the mining module, we need to identify the parameters that affect the processing speed of the mining algorithm. In Lossy Counting, the key factor that determines the processing speed is the error threshold ϵ. In AFC, the algorithm determines a suitable error threshold ϵ_i for processing bucket T_i in order that the target processing time p_i is achieved.

In order to determine the relationship between ϵ_i and p_i, we have performed both an analytical study and an empirical study on the Lossy Counting algorithm. We argue that the processing time of a bucket T_i is linearly proportional to the following quantity

$$C_i = \sum_{X \in D_i} \sigma_i(X).$$

That is the sum of all the support counts in T_i of the itemsets that are retained in the summary structure D_i after the bucket T_i is processed[2]. Moreover, we can discover the relationship between an ϵ_i and this sum C_i. Recall that when the data structure D is updated, the entry for an itemset X is removed from D if the maximum possible support count of X is not greater than ϵN, where ϵ is the error threshold and N is the total number of transactions processed. As a result, the larger ϵ_i (i.e., the error threshold used when processing T_i) is, the fewer itemsets will be kept in D_i, leading to a smaller C_i. These two pieces of information together allow us to determine the value of ϵ_i given a target processing time p_i. We now describe a two-step approach for determining ϵ_i.

[2]Recall that D_i represents the data structure D after bucket T_i is processed.

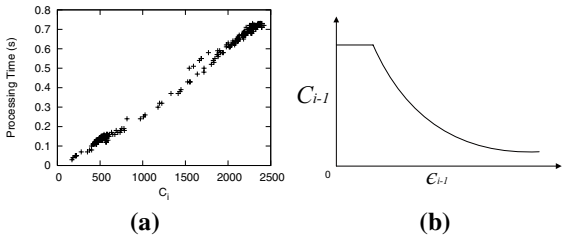

Fig. 3. The relationships between (a) bucket processing time and C_i, (b) C_{i-1} and ϵ_{i-1}.

1) Step 1. Estimate a target value of C_i from p_i: We conducted an experiment to verify our hypothesis of a linear relationship between the time to process a bucket and C_i. In the experiment, 300 buckets of transactions simulating a data stream are processed by Lossy Counting. We recorded the processing time of each bucket T_i and the corresponding value of C_i. These 300 pairs of values are plotted in Figure 3(a). From the figure, we see a linear relationship between the two quantities.

Now, when AFC is applied to mine a real data stream, the mining module will be instructed by the speed regulator to finish the processing of a bucket T_i within a target processing time p_i. Assume that the mining module has recorded the amount of time (t_{i-1}) it took to process bucket T_{i-1}, and the value C_{i-1}, we can estimate that in order to limit the processing time of T_i to p_i, the target value of C_i should be given by

$$C_i = \frac{C_{i-1}}{t_{i-1}} \times p_i, \tag{2}$$

due to the linear relationship as illustrated by Figure 3(a).

What remains to be solved is to investigate how we can control C_i by choosing the right error threshold ϵ_i.

2) Step 2. Estimate ϵ_i from C_i: As we have explained, a larger ϵ_i leads to more entries removed from D and thus a smaller C_i. To capture the relationship between the two quantities, we construct a histogram based on the statistics obtained from processing the previous bucket (T_{i-1}). More specifically, after bucket T_{i-1} is processed, AFC obtained the structure D_{i-1}. We note that the value of ϵ_{i-1} determines the entries in D_{i-1} and thus the value of C_{i-1}. Conceptually, based on D_{i-1}, we can calculate the values of C_{i-1} if ϵ_{i-1} were to assume different values. We thus plot a curve showing the (hypothetical) relationship between C_{i-1} and ϵ_{i-1}. Figure 3(b) shows the general shape of such a curve. We then use the curve to approximate the relationship between C_i and ϵ_i. Hence, from the curve, we are able to estimate an appropriate value of ϵ_i given C_i.

Here, we summarize our approach to adjust the mining speed of the mining module in case the buffer occupancy has exceeded the target occupancy level. AFC constructs a curve relating C_{i-1} and ϵ_{i-1} after it processes a bucket T_{i-1}. It also records the processing time t_{i-1} of bucket T_{i-1} and the value of C_{i-1} is calculated from D_{i-1}. Right before bucket T_i is processed, if $q_i > fQ$, AFC determines a target processing time (p_i) based on Equation 1. It then calculates a target value of C_i by Equation 2. Using the $C_{i-1}\epsilon_{i-1}$ curve (Figure 3(b)), AFC determines the value of ϵ_i given the estimated value of C_i. This error threshold is then applied when the mining module processes bucket T_i.

Finally, if the buffer occupancy drops below the target occupancy (i.e., $q_i < fQ$) during low traffic periods, the system should lower the error threshold in order to provide a more accurate result. In our implementation, AFC reduces ϵ by a constant fraction (i.e., $\epsilon_i = \mu\epsilon_{i-1}$ for some $0 < \mu < 1$).

IV. Algorithms and Implementation

We present the algorithms implemented in the AFC system model in this section. Similar to LC, AFC maintains a summary data structure D_i after each bucket T_i is processed. D_i contains a number of entries, each of the form $\langle X, \hat{\sigma}_{1..i}(X), \delta_{1..i}(X)\rangle$. Associated with each D_i are the error threshold ϵ_i and the number of transactions processed (aged), denote as N^i_{age}. More specifically, $N^i_{age} = \sum_{j=1}^{i} \alpha^{i-j}|T_j|$ is the total number of transactions processed through buckets $T_1 \ldots T_i$ weighted due to data aging (see Section II-C). AFC guarantees that the true time-weighted support count of an itemset X, $\sigma_{1..i}(X)$, is bounded by $\hat{\sigma}_{1..i}(X) \leq \sigma_{1..i}(X) \leq \hat{\sigma}_{1..i}(X) + \delta_{1..i}(X)$. It also guarantees that if an itemset X is not in D_i, then $\sigma_{1..i}(X) \leq \epsilon_i N^i_{age}$.

If AFC is requested by a user to report the mining result after bucket T_i is processed, it returns a set of itemsets X such that (i) X has an entry in D_i and (ii) $\hat{\sigma}_{1..i}(X) + \delta_{1..i}(X) \geq \rho_s N^i_{age}$. AFC provides the following accuracy guarantees:

- All itemsets whose true time-weighted support counts exceed $\rho_s N^i_{age}$ are returned.
- No itemset whose true time-weighted support count is less than $(\rho_s - \epsilon_i)N^i_{age}$ is returned.
- If no buckets are dropped in the processing of the data stream, then the difference between the reported support count $(\hat{\sigma}_{1..i}(X) + \delta_{1..i}(X))$ of an itemset X and the true time-weighted support count of X is at most $\epsilon_i N^i_{age}$.

The AFC algorithm includes two smaller algorithms: the **UpdateD** algorithm is a modified version of Lossy Counting that maintains the summary data structure D and, the **Feedback** algorithm implements the feedback mechanism discussed in Section III-B. Like LC, UpdateD is a one-pass algorithm which processes data one bucket at a time. It differs from Lossy Counting in that it handles data aging and that it allows the error threshold to be dynamically adjusted to control the mining speed. Despite this flexibility, UpdateD ensures that the accuracy guarantees mentioned above are maintained. Due to space limitation, readers are referred to [9] for the proofs of the accuracy guarantees. In addition, this proof also shows that LC can maintain its accuracy guarantee with data aging. This is because LC can be treated as a special case of AFC in which ϵ is a constant over all buckets.

The AFC algorithm invokes UpdateD whenever a bucket T_i is to be processed. The UpdateD algorithm takes the following parameters:

Algorithm UpdateD
INPUT: T_i, D_{i-1}, ϵ_{i-1}, N_{age}^{i-1} and α.
OUTPUT: The updated summary structure D_i.
1: **for all** entries $\langle X, \hat{\sigma}_{1..i-1}(X), \delta_{1..i-1}(X)\rangle \in D_{i-1}$ **do**
2: $\quad\hat{\sigma}_{1..i-1}(X) \leftarrow \hat{\sigma}_{1..i-1}(X)\alpha$.
3: $\quad\delta_{1..i-1}(X) \leftarrow \delta_{1..i-1}(X)\alpha$.
4: **end for**
5: $N_{age}^{i-1} = N_{age}^{i-1}\alpha$.
6: $N_{age}^{i} = N_{age}^{i-1} + |T_i|$.
7: **if** T_i is dropped **then**
8: $\quad$**for all** entries $\langle X, \hat{\sigma}_{1..i-1}(X), \delta_{1..i-1}(X)\rangle \in D_{i-1}$ **do**
9: $\quad\quad$Create $\langle X, \hat{\sigma}_{1..i-1}(X), \delta_{1..i-1}(X) + |T_i|\rangle$ in D_i.
10: $\quad$**end for**
11: $\quad\epsilon_i = \frac{\epsilon_{i-1}N_{age}^{i-1}+|T_i|}{N_{age}^{i}}$.
12: **else**
13: $\quad\epsilon_i$ = FEEDBACK().
14: $\quad$**for all** Itemset X **do**
15: $\quad\quad$Compute $\sigma_i(X)$.
16: $\quad\quad$**if** $X \notin D_{i-1}$ **then**
17: $\quad\quad\quad$**if** $\sigma_i(X) > \epsilon_i N_{age}^{i} - \epsilon_{i-1}N_{age}^{i-1}$ **then**
18: $\quad\quad\quad\quad$Create $\langle X, \sigma_i(X), \epsilon_{i-1}N_{age}^{i-1}\rangle$ in D_i.
19: $\quad\quad\quad$**end if**
20: $\quad\quad$**else**
21: $\quad\quad\quad$**if** $\hat{\sigma}_{1..i-1}(X) + \sigma_i(X) + \delta_{1..i-1}(X) > \epsilon_i N_{age}^{i}$ **then**
22: $\quad\quad\quad\quad$Create $\langle X, \hat{\sigma}_{1..i-1}(X) + \sigma_i(X), \delta_{1..i-1}(X)\rangle$ in D_i.
23: $\quad\quad\quad$**end if**
24: $\quad\quad$**end if**
25: $\quad\quad$Update H_i with $\sigma_i(X)$.
26: $\quad$**end for**
27: **end if**

Fig. 4. The algorithm UpdateD.

- T_i: The current bucket of transactions.
- D_{i-1}: The summary structure covering all previous transactions processed from bucket T_1 to bucket T_{i-1}.
- ϵ_{i-1}: The error threshold after mining the bucket T_{i-1}.
- N_{age}^{i-1}: The aged number of transactions processed from bucket T_1 to bucket T_{i-1}.
- α: The decay factor.

Figure 4 shows the UpdateD algorithm. UpdateD first decays the counts and errors in the old summary structure by α (Lines 1 to 6). If bucket T_i has been dropped by the system, for example, due to buffer overflow, T_i will be a null bucket (Lines 7-11). In that case, we increase the error bound of each itemset in D_i by $|T_i|$ and set $\epsilon_i = \frac{\epsilon_{i-1}N_{age}^{i-1}+|T_i|}{N_{age}^{i}}$. If bucket T_i is not dropped, we proceed to counting itemsets' supports in T_i (Lines 12 to 27). Support counting here is similar to that of Lossy Counting. The only difference is that the error threshold ϵ_i for mining the bucket T_i is dynamically determined by the Feedback algorithm (Line 13).

The Feedback algorithm determines an error threshold ϵ_i for mining T_i according to the feedback mechanism mentioned in Section III-B. It considers the following information:

- q_{i-1}, q_i: The buffer occupancies before and after the processing of bucket T_{i-1}, respectively.
- fQ: Target buffer occupancy.
- t_{i-1}: Processing time of bucket T_{i-1}.
- H_{i-1}: The histogram showing the hypothetical relationship between C_{i-1} and ϵ_{i-1} (see Figure 3(b)).
- C_{i-1}: The sum of all support counts of itemsets that occur in D_{i-1}, i.e., $C_{i-1} = \sum_{X \in D_{i-1}} \sigma_{i-1}(X)$.

Algorithm FEEDBACK
INPUT: $q_i, q_{i-1}, fQ, t_{i-1}, H_{i-1}$, and C_{i-1}.
OUTPUT: The estimated error threshold ϵ_i.
1: Get q_i, q_{i-1} from the buffer module.
2: Get t_{i-1}, C_{i-1} and H_{i-1} from the mining module.
3: Determine k given q_i and fQ.
4: Compute target processing time

$$p_i = \frac{k \cdot t_{i-1}}{(1 + q_i - q_{i-1})(q_i + k - fQ)}.$$

5: Compute C_i given p_i.

$$C_i = \frac{C_{i-1}}{t_{i-1}} \times p_i.$$

6: Consult H_{i-1} for the estimated value of ϵ_i given C_i.
7: Return ϵ_i.

Fig. 5. The algorithm FeedBack.

V. EXPERIMENTS

We evaluated AFC and compared its performance against Lossy Counting (LC) through extensive experiments. We focus on the algorithms' abilities in handling bursty data streams. As we have explained in our discussion. One disadvantage of LC is that the error threshold has to be preset. On the hand, AFC has the ability to dynamically adjust the error threshold to cope with the changing data arrival rate. We will illustrate this adaptability by showing how the error threshold is updated by AFC given a range of different initial value of the error threshold.

A. Experiment Settings

We evaluated AFC and LC by applying them to a set of synthetic data streams. The synthetic data streams were created in two steps. First, we used the IBM synthetic data generator [1] to generate a sequence of transactions. We have conducted the experiments on a number of transaction sequences. For illustration purpose, we show the performance result obtained from the data sequence generated by the dataset T10.I4.30M (i.e., the average size of a transaction = 10 items, the average size of a frequent itemset = 4, 30 million transactions in the dataset). The items were drawn from a universe of 10,000 items.

Next, the transactions were divided into 300 equal-sized buckets, each with 100,000 transactions. Each bucket is given a timestamp specifying the arrival time of the transactions in the bucket. To simulate a bursty data stream, bucket arrival times are distributed unevenly over a time period. A data burst is thus simulated by a relative large number of bucket arrivals during a short time segment. We adopted the *b-model* [12] to obtain such an uneven bucket arrivals distribution.

The b-model models the burstiness of a data stream using a *bias factor* b. Given a time period d over which transactions are distributed, we first divide d into two equal-length segments. Then, a fraction b of all the transactions are assigned to one segment, while the rest are assigned to the other segment. This assignment is performed recursively for 4 levels. Hence, the time period d is divided into 16 equal-length segments, each is assigned a certain number of buckets. For each time segment, the buckets assigned to it are evenly distributed

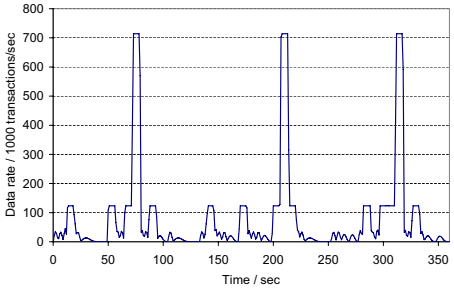

Fig. 6. Date rate over different time in a sample stream.

over the segment's time period. Note that this method assigns transactions to the segments unevenly. One segment is given the largest number of transactions while 8 segments have the fewest transactions. For example, if $b = 0.85$, the data rate of the segment with the highest data rate is $b^4/(1-b)^4 \approx 1031$ times higher than that of the segment with the lowest data rate. In our experiments we varied b from 0.6 to 0.85 to simulate different levels of burstiness.

Note that each application of the b model results in only one segment out of 16 that has the highest data rate, i.e., there is only one sharp burst in the stream. In order to simulate a data stream with more than one sharp burst, we concatenated the data sequences generated by 3 separate runs of the data generation to construct a data stream. Figure 6 shows the transaction arrival rate of one such data stream. This data stream was generated with $b = 0.85$ and $d = 360$ seconds.

We implemented both LC and AFC using the C programming language. The experiments were conducted on a machine with a 2.6GHz P4 CPU and 512MB RAM operating on Linux Kernel 2.6.10.

B. Comparison with Lossy Counting

Since both LC and AFC report only an *approximation* of the set of frequent itemsets and their support counts, *accuracy* is an important performance measure of the algorithms. We evaluated the accuracy of LC and AFC over three metrics, *recall*, *precision* and *average relative count error*. *Recall* is the fraction of actual frequent itemsets that are reported by an algorithm. *Precision* is the fraction of the reported frequent itemsets that are actually frequent. Note that in both LC and AFC, the reported count of an itemset X is $\hat{\sigma}X + \delta(X)$[3]. The relative count error for X is defined as $\frac{(\hat{\sigma}(X)+\delta(X))-\sigma(X)}{\sigma(X)}$, where $\sigma(X)$ is the actual support count of X.

In our experiment, there were 300 buckets in a data stream. We queried both LC and AFC to obtain their mining results after each bucket was processed. We thus compared the performance of LC and AFC with respect to the three accuracy metrics at 300 different instants. Figures 7(a), 7(b) and 7(c) clear show the impact of data bursts on LC and AFC to their performance. We have conducted experiments on other data streams, similar results were obtained and those results are not reported here due to space limitation.

[3]The values of $\hat{\sigma}(X)$ and $\delta(X)$ are stored in the structure D.

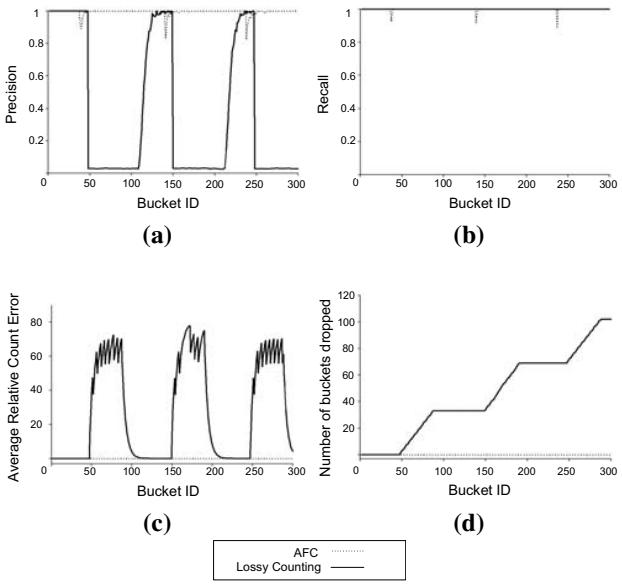

Fig. 7. The precision/recall, number of buckets dropped, and average relative count error after mining each bucket of data.

In the experiment, we set the decay factor $\alpha = 0.8$ and there are 10 buffer slots in the system ($Q = 10$), for AFC, the error threshold was initially set at $\epsilon_0 = 0.3\%$ and the target buffer occupancy fQ was set at 2. From Figure 7(a), we observe that LC suffered from a significant loss in precision at three different periods. During these periods, LC also suffered from an extremely high average relative count error as shown in Figure 7(c). The average relative count error reached 80 at some points. That is, at certain points in the experiment, the estimated count of an average itemset was roughly 80 times of its actual count.

As discussed in Section V-A, there are three sharp bursts in the synthetic data streams. In this sample data stream, the sharp bursts started at bucket numbers 36, 138 and 237 and ended at bucket numbers 87, 189 and 287, respectively. From the figures, we observe that LC's accuracy degraded significantly during and after the data bursts. On the other hand, AFC maintained a 100% precision, a 100% recall and a low relative count error for most of the mining period. Therefore, AFC is able to handle data bursts much more gracefully than LC.

Figure 7(d) shows that LC dropped some buckets of transactions during the data bursts. This is why LC had such a low precision and high average relative count error. LC, which mines the data with a preset error threshold and thus a preset processing rate, cannot cope with the high data arrival rate during data bursts. As a result, buffer overflows and some buckets of transactions are dropped. Losing the transactions of a bucket T_i implies that the maximum error bound $\delta(X)$ of each entry X in D_i has to be incremented by $|T_i|$. This results in high relative count errors.

We note that the loss of accuracy in LC was mainly caused

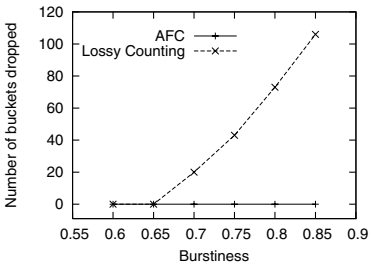

Fig. 8. Buckets dropped vs. burstiness

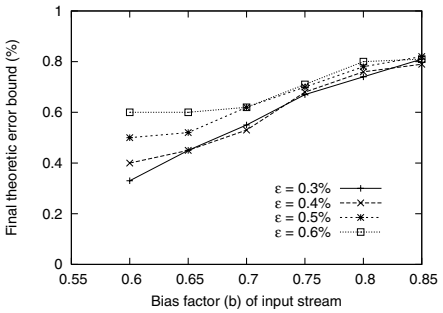

Fig. 9. The final theoretic error thresholds against data burstiness over different initial error thresholds.

by the buckets dropped during data bursts. Thus, we evaluated the number of buckets dropped by LC and AFC over different data streams. Figure 8 shows the number of buckets dropped by LC and AFC in streams with different burstiness (i.e., the bias factor b). In this experiment, we set $\alpha = 0.8$, $\epsilon_0 = 0.3\%$, and $Q = 10$ buckets. We observe that LC could handle all data buckets for data streams with mild burstiness ($b \leq 0.65$). However, as burstiness increased, LC could not cope with the increased data rate during data bursts and more buckets were dropped. On the other hand, AFC avoided bucket drops even for very bursty data streams ($b = 0.85$).

C. Adaptability

Recall that LC requires a fixed preset error threshold (ϵ). In some cases, it is hard to predict the data arrival rate and thus it is difficult to select an appropriate error threshold for LC. A small ϵ leads to slow processing speed and LC cannot cope with data bursts while a large ϵ implies a loose accuracy guarantee. On the other hand, since AFC is capable of dynamically adjusting ϵ during the mining exercise, the initial value of the error threshold ϵ_0 does not affect the system performance much. In the next experiment, we executed AFC a number of times, each with a different value of ϵ_0 and b. At the end of each run (i.e., after 300 buckets were processed), we recorded the final error threshold (ϵ_{300}). Figure 9 shows the result. From the figure, we observe that when the burstiness was high (e.g., when $b \geq 0.75$), the final error thresholds were very similar even under very different initial ϵ_0 settings. This shows that the initial choice of the error threshold parameter is not critical for AFC to function properly. It has the ability to adjust itself in search of an appropriate threshold value.

Also, when the burstiness was mild (e.g., when $b = 0.6$), the final threshold value stabilized at different levels. This is because, in those cases, the transaction arrival rate was manageable even during data bursts. The target buffer occupancy was rarely exceeded and the the feedback mechanism rarely increased the error threshold value. As a result, the final error threshold value stayed close to the initial setting ϵ_0.

VI. Conclusions

In this paper we reviewed the Lossy Counting algorithm and identified its weakness in handling bursty data streams. We proposed the AFC algorithm and described its architecture. We analyzed the properties of LC and proposed a method of controlling its processing speed through the manipulation of the error threshold parameter. We studied the relationship between processing speed and the error threshold and designed a feedback mechanism that can dynamically adjust the bucket mining time to cope with bursty data traffics. We evaluated the resulting AFC algorithm and compared its performance against LC's in terms of the accuracy of the results reported by the two algorithms. Through an extensive set of experiments, we showed that AFC was able to handle bursty data streams without sacrificing accuracy by much.

References

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of 20th Intl. Conf. on Very Large Data Bases*, pages 487–499, 1994.

[2] B. Babcock and C. Olston, "Distributed top-k monitoring," in *Proc. of ACM SIGMOD*, pages 28–39, 2003.

[3] J. H. Chang and W. S. Lee, "Finding recent frequent itemsets adaptively over online data streams," in *Proc. of KDD*, pages 487–492, 2003.

[4] J. H. Chang and W. S. Lee, "estWin: Adaptively monitoring the recent change of frequent itemsets over online data streams," in *Proc. of CIKM*, pages 536–539, 2003.

[5] J. H. Chang and W. S. Lee, "A sliding window method for finding recently frequent itemsets over online data streams," in *Journal of Information Science and Engineering*, 20(4), pages 753–762, 2004.

[6] J. Cheng, Y. Ke, and W. Ng, "Maintaining frequent itemsets over high-speed data streams," in *Proc. of PAKDD*, pages 462–467, 2006.

[7] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," in *Proc. of NSF Workshop on Next Generation Data Mining*, 2002.

[8] H. Li, S. Lee, and M. Shan, "An efficient algorithm for mining frequent itemsets over the entire history of data streams," in *Proc. 1st Intl. Workshop on Knowledge Discovery in Data Streams*, 2004.

[9] B. Lin, W.-S. Ho, Ben Kao, and C.-K. Chui, "Adaptive frequency counting over bursty data streams," Technical Report TR-2007-03, The University of Hong Kong, 2007.

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

[11] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *Proc. of IEEE 21st Intl. Conf. on Data Engineering*, pages 767–778, 2005.

[12] M. Wang, T. Madhyastha, N. H. Chan, S. Papadimitriou, and C. Faloutsos, "Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic," in *Proc. of IEEE 18th Intl. Conf. on Data Engineering*, pages 507–516, 2002.

[13] J. X. Yu, Z. Chong, H. Lu, and A. Zhou, "False positive or false negative: Mining frequent itemsets from high speed transactional data streams," in *Proc. of 30th Intl. Conf. on Very Large Data Bases*, pages 204–215, 2004.