False Positive or False Negative: Mining Frequent Itemsets from High Speed Transactional Data Streams

Jeffrey Xu Yu[1], Zhihong Chong[2], Hongjun Lu[3], Aoying Zhou[2]

[1] The Chinese University of Hong Kong, Hong Kong, China, yu@se.cuhk.edu.hk
[2] Fudan University, Shanghai, China, {zhchong,ayzhou}@fudan.edu.cn
[3] The Hong Kong University of Science and Technology, Hong Kong, China, luhj@cs.ust.hk

Abstract

The problem of finding frequent items has been recently studied over high speed data streams. However, mining frequent itemsets from transactional data streams has not been well addressed yet in terms of its bounds of memory consumption. The main difficulty is due to the nature of the exponential explosion of itemsets. Given a domain of I unique items, the possible number of itemsets can be up to $2^I - 1$. When the length of data streams approaches to a very large number N, the possibility of an itemset to be frequent becomes larger and difficult to track with limited memory. However, the real killer of effective frequent itemset mining is that most of existing algorithms are false-positive oriented. That is, they control memory consumption in the counting processes by an error parameter ϵ, and allow items with support below the specified minimum support s but above $s - \epsilon$ counted as frequent ones. Such false-positive items increase the number of false-positive frequent itemsets exponentially, which may make the problem computationally intractable with bounded memory consumption. In this paper, we developed algorithms that can effectively mine frequent item(set)s from high speed transactional data streams with a bound of memory consumption. While our algorithms are false-negative oriented, that is, certain frequent itemsets may not appear in the results, the number of false-negative itemsets can be controlled by a predefined parameter so that desired recall rate of frequent itemsets can be guaranteed. We developed algorithms based on Chernoff bound. Our extensive experimental studies show that the proposed algorithms have high accuracy, require less memory, and consume less CPU time. They significantly outperform the existing false-positive algorithms.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

Recently, data streams emerged as a new data type that attracted great attention from both researchers and practitioners. A data stream is essentially a virtually unbounded sequence of data items arriving at a rapid rate. Since data items arrive continuously, it is only feasible to store certain form of synopsis (in memory or disk) rather than the raw data for analysis or information extraction. It is also infeasible to multiple scan the original data to build such synopsis because of the massive volume as well as the rapid arrival rate. Research work related to data streams boils down to the problem of finding the right form of synopsis and related construction algorithms so that the required statistics or patterns can be obtained with a bounded error for unbounded input data items with limited memory. A large amount of work has been reported for various statistics and patterns, including simple aggregates and statistics such as maximum, minimum, average, median values and quantiles as well as complex patterns such as decision trees, clusters, and frequent itemsets.

In this paper we study the problem of mining frequent item(set)s (or pattern) from high speed *transactional data streams*. Manku and Motwani gave an excellent review of wide range applications for the problem of frequent data stream pattern mining [12]. The problem can be stated as follows. Let $I = \{x_1, x_2, \cdots, x_n\}$ be a set of items. An itemset is a subset of items I. A transactional data stream, $\mathcal{D}$, is a sequence of incoming transactions, $(t_1, t_2, \cdots, t_N)$, where a transaction t_i is an itemset and N is a unknown large number of transactions that will arrive. The number of transactions in $\mathcal{D}$ that contain X is called the support of X, denoted as $sup(X)$. An itemset X is a frequent pattern, if and only if $sup(X) \geq sN$, where $s = sup(X)/N$ is a threshold called a minimum support such that $s \in (0, 1)$. The *frequent data stream pattern mining*, denoted FDPM, is to find an *approximate* set of frequent patterns (itemsets) in $\mathcal{D}$ with respect to a given support threshold, s. The approximation is controlled by two parameters, ϵ and δ, where ϵ ($\in (0, 1)$) controls errors and δ ($\in (0, 1)$) controls reliability. We call it (ϵ, δ) approximation scheme.

The challenge is to devise algorithms to support (ϵ, δ) approximation for the FDPM problem with a bound regarding space complexity. The main difficulty is the nature of the exponential explosion of frequent patterns mining. For data streams, the incoming transactions will not be stored, and we can only scan them once. If counts are required for each itemsets, an application with m distinct items will require $2^m - 1$ counts. Even with a moderate set of items, for example $m = 1,000$. The total number of itemsets is $2^m - 1 = 2^{1000} - 1$, which is obviously intractable.

A simple version of FDPM problem, that is, mining frequent items but not itemsets, has been recently widely studied in data stream environments with bounded memory [2, 3, 5, 6, 7, 9, 10, 12]. After a careful study of these published work, we observed that, while the detailed algorithms are different, almost all of them are *false-positive* oriented approaches. That is, given a minimum support s, they control memory consumption in the counting processes by an error parameter ϵ, and allow items with support below the specified minimum support s but above $s-\epsilon$ counted as frequent ones.

In this paper, we argue that since frequent item mining is the first step in frequent itemsets mining, even a small number of false-positive items, resulted from the false-positive oriented item counting, could lead to a large number of false-positive itemsets, which makes efficient and effective frequent itemsets mining infeasible. This motivated us to develop a false-negative oriented approach for frequent items mining. In addition, to further address the problem caused by explosion of frequent itemsets, we explored a tight bound to control the counting process for frequent item(set)s mining. As a summary, our contribution can be summarized as follows.

- While most existing work follows the approach of false-positive oriented frequent item counting, we show that false-negative oriented approach that allows a controlled number of frequent itemsets missing from the output is a more promising solution for mining frequent itemsets from high speed transactional data streams.
- We developed the first set of one-scan false-negative oriented algorithms which significantly outperform the existing false-positive oriented approaches for frequent itemsets mining as well as frequent items mining. We also derived memory bounds for both cases.
- Most existing approaches use the error parameter for two purposes which are conflict: quality control (ϵ) and memory size control ($1/\epsilon$ or $1/\epsilon^2$), which leads to a dilemma: a little increase of ϵ will make the number of false-positive items large, and a little decrease of ϵ will make memory consumption large. Our algorithms adopt a (ϵ, δ) approximation scheme with $\epsilon = 0, \delta > 0$ that decouples the two interrelated but conflict purposes and makes the parameter setting of the mining process easier.

The remainder of the paper is organized as follows. Section 2 analyzes the false-positive and false-negative approaches in frequent item(set)s mining. Section 3 and 4 present our frequent items mining algorithm, and the results of performance study. Section 5 and 6 present our frequent itemsets mining algorithm, and the results of performance study. Section 7 concludes the paper.

2 False-Positive versus False-Negative

Due to the little space allowed to mine frequent data stream patterns, the key point becomes how to prune those potentially infrequent patterns and how to maintain potentially frequent patterns with probabilistic guarantees [11]. Approximate mining frequent patterns with probabilistic guarantee can take two possible approaches, namely, false-positive oriented and false-negative oriented. The former includes some infrequent patterns in the final result, whereas the latter misses some frequent patterns.

There are a large number of publications on the false-positive oriented approaches [2, 3, 5, 6, 7, 9, 10, 12], and there is no reported study on one-scan false-negative oriented approach. All false-positive oriented approaches focused on frequent items mining, rather than frequent itemsets mining. In [12], as the first attempt, Manku and Motwani also studied false-positive oriented frequent itemsets mining in a less theoretical nature and with a focus on system-level issues.

2.1 Deficiency of False-Positive Oriented Approaches

Because the focus of this paper is on frequent itemsets mining, we concentrate ourselves on frequent itemsets mining, and we mainly address it in comparison with the algorithms proposed by Manku and Motwani [12].

In [12], Manku and Motwani developed two false-positive oriented algorithms for frequent items counting, Sticky-Sampling and Lossy-Counting. The Sticky-Sampling uses $O(\frac{1}{\epsilon}\log(s^{-1}\delta^{-1}))$ expected number of entries, and Lossy-Counting uses $O(\frac{1}{\epsilon}\log(\epsilon N))$ entries. In theory, Sticking-Sampling requires constant space, while Lossy-Counting requires space that grows logarithmically with N. In practice, as shown in [12], Sticky-Sampling performs worse because of its tendency to remember unique items sampled. Lossy-Counting can prune low frequency items quickly and keep only high frequent items. Based on this fact, Manku and Motwani give a Lossy-Counting based three module system (Buffer-Trie-SetGen) for mining frequent itemsets in a less theoretical nature. The main features of their algorithms include (1) All item(set)s whose true frequency exceeds sN are output, (2) no item(set)s whose true frequency is less than $(s-\epsilon)N$ is output, and (3) estimated frequencies are less than the true frequencies by at most ϵ.

In the following, without loss of generality, we address our false-negative approach in comparison with the Lossy-Counting algorithm and the Buffer-Trie-SetGen approach.

Remark 1 *Like Sticky-Sampling, Lossy-Counting is false-positive oriented and is ϵ-deficient. The parameter ϵ is coupled with two conflict goals. First, let f_ϵ be the number of items in $[s-\epsilon, s]$. Then $f_\epsilon \geq f_{\epsilon'}$ if $\epsilon > \epsilon'$. The smaller ϵ is, a less number of false-positive items are included in the result set. Second, because the memory consumption is a factor of $1/\epsilon$, the memory consumption increases reciprocally in terms of ϵ.*

The Remark 1 states the dilemma of false-positive oriented approaches (ϵ-deficient). The memory consumption increases reciprocally in terms of ϵ where ϵ controls the error bound. It is difficult to decouple the two functions, memory consumption control and error control, from the error bound ϵ. In Sticky-Sampling, ϵ is used to determine a sampling rate, and in Lossy-Counting, ϵ is used to determine the bucket width. Changing their ways of dealing with ϵ means to change the worst case space-complexity analysis.

The impacts of parameter ϵ will be even great when frequent itemsets mining is concerned, which is in fact related to the fundamental issue on application of Apriori property [1]. The Apriori property states: if any length k pattern is not frequent in a dataset, its length (k+1)-th super-patterns can never be frequent. In other words, the Apriori property suggests to use possibly smallest k-th frequent itemsets to generate the (k+1)-th candidate itemsets, and then mine the (k+1)-th candidate itemsets. The false-positive oriented approaches allow 1-itemsets with support below s but above $s-\epsilon$ counted as frequent. Consequently, when there are some false-positive 1-itemsets in $[s-\epsilon, s]$, the nature of the exponential explosion makes the number of potential frequent itemsets be very large and makes false-positive oriented approaches difficult to manage it.

2.2 Our False-Negative Oriented Approach: ϵ-Decoupling

False-positive oriented approaches have their limit to support frequent item(set)s mining. One of the main difficulties is caused by the conflicts of the error parameter ϵ as stated in Remark 1. In this paper, we decouple the two conflict functions of the error parameter ϵ as follows.

- **Error Control and Pruning**: We use an effective ϵ to control error bound, which is *changeable* and is not fixed. The effective value of ϵ becomes smaller while more data items are received from a data stream. In brief, we compute the effective value of ϵ using minimum support s (user given), reliability δ (user given), and the number of observations n (variable), where ϵ is reciprocal to n. The effective value of ϵ approaches to zero when the number of observations increases. Therefore, the frequent item(set)s mining becomes more accurate. It is important to note that we use ϵ to prune data but do not use it to control memory.

- **Memory Control**: We use the reliability δ instead of ϵ to control memory consumption. Different from false-positive oriented approaches whose memory consumption is determined by $1/\epsilon$, the memory consumption in our algorithms is related to $\ln(1/\delta)$. Consider the same memory space using either ϵ or δ, we have $1/\epsilon = \ln(1/\delta)$. For getting the same memory space, when $\epsilon = 0.1$, $\delta = 0.00005$; when $\epsilon = 0.01$, $\delta = 3.7 \times 10^{-44}$. Because in practice, $\delta = 0.0001$, our approach can significantly reduce the memory consumption and processing cost for frequent item(set)s mining, while achieving high accuracy. We will discuss bounds for frequent item(set)s mining later.

Our approach does not allow 1-itemsets with support below s counted as frequent, and therefore is a false-negative oriented approach. We will give the details of our approach, and show that the possibility of missing frequent item(set)s is considerably small later in this paper.

Our one-scan false-negative oriented approach is different from Toivonen's two-scan false-negative oriented approach [13]. In brief, Toivonen's algorithm is to pick a random sample and find all association rules using this sample that probably hold in the whole dataset in one pass, and to verify the results with the rest of the dataset. It allows false-negative with probabilistic guarantees, and the sample size can be at least $O((\frac{c}{\epsilon^2}\log(\delta^{-1}))$. One of the problem of the Toivonen's algorithm is that, because the error parameter ϵ can be very small, the memory consumption using Toivonen's algorithm can be very large ($1/\epsilon^2$). We summarized some bounds in Table 1 for comparison. Note, in Table 1, as a false-positive approach, GroupTest does not rely on ϵ. But it requests the knowledge of the domain of a data stream, which is difficult to obtain beforehand.

2.3 Frequent Itemsets Mining: A Comparison

To verify our analysis, we conducted experiments to study the impacts of a large number of itemsets in the range of $[s-\epsilon, s+\epsilon]$ on frequent itemsets mining. We report here one of the experiments. We generated a data stream of length 1,000K which has an average transaction size 15 and maximal potentially frequent itemset size 6, with 10K unique items. We implemented the Lossy-Counting based frequent itemset mining approach, denoted as BTS (Buffer-Trie-SetGen) [12]. With $\epsilon = \frac{s}{10}$ and $\delta = 0.1$, we obtained results as shown in Table 2. To measure the quality, we use two metrics, recall and precision, that are defined as follows. Given a set of true frequent itemsets A and a set of obtained frequent itemsets B, the recall is $\frac{|A \cap B|}{|A|}$ and the precision is $\frac{|A \cap B|}{|B|}$.

s (%)	True Size	Mined Size	Recall	Precision
0.08	21,361	126,307	1.00	0.17
0.10	12,252	68,275	1.00	0.18
0.20	2,359	23,154	1.00	0.16

Table 2: Impact of false positives in BTS

In Table 2, the first column is the minimum support (s), and the second is the true size of frequent

Algorithm	Type	Space
Charikar et al [3]	False-Positive	$O(\frac{k}{\epsilon^2}\log(n/\delta))$
Sticky-Sampling [12]	False-Positive	$O(\frac{1}{\epsilon}\log(s^{-1}\delta^{-1}))$
Lossy-Counting [12]	False-Positive	$O(\frac{1}{\epsilon}\log(\epsilon N))$
GroupTest [6]	False-Positive	$O(k(\log(k)+\log(\delta^{-1}))\log(M))$
Toivonen [13]	False-Negative	$O((\frac{c}{\epsilon^2}\log(\delta^{-1}))$
FDPM-1 (this paper)	False-Negative	$O((2+2\ln(2/\delta))/s)$

Table 1: Theoretical Memory Bounds

itemsets($|A|$). The next three columns are a summary for the quality of BTS using minimum support s. The first of the three columns is the result size ($|B|$). The second and third columns of the three columns are its recall and precision. It can be seen that the sizes of obtained results are about 10 times larger than the true size. All the three recalls are 1, which means that the obtained results contain all the true frequent itemsets. All the three precisions are less than 0.2, which means that the obtained results contain a large number of itemsets below s but above $s-\epsilon$. The number of false positive is large, and its impact is significant in two ways: i) the quality of mining result is low, and ii) the memory needed at run time is even larger accordingly.

Astute readers may suggest to turn a false-positive algorithm into a false-negative one for frequent itemset mining. That is, for user given s and ϵ, we can deliberately use $s+\epsilon$ as the minimum support to mine the frequent itemset so that the output will contain only those frequent itemsets with support greater than s but some of frequent itemsets between s and $s+\epsilon$ may not be in the output, which makes the algorithm false-negative. We implemented such idea and obtained results as shown in Table 3. Note, the true frequent itemsets in Table 2 and Table 3 are the same. We can see that, in Table 3, the precisions become 1.0 as there are no false-positive. However, the recall rate drops 15-26% that seems unsatisfactory low.

s (%)	True Size	Mined Size	Recall	Precision
0.08	21,361	18,351	0.86	1.00
0.10	12,252	10,411	0.85	1.00
0.20	2,359	1,739	0.74	1.00

Table 3: Impact of false negatives: BTS($s+\epsilon$) where $\epsilon = s/10$.

We tested our false-negative oriented approach. For the same minimum support (s) in Table 2, with $\epsilon = s/10$ and $\delta = 0.1$, we achieve 0.99 recall and 1.0 precision in all the setting. Recall: BTS does not perform well, because there are many itemsets in $[s-\epsilon, s+\epsilon]$ (Table 2). In order to test whether our false-negative oriented approach misses itemsets, we used the same setting as Table 2 but set minimum support to be $s-\epsilon$ instead, and tested if we miss many itemsets in $[s-\epsilon, s+\epsilon]$. We found that we can still achieve 0.99 recall and 1.0 precision in all the setting.

As a conclusion, we believe that contrary to most existing approaches, the false-negative oriented approach is more promising to solve the FDPM problem.

3 Mining Frequent Items from a Data Stream

In this section, we focus on frequent items mining, and discuss Chernoff bound [4], our basic approach and algorithm. We will discuss frequent itemsets mining in Section 5.

3.1 Chernoff Bound

Suppose there is a sequence of observations, $o_1, o_2, \cdots, o_n, o_{n+1}, \cdots$. Chernoff bound gives us certain probabilistic guarantees on the estimation of statistics about the underlying data, that generates these observations, based on the n observations obtained so far. Consider the sequence of observations, $o_1, o_2, \cdots, o_n$, as n independent Bernoulli trails (coin flips) such that $Pr[o_i = 1] = p, Pr[o_i = 0] = 1-p$ for a probability p. Let r be the number of heads in the n coin flips. The expectation of r is np. Chernoff bound states, for any $\gamma > 0$,

$$Pr\{|r - np| \geq np\gamma\} \leq 2e^{\frac{-np\gamma^2}{2}}$$

Let $\overline{r}$ be r/n, and consider the minimum support s as the probability p. The above equation becomes

$$Pr\{|\overline{r} - s| \geq s\gamma\} \leq 2e^{\frac{-n\,s\gamma^2}{2}}$$

Further, we replace $s\gamma$ with ϵ.

$$Pr\{|\overline{r} - s| \geq \epsilon\} \leq 2e^{\frac{-n\,\epsilon^2}{2s}} \quad (1)$$

Let the right side of Equation (1) be δ. We see that, with probability $\leq \delta$, the running average $\overline{r}$ is beyond $\pm\epsilon$ of s, where

$$\epsilon = \sqrt{\frac{2s\ln(2/\delta)}{n}} \quad (2)$$

FDPM can be considered as an application of Chernoff bound as follows. Given a sequence of 1-item transactions, $\mathcal{D} = t_1, t_2, \cdots, t_n, t_{n+1}, \cdots, t_N$, where n is the number of first n transactions being observed such as $n \ll N$. For a pattern X, its running support up to n is $\overline{sup}(X)$ and its true support up to N is $\sup(X)$. By replacing $\overline{r}$ with $\overline{sup}(X)/n$ and r with $s\ (= sup(X)/N)$, respectively, we can make the following statement. For a pattern X, when n observations have been made, the running support of X is beyond $\pm\epsilon$ of s with probability $\leq \delta$. In other words, the running support of X is within $\pm\epsilon$ of s with probability

$\geq 1 - \delta$.

Consider $s = 0.1$, $\delta = 0.1$ and $\epsilon = 0.01$. With Chernoff bound, $n \approx 5,991$ (Equation (2)). This implies the following for a pattern X. If we have about $5,991$ observations, its true value $sup(X)/N$ is in the range of $(\overline{sup}(X)/n - 0.01, \overline{sup}(X)/n + 0.01)$ with high probability 0.9.

3.2 The Basic Approach

Based on the Chernoff bound, we group arrival items into two groups, namely potentially infrequent patterns and potentially frequent patterns. They are defined as follows.

Definition 1 *Given n observations, a running error ϵ_n in term of n can be obtained (Eq. (2)). A pattern X is potential infrequent if $\overline{sup}(X)/n < s - \epsilon_n$ in terms of n. A pattern X is potential frequent if it is not potential infrequent in terms of n.*

The conditions for determining potential infrequent pattern can be represented alternatively as $\overline{sup}(X) < (s - \epsilon_n)n$ for a given n observations. A pattern X is potential frequent if $\overline{sup}(X) \geq (s - \epsilon_n)n$.

It is important to note that ϵ_n is not the user-specified parameter ϵ but a running variable. The running error ϵ_n decreases, while the number of observations n increases. When n becomes a very large number N, $\epsilon_n \approx 0$. Therefore, $\overline{sup}(X) \approx sN$.

Remark 2 *Our algorithm is false-negative oriented and is a $(0, \delta)$ approximate scheme.*

Remark 3 *For a given minimum support s and reliability δ. The memory consumption is bounded in terms of the number of observations, and is much less than the number of observations in practice.*

The Remark 3 states the fact that the same transactions may appear many times in a transactional data stream. As discussed later, our bound does not rely on the user-specified error ϵ, but on a running error ϵ_n which decreases while the number of observations n increases.

3.3 Mining Frequent Items

Our algorithm for mining frequent items from a data stream, denoted `FDPM-1`, is outlined in Algorithm 1, which takes s and δ as inputs. Note that we do not take ϵ as input. Algorithm 1 makes use of the Chernoff bound. In line 1, n_0 is the required number of observations, which is given below.

$$n_0 = \frac{2 + 2\ln(2/\delta)}{s} \tag{3}$$

We will show how we determine n_0 later, which in fact is the memory bound. Now, when we receive a transaction t from a 1-itemset transactional data stream, we check whether it exists in the pool of P. If it exits, we increase its count by 1 (line 4-5). Otherwise, we insert t into P if the number of entries in P is less than n_0. When P becomes full ($P \geq n_0$) (line 7), we prune potential infrequent patterns X in P based on Definition 1.

Algorithm 1 FDPM-1(s, δ)

```
1:  let n_0 be the required number of observations (Eq. (3));
2:  n ← 0, P ← ∅;
3:  while a new transaction t arrives do
4:    if t ∈ P then
5:      increase t's count by 1;
6:    else
7:      if |P| > n_0 then
8:        calculate the running ε_n for the n observations;
9:        delete all entries in P that are potentially infrequent;
10:     end if
11:     insert t with an initial count 1 into P;
12:   end if
13:   n ← n + 1;
14:   output P on demand;
15: end while
```

We output the mining results (P) only when there is such a demand at line 14. Note: we do not initially allocate memory for keeping n_0 entries in P. We increase the size of P incrementally.

Theorem 1 *Algorithm 1 finds frequent 1-itemsets in a data stream, with two parameters s and δ. Algorithm 1 ensures the followings, when data is independent.*

(a) All items whose true frequency exceeds sN are output with probability of at least $1 - \delta$.

(b) No items whose true frequency is less than sN are output.

(c) The probability of the estimated support that equals the true support is no less than $1 - \delta$.

(d) The bound on memory space is $(2 + 2\ln(2/\delta))/s$ when the Chernoff bound is used.

The proof of Theorem 1 is sketched below.

First, the first three properties (a), (b) and (c) can be directly derived from the Chernoff bound. When n transactions have been received, the true support of a pattern X, $sup(X)/N$, for $N \gg n$, is within $\pm\epsilon_n$ of the running support $\overline{sup}(X)/n$ when the Chernoff bound is used. Recall ϵ_n approaches 0 when the number of observations n increases. Because we prune potential infrequent patterns whose true support is not in the given range with probability δ, the probability of pruning a frequent pattern is at most δ. Therefore, the probability of the estimated support that equals the true support is no less than $1 - \delta$.

Second, we show the proof for the property (d) when the Chernoff bound is concerned. As shown in Algorithm 1, P always keeps all potential frequent patterns

X such that $\overline{sup}(X) \geq (s - \epsilon_n)n$, when n transactions have been received. Therefore, $|P| \leq 1/(s - \epsilon_n)$, when $s - \epsilon_n > 0$, otherwise $|P| \cdot (s - \epsilon_n)n > n$, which is impossible. let $|P| = n = 1/(s - \epsilon_n)$. We have the following equation.

$$n = \frac{1}{s - \epsilon_n} = \frac{1}{s - \sqrt{\frac{2s \ln(2/\delta)}{n}}} \quad (4)$$

Solve the equation, we get

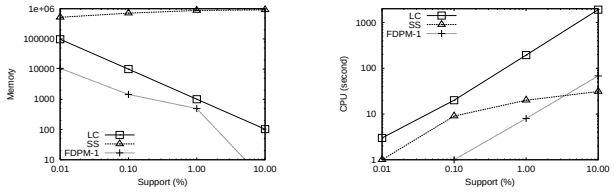
$$n = \frac{2 + 2\ln(2/\delta)}{s} \quad (5)$$

as the proof of the last property (d) of Theorem 1.

The last property of Theorem 1 is proved for the minimum number of observations. As an example, suppose $s = 0.001$, $\epsilon = s/10$ and $\delta = 0.1$. The memory bound is $n_0 = 7,991$. Consider ϵ_n as follows. When $n \leq n_0$, $\epsilon_n = 0$, because all items can be kept in the pool. When $n = 7,992$ $(= n_0 + 1)$, $\epsilon_n = 0.000866$ (the largest possible error). When $n = 100,000$, $\epsilon_n = 0.000245$. When $n = 1,000,000$, $\epsilon_n = 0.000077$.

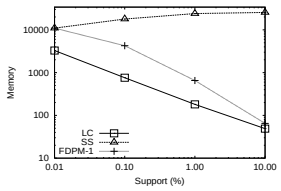
We discuss the time complexity regarding Algorithm 1 in brief. In Algorithm 1, the cost for inserting a new item is $O(1)$. The cost for one pruning is $O(1)$ because we only maintain n_0 items. The maximum number of pruning is at most N/n_0 where N is the length of the data stream.

Algorithm 1 is designed on top of the Chernoff bound which assumes data independent. In reality, data in a data stream is highly possible to be dependent. When data is dependent in a transactional data stream, the quality of Algorithm 1 cannot be guaranteed. Several approaches can be taken to handle data dependent data streams. One is to conduct random sampling with a reservoir [14], as indicated in [12]. The technique of random sampling with a reservoir is, in one sequential pass, to select a random sample of n records without replacement from a pool of N records where N is unknown [14]. With this technique, we can handle a data dependent data stream as a data independent data stream. In [8], a probabilistic-inplace algorithm was introduced to handle different distributions. Given m counters, the probabilistic-inplace algorithm reserves $m/2$ to store the current best candidates, and uses the unreserved $m/2$ to monitor network traffic. For every run, the probabilistic-inplace algorithm replaces the $m/2$ reserved counters with the top out of all m counters. With this technique, we can divide a transactional data stream into segments and apply Algorithm 1 to segments one-by-one continuously. The length of each segment is $k \cdot n_0$ where k is a positive number and n_0 is the smallest number of observations. The memory required is $2n_0$ – one for reserving potentially frequent patterns and the other for monitoring a segment.

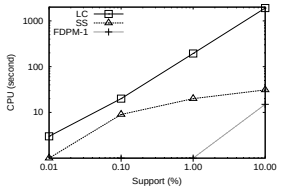


(a) Memory (Zipf = 0.5)

(b) CPU (Zipf = 0.5)



(c) Memory (Zipf = 1.5)



(d) CPU (Zipf = 1.5)

Figure 1: The effectiveness of s ($\epsilon = s/10$, $\delta = 0.1$)

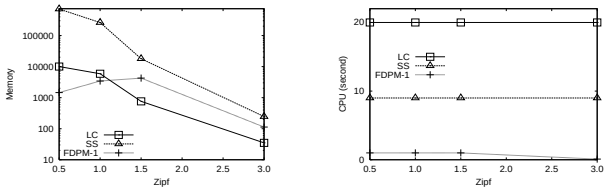
4 Performance Study I: Mining Frequent Items

We report our experimental results for frequent items mining in this section. For frequent items mining, we implemented our false-negative oriented algorithm `FDPM-1` (Algorithm 1). We also implemented false-positive oriented algorithms, Lossy-Counting [12] and Sticky Sampling [12], and denote them as `LC` and `SS`, respectively. For testing frequent items mining, we generate 1-itemset transactional data streams using Zipf distribution.

We implemented all the frequent item(set)s mining algorithms using Microsoft Visual `C++` Version 6.0. We used the same data structures and subroutines in all implementations, in order to minimize any performance differences caused by minor differences in implementation. We conducted all testings (Section 4 and Section 6) on a 1.7GHz CPU Dell PC with 1GB memory. Because the memory size is 1GB, there were no I/Os in all our testings. We report our results in terms of memory consumption (the number of counters) and CPU time (seconds), as well as the recall and precision.

4.1 Data Distribution

We first test two data sets of length 1000K using two Zipf factors, 0.5 and 1.5. We compare the three algorithms: `CL`, `SS`, and `FDPM-1`, by varying s, where $\epsilon = s/10$ and $\delta = 0.1$. The memory and CPU are shown in Figure 1. In both cases, `SS` consumes the largest memory for different s. Different from `SS`, the memory consumption of both `LC` and `FDPM-1` decreases while s increases. It is interesting to note that when Zipf = 0.5, `FDPM-1` outperforms `LC` in terms of memory consumption. On the other hand, when Zipf = 1.5, `LC` outperforms `FDPM-1` in terms of memory consumption. In terms of CPU time, `FDPM-1` outperforms `LC` in all the cases.



(a) Memory (b) CPU

Figure 2: Effectiveness of Zipf factors

Some explanations can be made below. When Zipf = 0.5, the data distribution is near uniform. Because of uniform distribution, there are only a small number of items whose support is greater than the minimum support s. LC and SS need more memory to maintain items in a rather sparse data stream. FDPM-1 prunes items using the running error ϵ_n. While n increases, ϵ_n approaches zero, and it allows us to track those near s items with less memory consumption. When Zipf = 1.5, data is more skewed, and the number of unique items is less. FDPM-1 cannot prune as it does when Zipf = 0.5. LC can effectively prune items whose support is less than $s - \epsilon$. The recall and precision for Zipf = 1.5 are given in Table 4. FDPM-1 achieves 100% recall and 100% precision. SS and LC ensure recall to be 100%, but allow precision to be down to (91%, 92%), despite the fact that the patterns are skewed.

s (%)	LC		SS		FDPM-1	
	R	P	R	P	R	P
0.01	1	0.91	1	0.91	1	1
0.1	1	0.96	1	0.96	1	1
1	1	0.92	1	0.92	1	1
10	1	1	1	1	1	1

Table 4: Varying s (Zipf = 1.5)

In order to investigate the effectiveness of Zipf distribution, we fix $s = 0.1\%$, $\epsilon = s/10$, and $\delta = 0.1$, and test different Zipf factors. The results are shown in Figure 2 in which there is turn over when Zipf is about 1.25 (Figure 2 (a)). When Zipf is less than 1.25, FDPM-1 consumes less memory than LC. FDPM-1 performs best in terms of CPU cost. The recall and precision are shown in Table 5. When Zipf = 0.5, $s = 0.1\%$, no frequent items can be found. When Zipf = 1.0, the precisions of LC and SS are even down to 0.87. FDPM-1 ensures high recall and precision.

Zipf	LC		SS		FDPM-1	
	R	P	R	P	R	P
0.5	-	-	-	-	-	-
1.0	1	0.87	1	0.87	0.99	1
1.5	1	0.96	1	0.96	1	1
3.0	1	1	1	1	1	1

Table 5: Varying Zipf factors

s (%)	LC		SS		FDPM-1	
	R	P	R	P	R	P
0.010	1	0.79	1	0.79	1	1
0.009	1	0.73	1	0.73	1	1
0.008	1	0.79	1	0.79	1	1
0.007	1	0.80	1	0.80	1	1
0.006	1	0.82	1	0.82	1	1
0.005	1	0.80	1	0.80	1	1
0.004	1	0.78	1	0.78	1	1

Table 6: Sliding Window (Zipf = 0.5)

s (%)	LC		SS		FDPM-1	
	R	P	R	P	R	P
0.010	1	0.91	1	0.91	1	1
0.009	1	0.95	1	0.95	1	1
0.008	1	0.92	1	0.92	1	1
0.007	1	0.96	1	0.96	1	1
0.006	1	0.93	1	0.93	1	1
0.005	1	0.93	1	0.93	1	1
0.004	1	0.93	1	0.93	1	1

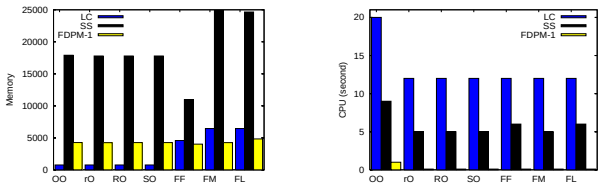
Table 7: Sliding Window (Zipf = 1.5)

4.2 Critical Region Testing

In this section, we further conduct several testing to test a critical region of $[s - \epsilon, s]$. Suppose that many frequent items reside in the critical region. LC and SS may suffer if they use s and ϵ to mine, because it is most likely to include many false-positive and affect the precision. On the other hand, FDPM-1 may suffer if it uses $s - \epsilon$ to mine, because it is most likely to miss items.

First, with a window size $\epsilon = 0.00001$, we slide the minimum support starting from $s = 0.01\%$ to $s - i \cdot \epsilon$ where $i = 1, 2, \cdots, 6$, and test two data streams with Zipf = 0.5 and Zipf = 1.5. The recall and precision are shown in Table 6 and Table 7. Both LC and SS perform in a similar way. When data is not skewed (Zipf = 0.5), LC and SS are easier to include false-positives. FDPM-1 reaches 100% recall and 100% precision in all the cases.

Second, we identify a region with $[s - \epsilon, s]$ using the data stream of Zipf = 1.5 where LC and SS perform well. We artificially move frequent items from $(s, 1]$ to $[s - \epsilon, s]$. where $s = 0.1\%$ and $\epsilon = s/10$. We test LC and SS using s as the minimum support, and test FDPM-1 using $s - \epsilon$ as the minimum support. The recall and precision are shown in Table 8. As expected, the precision of LC and SS decreases while more items reside in $[s - \epsilon, s]$. But, FDPM-1 is insensitive to the number of items in the critical region.



(a) Memory (b) CPU

Figure 3: Effectiveness of data arrival order

$[s-\epsilon, s]$	$(s, 1]$	LC		SS		FDPM-1	
		R	P	R	P	R	P
25	247	1	0.91	1	0.91	1	1
73	200	1	0.74	1	0.74	1	1
123	150	1	0.55	1	0.55	1	1
173	100	1	0.36	1	0.36	1	1
223	50	1	0.17	1	0.17	1	1

Table 8: Critical Region: Test LC/SS with $s = 0.1\%$, and test FDPM-1 with $s = 0.09\%$ where $\epsilon = s/10$, $\delta = 0.1$, Zipf = 1.5

Data	LC		SS		FDPM-1	
	R	P	R	P	R	P
OO	1	0.96	1	0.96	1	1
rO	1	0.96	1	0.96	1	1
RO	1	1	1	1	1	1
SO	1	1	1	1	1	1
FF	1	0.95	1	0.95	1	1
FM	1	0.95	1	0.95	1	1
FL	1	0.95	1	0.95	1	1

Table 9: Data arrival order

4.3 The Impacts of Data Arrival Order

We test data arrival orders, in order to ensure whether our approaches are order sensitive. Let $s = 0.1\%$, $\epsilon = s/10$, $\delta = 0.1$, and Zipf = 1.5. Several data arriving orders are tested: OO (Original Order), rO (reverse Order), RO (Random Order), SO (segment-based random order[1], FF (Frequent First), FM (Frequent Middle), FL (Frequent Last). FDPM-1 is shown to be insensitive to data arrival order. The results are shown in Figure 3 and Table 9. It achieves 100% recall and 100% precision. It outperforms the others in terms of CPU. The memory consumption is not influenced by the data arrival order. LC and SS are rather sensitive to the data arrival order. For example, when frequent items arrive late (FM or FL), both LC and SS consume more than FDPM-1.

Algorithm 2 FDPM(s, δ)

1: let n_0 be the required number of observations (Eq. (3));
2: $n_1 \leftarrow k \cdot n_0$;
3: $n \leftarrow 0$, $\mathcal{F} \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$;
4: **for** every n_1 transactions **do**
5: keep potential frequent patterns in $\mathcal{P}$ in terms of n_1;
6: $\mathcal{F} \leftarrow \mathcal{P} \cup \mathcal{F}$;
7: prune potential infrequent patterns from $\mathcal{F}$ further if $|\mathcal{F}| > c_u \cdot n_0$;
8: $\mathcal{P} \leftarrow \emptyset$;
9: $n \leftarrow n + n_1$;
10: **end for**
11: output the patterns in $\mathcal{F}$ whose count $\geq sn$ on-demand;

[1] We randomly reorder data in a unit of segment (1,000 items)

5 Mining Frequent Itemsets from a Data Stream

We show our frequent data stream pattern (itemsets) mining algorithm in Algorithm 2. In line 1, we obtain n_0 based on the Chernoff bound. Here n_0 is the number of transactions. We divide a transactional data stream into segments. The length of segment is $n_1 = k \cdot n_0$ (line 2). The parameter k controls the size of transactions we process in each run in a similar way like the probabilistic-inplace algorithm in [8]. We maintain potential frequent patterns in $\mathcal{F}$, and use $\mathcal{P}$ for each segment in a run. Both are initialized in line 3. We will discuss the size of $\mathcal{P}$ and $\mathcal{F}$ in detail later. In a for loop statement (line 4-10), we deal with every segment of length of n_1 transactions as an individual data stream repeatedly. For each segment, first, we prune potential infrequent patterns (line 5), using the same techniques given in Algorithm 1. A pattern X is potential infrequent if $\overline{sup}(X) < (s - \epsilon_n)n$ where n increases from 0 to n_1 and ϵ_n is computed in terms of n (Definition 1). Second, we merge the potential frequent patterns in $\mathcal{P}$ with $\mathcal{F}$. That is for every pattern $X \in \mathcal{P}$ with a count c, we increase the count of the same pattern X by c if we can find it in $\mathcal{F}$. Otherwise, we create X in $\mathcal{F}$ with an initial count of c. Third, we further prune potential infrequent patterns in $\mathcal{F}$, when $|\mathcal{F}| > c_u \cdot n_0$ using an existing association rule mining algorithm. We will discuss c_u in detail next.

In Algorithm 2, the k controls the size of segment ($k \cdot n_0$) in a run. If k is small, Algorithm 2 will prune potential infrequent patterns frequently, which leads to less memory but more CPU time. On the other hand, a large k may lead to more memory but less CPU time. Regarding data dependent, we found in our extensive testing that a small k does not necessarily decrease the quality of frequent itemsets mining, because the number of combinations is large, in comparison with frequent items mining.

Theorem 2 *Algorithm 2 finds frequent itemsets in a data stream, with two parameters s and δ. Algorithm 2 ensures the same properties.*

(a) All itemsets whose true frequency exceeds sN are output with probability of at least $1 - \delta$.

(b) No itemsets whose true frequency is less than sN are output.

(c) The probability of the estimated support that equals the true support is no less than $1 - \delta$.

Theorem 2 can be directly derived from the Chernoff bound. Below, we concentrate ourselves on bounds of Algorithm 2.

In Algorithm 2, $\mathcal{P}$ keeps potential frequent itemsets in a segment of n_1 transactions, and $\mathcal{F}$ keeps potential frequent itemsets in all n transactions received so far. At run time, some potential infrequent itemsets may exist in $\mathcal{P}$ ($\mathcal{F}$). An itemset in $\mathcal{P}$ ($\mathcal{F}$) is an entry (a pair of itemset and count). We discuss the size of $\mathcal{P}$ ($\mathcal{F}$) in terms of the number of entries, denoted $|\mathcal{P}|$

($|\mathcal{F}|$). Obviously, $|\mathcal{P}| \leq |\mathcal{F}|$. The size $|\mathcal{P}|$ ($|\mathcal{F}|$) can be possibly larger than n_1 (n). For example, suppose that we receive 2 transactions, $\cdots, t_1, t_2, \cdots$, where $t_1 = \{1,2\}$ and $t_2 = \{2,3\}$. The possible potential frequent itemsets can be $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, and $\{1,2,3\}$. Because $\mathcal{P}$ ($\mathcal{F}$) may contain potential infrequent patterns, the theoretical upper bound of $\mathcal{P}$ ($\mathcal{F}$) is difficult to be determined due to the nature of the exponential explosion of itemsets.

In this paper, we address an empirical upper bound of $|\mathcal{F}|$ ($|\mathcal{P}| \leq |\mathcal{F}|$) using the Chernoff bound. We show that the empirical upper bound of $|\mathcal{F}|$, u_F, can be determined as a factor of n_0, that is,

$$u_F = c_u \cdot n_0$$

such as $|\mathcal{F}| \leq u_F$. Here, n_0 is determined by the Chernoff bound (Eq. (3)). The empirical upper bound (u_F) is determined as follows.

First, let $\mathcal{F}_{max}$ denote the largest $|\mathcal{F}|$ for a given minimum support s in the process of frequent itemsets mining. Here, $\mathcal{F}_{max}$ is the number of entries used for processing transactions up to the current n transactions ($n \geq n_1$). We obtained different $\mathcal{F}_{max}$ values using `T10.I4.D1000K` and `T15.I6.D1000K`, by varying s and δ. In Table 10, due to the space limit, we only show the results with $\delta = 0.1$. We find that $c_{max} = \mathcal{F}_{max}/s^{-3}$ is about the same for different minimum support values (s), for a data stream with a given δ. The c_{max} value obtained from `T15.I6.D1000K` is larger than the c_{max} value obtained from `T10.I4.D1000K`, because the average of transaction size and the maximal potentially frequent itemsets of `T15.I6.D1000K` are larger than those of `T10.I4.D1000K`. Note: when δ decreases (higher reliability), c_{max} increases a little. For example, when $s = 0.1\%$ and $\delta = 0.01$, $c_{max} = 0.0001$ and $c_{max} = 0.00072$ for `T10.I4.D1000K` and `T15.I6.D1000K`, respectively.

Second, based on our finding, consider $F_{max} = b_{max} \cdot n_0$ for a given minimum support s, then, we have

$$b_{max} = \mathcal{F}_{max}/n_0 = (c_{max}/s^3)/n_0.$$

Some b_{max} values are shown in Table 11 for different minimum supports. Several points can be made: i) b_{max} increases while the minimum support s decreases. ii) b_{max} can be greater than, equal to, or less than 1.

Third, for determining the empirical upper bound of $|\mathcal{F}|$ for different data streams, $\mathcal{D}_1, \mathcal{D}_2, \cdots$, the above finding suggests that we can select the largest c_{max}, $\overline{c}_{max}$, to determine the largest b_{max} value using a *representative* data stream $\mathcal{D}_r$. For example, `T15.I6.D1000K` is the representative in comparison with `T10.I4.D1000K`, because `T15.I6.D1000K` has a larger transaction size and a larger maximal potentially frequent itemsets than `T10.I4.D1000K`. As future work, we will further study the issues related to the representative data streams. In Table 10, $\overline{c}_{max} = 0.00045$. Alternatively, we can determine $\overline{c}_{max}$ based on a regression line among c_{max} values. Consequently, we can determine the largest b_{max} value, $\overline{b}_{max}$, for a transactional data stream ($\mathcal{D}_i$), that is represented by the representative data stream ($\mathcal{D}_r$), with an arbitrary minimum support s.

s (%)	n_0	T10.I4.D1000K		T15.I6.D1000K	
		$\mathcal{F}_{max}$	c_{max}	$\mathcal{F}_{max}$	c_{max}
0.1	7,991	59,385	0.00006	454,092	0.00045
0.2	3,996	6,874	0.00005	19,690	0.00015
0.4	1,998	875	0.00006	1,722	0.00011
0.6	1,332	233	0.00005	660	0.00014
0.8	999	71	0.00004	245	0.00012
1.0	799	20	0.00002	102	0.00010

Table 10: The c_{max} for $|\mathcal{F}|$

s (%)	n_0	b_{max} (T10.I4)	b_{max} (T15.I6)
0.1	7,991	7.43	56.82
0.2	3,996	1.72	4.93
0.4	1,998	0.44	0.86
0.6	1,332	0.17	0.50
0.8	999	0.07	0.25
1.0	799	0.03	0.13

Table 11: The b_{max} for Table 10

$$\overline{b}_{max} = (\overline{c}_{max}/s^3)/n_0.$$

Finally, we identify $c_u = \overline{b}_{max}$.

Remark 4 *For mining frequent patterns from a transactional data stream, the number of entries in $\mathcal{F}$ can empirically be bounded by $c_u \cdot n_0$ where c_u is selected using a representative data stream.*

Remark 4 is important because it states that in fact the number of potential frequent itemsets can be possibly bounded by $c_u \cdot n_0$. In addition, c_u is a considerably small constant, and is not necessarily related with the domain of I unique items. Recall the number of potential frequent itemsets can be up to 2^I for a transactional data stream in a domain of I unique items. In other words, it states that the memory required for $|\mathcal{F}|$ is possible to be multiplication of n_0 (linearity).

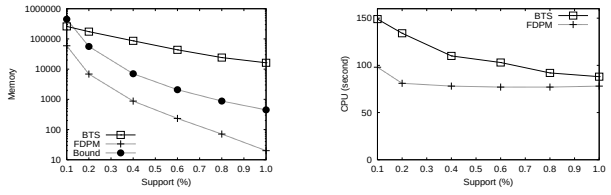
The value c_u is used as a way to determine pruning (line 7) in Algorithm 2. In addition, we are able to do *eager* pruning. There are patterns that we can possibly prune if the running error $\epsilon_n > s$ in terms of n observations. It is based on Definition 1. A pattern X is potential infrequent if $\overline{sup}(X)/n < s - \epsilon_n$. Because $\overline{sup}(X) \geq 0$, $\epsilon_n < s$ means no patterns can be pruned.

Remark 5 *Based on Algorithm 2, the empirical upper bound for transactional data streams is $O(1/s^3)$ if c_u is selected from a representative data stream with a fixed δ.*

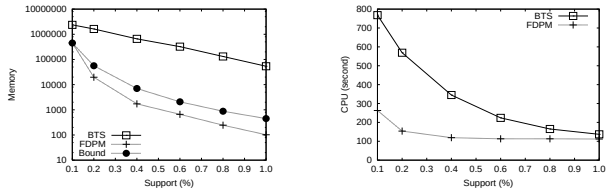
The Remark 5 is based on $u_F = c_u \cdot n_0$ where n_0 is a denominator of c_u. Note: the bound, $(2+2\ln(2/\delta))/s$, for Algorithm 1 is an exact bound.

6 Performance Study II: Mining Frequent Itemsets

We report our experimental results for frequent itemsets mining. For frequent itemsets mining, we implemented our false-negative oriented algorithm `FDPM`



(a) Mem (T10.I4.D1000K) (b) CPU (T10.I4.D1000K)



(c) Mem (T15.I6.D1000K) (d) CPU (T15.I6.D1000K)

Figure 4: Varying s ($\epsilon = s/10$, $\delta = 0.1$)

(Algorithm 2). The idea of probabilistic-inplace is also used in Algorithm 2. For comparison purposes, we implemented Manku and Motwani's false-positive oriented three module system BTS (Buffer-Trie-SetGen). The Apriori implementation we used is available from, http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html#assoc, which is used in many commercial data mining tools. Its version is 4.07.

For testing frequent itemsets mining, we generate transactional data streams using IBM data generator [12]. We mainly use two datasets, T10.I4.D1000K and T15.I6.D1000K with 10K unique items (as default). We process transactional data in batches. The size of a batch is 50,000 transactions. The parameter k used in FDPM and β used in BTS are adjusted accordingly.

6.1 Effect of Minimum Support

We fix $\epsilon = s/10$ and $\delta = 0.1$, and vary s from 0.1% to 1.0%. Figure 4 (a) and (b) show memory consumption and CPU for T10.I4.D1000K, and Figure 4 (c) and (d) show memory consumption and CPU for T15.I6.D1000K. Recall memory consumption is the number of counters. In addition to BTS and FDPM, we show our empirical bounds (Bound) of FDPM, which is computed by $c_u \cdot n_0$ and c_u is computed using $\overline{c}_{max} = 0.00045$.

As shown in Figure 4, FDPM significantly outperforms BTS. In the worst case, when $s = 0.1\%$, FDPM only consumes 59,385 entries for T10.I4.D1000K and 454,092 entries for T15.I6.D1000K, whereas BTS consumes 259,581 and 2,373,968, accordingly. In the best case, when $s = 1.0\%$, FDPM consumes only 20 entries for T10.I4.D1000K and 102 entries for T15.I6.D1000K, whereas BTS consumes 16,218 and 53,767, accordingly. FDPM significantly outperforms BTS for both memory consumption and CPU cost. Figure 4 (a) and (c) show that the memory consumption is bounded by our empirical bound $c_u \cdot n_0$.

s (%)	BTS		FDPM	
	R	P	R	P
0.1	1	0.85	1	1
0.2	1	0.84	1	1
0.4	1	0.70	0.99	1
0.6	1	0.68	0.99	1
0.8	1	0.46	1	1
1.0	1	0.55	1	1

Table 12: Varying s (T10.I4.D1000K, $\epsilon = s/10$, $\delta = 0.1$)

s (%)	BTS		FDPM	
	R	P	R	P
0.1	1	0.72	1	1
0.2	1	0.91	1	1
0.4	1	0.80	0.99	1
0.6	1	0.72	0.99	1
0.8	1	0.64	0.99	1
1.0	1	0.58	0.95	1

Table 13: Varying s (T15.I6.D1000K, $\epsilon = s/10$, $\delta = 0.1$)

Table 12 and Table 13 show the recall and precision for Figure 4. Here, FDPM achieves high recall (at least 95%) and ensures 100% precision.

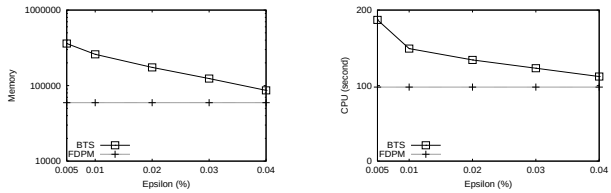
6.2 Effect of Error Control

We fix $s = 0.1\%$ and $\delta = 0.1$, and vary ϵ. Figure 5 (a) and (b) show memory consumption and CPU for T10.I4.D1000K, and Figure 5 (c) and (d) show memory consumption and CPU for T15.I6.D1000K. The recall and precision are shown in in Table 14 and Table 15.

As shown in Figure 5, our false-negative oriented algorithm FDPM is not influenced by ϵ. Both memory consumption and CPU are constant while varying ϵ. FDPM only needs 59,385 and 454,092 entries for T10.I4.D1000K and T15.I6.D1000K, respectively. However, ϵ has great impacts on the false-positive oriented approach BTS. Its memory consumption increases while ϵ decreases. When $\epsilon = 0.005\%$, BTS needs large memory to keep 360,476 entries for T10.I4.D1000K, and 3,196,445 entries for T15.I6.D1000K, and achieves 93% and 85% precision, respectively. When $\epsilon = 0.04\%$, BTS needs small memory to keep 86,537 entries for T10.I4.D1000K, and 659,233 entries for T15.I6.D1000K. But, BTS can only have 32% precision, and 16% precision for T10.I4.D1000K and T15.I6.D1000K, respectively. In sequent, BTS faces a dilemma: a little increase of ϵ will make the number of false-positive items large, and a little decrease of ϵ will make memory consumption large.

ϵ (%)	BTS		FDPM	
	R	P	R	P
0.040	1	0.32	1	1
0.030	1	0.42	1	1
0.020	1	0.58	1	1
0.010	1	0.85	1	1
0.005	1	0.93	1	1

Table 14: Varying ϵ (T10.I4.D1000K, $s = 0.1\%$, $\delta = 0.1$)



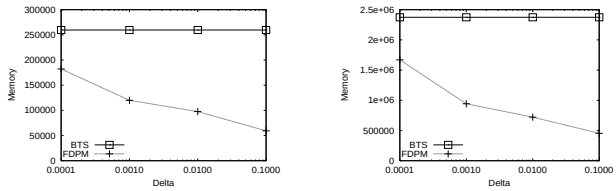
(a) Mem (T10.I4.D1000K) (b) CPU (T10.I4.D1000K)

(c) Mem (T15.I6.D1000K) (d) CPU (T15.I6.D1000K)

Figure 5: Varying ϵ ($s = 0.1\%$, $\delta = 0.1$)

ϵ (%)	BTS		FDPM	
	R	P	R	P
0.040	1	0.16	1	1
0.030	1	0.27	1	1
0.020	1	0.48	1	1
0.010	1	0.72	1	1
0.005	1	0.85	1	1

Table 15: Varying ϵ (T15.I6.D1000K, $s = 0.1\%$, $\delta = 0.1$)



(a) Mem (T10.I4.D1000K) (b) Mem (T15.I6.D1000K)

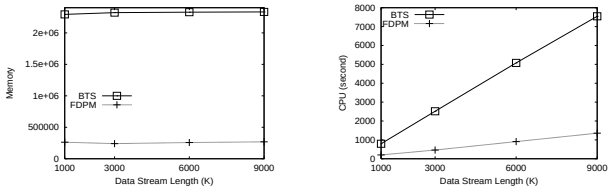
Figure 6: Varying δ

δ	BTS		FDPM	
	R	P	R	P
0.1	1	0.85	1	1
0.01	1	0.85	1	1
0.001	1	0.85	1	1
0.0001	1	0.85	1	1

Table 16: Varying δ (T10.I4.D1000K)

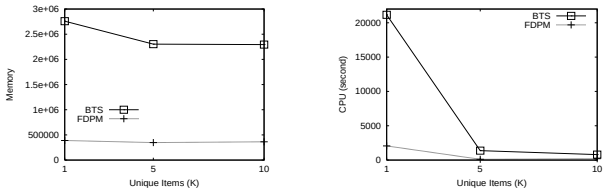
$\|\mathcal{D}\|$	BTS		FDPM	
	R	P	R	P
1000K	1	0.72	0.99	1
3000K	1	0.72	0.99	1
6000K	1	0.72	0.99	1
9000K	1	0.72	1	1

Table 17: Varying length (T15.I6.D1000K)



(a) Memory (b) CPU

Figure 7: Varying length (T15.I6.D1000K)



(a) Memory (b) CPU

Figure 8: Varying domain (T15.I6.D1000K)

6.3 Effect of Reliability Control

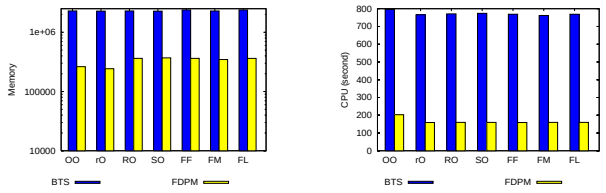
We fix $s = 0.1\%$ and $\epsilon = s/10$, and vary δ. We compare FDPM with BTS, and show results in Figure 6. As expected, varying δ does not affect BTS, because it treats $\delta = 0$. As shown in Figure 6, while the reliability increases (smaller δ), the memory consumption of FDPM increases, because it uses δ to approximate the memory consumption. Even when $\delta = 0.0001$, the memory consumption of FDPM is much smaller than the memory consumption of BTS. As shown in Table 16, FDPM achieves 100% recall and 100% precision, even with $\delta = 0.1$, while BTS achieves 100% recall and 85% precision. FDPM outperforms BTS.

6.4 The Impacts of Data Stream Length

We test the impacts of the data stream length on FDPM. We fix $s = 0.1\%$ ($\epsilon = s/10$, $\delta = 0.1$), and vary the length of T15.I6.D1000K from 1000K to 9000K. The memory consumption and CPU cost are shown in Figure 7. FDPM significantly outperforms BTS. When dealing with a 9000K data stream, BTS consumes 2,333,510 entries, while as FDPM consumes only about its 10%, 269,126. Also, BTS requires 7,545 seconds to process it, whereas FDPM only needs 1,355 seconds. As shown in Table 17, FDPM guarantees high recall and precision (almost 100%). The precision of BTS is only 72%.

$\|I\|$	BTS		FDPM	
	R	P	R	P
1K	1	0.76	0.99	1
5K	1	0.71	0.99	1
10K	1	0.72	0.99	1

Table 18: Varying domain (T15.I6.D1000K)



(a) Memory (b) CPU

Figure 9: Impacts of data arrival order

Data	BTS		FDPM	
	R	P	R	P
OO	1	0.72	0.99	1
rO	1	0.72	0.99	1
RO	1	0.72	0.97	1
SO	1	0.72	0.99	1
FF	1	0.72	0.99	1
FM	1	0.72	0.99	1
FL	1	0.72	0.96	1

Table 19: Impacts of data arrival order

6.5 The Impacts of Unique Items

We test the impacts of the domain sizes, 1K, 5K and 10K, using T15.I6.D1000K, where $s = 0.1\%$, $\epsilon = s/10$ and $\delta = 0.1$. The results are shown in Figure 8 and Table 18. When the data is dense (1K), there are many patterns. BTS needs 10 times of CPU than FDPM. Also, as shown in Table 18, FDPM ensures 100% precision and high recall (99%). BTS can only achieve about 71% precision.

6.6 The Impacts of Data Arrival Order

We test several data arrival orders using T15.I6.D1000K: OO (Original Order), rO (reverse Order), RO (Random Order), SO (segment-based random order, FF (Frequent First), FM (Frequent Middle), FL (Frequent Last) where $s = 0.1\%$, $\epsilon = s/10$, and $\delta = 0.1$. As shown in Figure 9 and Table 19. FDPM and BTS are insensitive to data arrival orders regarding frequent itemsets mining. FDPM achieves high recall (almost all 99% only one 97%) and ensures 100% precision. The precision of BTS is low (72%). In addition, FDPM outperforms BTS in terms of CPU and memory consumption.

7 Conclusion

In this paper, we study the problem of mining frequent patterns from transactional data streams, the problem of FDPM. While most existing algorithms in mining frequent items for data streams using false-positive oriented approaches to control the error on the estimated frequency of mined patterns and memory requirement, we explored a new paradigm in FDPM, the false-negative oriented approach. That is, we control the data mining process by limiting the probability of a frequent pattern that misses in the result, but all mined patterns are frequent. We developed both frequent item and itemset mining algorithms using the Chernoff bound. The bound enables us pruning infrequent patterns from the continuously arriving transactions with the guarantee of the required recall rate of frequent patterns. The performance study demonstrated the effectiveness and efficiency of our false-negative oriented approach which uses a running error, ϵ_n, to prune infrequent item(set)s, and uses δ to control memory space.

The Chernoff bound assumes some underlying property of the underlying distributions of the data. Although the current performance study indicated that even the data does not follow strictly the assumptions, the bound is surprisingly effective. One of our immediate future work is to further study the data distribution issues and explore possible theoretical bounds for frequent data stream pattern mining.

Acknowledgment

The authors would like to thank Zhenjie Zhang for his contribution to this project. The work described in this paper was supported by grant from the Research Grants Council of the Hong Kong SAR, China (CUHK4229/01E).

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] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of ACM STOC*, 1996.

[3] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of the Intl. Colloquium on Automata, Languages and Programming (ICALP)*, pages 693 – 703, 2002.

[4] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of mathematical Statistics*, 23(4):493–507, 1952.

[5] S. Cohen and Y. Matias. Spectral bloom filter. In *Proc. of ACM SIGMOD*, 2003.

[6] G. Cormode and S.Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. In *Proc. of 22nd ACM Symposium on Principles of Database Systems (PODS)*, pages 296 – 306, 2003.

[7] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *13th Annual ACM-SIAM Symp. on Discrete Algorithms*, 2002.

[8] E. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proc. of 10th Annual European Symposium on Algorithms*, pages 348 – 360, 2002.

[9] J. Feigenbaum and S. Kannan. An approximate l1-difference algorithm for massive data streams. In *IEEE Symposium on Foundations of Computer Science*, 1999.

[10] P. Flajolet and G. N. Martin. Probabilistic counting algorithms. *J. of Comp. and Sys. Sci*, (31):182–209, 1985.

[11] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *Tutorial in 28th Intl. Conf. on Very Large Data Bases*, 2002.

[12] 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.

[13] H. Toivonen. Sampling large databases for association rules. In *Proc. of 22nd Intl. Conf. on Very Large Data Bases*, pages 134 – 145, 1996.

[14] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.