An Algorithm for In-Core Frequent Itemset Mining on Streaming Data

Ruoming Jin
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
jinr@cis.ohio-state.edu

Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
agrawal@cis.ohio-state.edu

ABSTRACT

Frequent itemset mining is a core data mining operation and has been extensively studied over the last decade. This paper takes a new approach for this problem and makes two major contributions. First, we present a one pass algorithm for frequent itemset mining, which has deterministic bounds on the accuracy, and does not require any out-of-core summary structure. Second, because our one pass algorithm does not produce any false negatives, it can be easily extended to a two pass accurate algorithm. Our two pass algorithm is very memory efficient, and allows mining of datasets with large number of distinct items and/or very low support levels.

Our detailed experimental evaluation on synthetic and real datasets shows the following. First, our one pass algorithm is very accurate in practice. Second, our algorithm requires significantly lower memory than Manku and Motwani's one pass algorithm and the multi-pass apriori algorithm. Our two pass algorithm outperforms apriori and FP-tree when the number of distinct items is large and/or support levels are very low. In other cases, it is quite competitive, with possible exception of cases where the average length of frequent itemsets is quite high.

1. INTRODUCTION

Frequent itemset mining is a core data mining operation and has been extensively studied over the last decade [1, 13, 14, 23, 24]. Algorithms for frequent itemset mining form the basis for algorithms for a number of other mining problems, including association mining, correlations mining, and mining sequential and emerging patterns [14].

Algorithms for frequent itemset mining have typically been developed for datasets stored in persistent storage and involve two or more passes over the dataset. Recently, there has been much interest in data arriving in the form of continuous and infinite data streams. In a streaming environment, a mining algorithm must take only a single pass over the data. Such algorithms can only guarantee an approximate result.

In this paper, we present a new approach for frequent itemset mining. Our work has two main contributions:

In-core Mining in Streaming Environment: We present a single pass algorithm for frequent itemset mining in a streaming environment. Our algorithm has provable deterministic bounds on accuracy. Unlike the only other existing work in this area that we are familiar with [18], our algorithm does not require any out-of-core summary structure. We believe that this is a very desirable property, since stream mining algorithms may need to be executed in small and mobile devices, which do not have attached disks for storing an out-of-core summary structure.

Memory Efficient Accurate Mining: A key limitation of the existing work on frequent itemset mining has been the high memory requirements when the number of distinct items is large and/or the support level desired is quite low. Our single pass algorithm has a property that it does not produce *false negatives*, i.e., all frequent itemsets with desired support level are reported. The false positives reported by our algorithm can be easily removed through a second pass on the dataset. Our two pass algorithm provides high memory efficiency, while not compromising accuracy in any way.

Our work derives from the recent work by Karp *et al.* on determining frequent items (or 1-itemsets) [17]. They present a two pass algorithm for this purpose, which requires only $(1/\theta)$ memory, where θ is the desired support or frequency level. Their first pass computes a superset of frequent items, and the second pass eliminates any false positives. Our work addresses three major challenges in applying their ideas for frequent itemset mining in a streaming environment. First, we have developed a method for finding frequent k-itemsets, while still keeping the memory requirements limited. Second, we have developed a way to have a bound on the superset computed after the first pass. Third, we have developed a new data structure and a number of other implementation optimizations to support efficient execution.

Our algorithm takes as input a parameter ϵ. Given the desired support level θ, our one pass algorithm reports all itemsets occurring with frequency level θ, and does not include any itemset occurring with frequency level less than $(1-\epsilon)\theta$. In the process, the memory requirements increase proportional to $1/\epsilon$.

To efficiently implement the new algorithm, we have also designed a new data structure, referred to as *TreeHash*. This data structure implements a prefix tree using a hash table. It has the compactness of a prefix tree and allows easy deletions like a hash table.

We have carried out a detailed evaluation using both synthetic and real datasets. Our results can be summarized as follows.

- Our one pass algorithm is very accurate in practice. Even when ϵ is 1, the accuracy is 94% or higher, and in fact 100% in several cases. Using $\epsilon = .75$ results in an accuracy of 98% or higher in all cases.
- Our algorithm is very memory efficient. For example, using the T10.I4.N10K dataset and a support level of 1%, we can consistently handle 4 million to 20 million transactions with less than 2.5 MB main memory. In comparison, Manku and Motwani's algorithm [18] requires an out-of-core data-structures on top of a 44 MB buffer to process 1 million transactions.

- The algorithm can handle large number of distinct items and small support levels using a reasonable amount of memory. For example, a dataset with 100,000 distinct items and a support level of 0.05% could be handled with less than 200 MB main memory, a factor of 5 improvement over apriori.

The rest of the paper is organized as follows. In Section 2, we present our new algorithm. Theoretical properties of the algorithm are established in Section 3. Details of our implementation and the data-structures used are presented in Section 4. Detailed experimental evaluation is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2. ALGORITHM

This section describes our algorithm for in-core and memory efficient frequent itemset mining.

2.1 Basic Approach

Our work is derived from the recent work by Karp, Papadimitriou and Shenker on finding frequent elements (or 1-item sets) [17]. Formally, given a sequence of length N and a threshold θ ($0 < \theta < 1$), the goal of their work is to determine the elements that occur with frequency greater than $N\theta$. In the process, their algorithm requires only $O(1/\theta)$ memory.

The basic idea on which their work develops is as follows. Suppose, $\theta = 0.5$, i.e, we want to find a *majority element*. A trivial algorithm for this will involve counting the frequency of all distinct elements, and checking if any of them is the majority element. If there are n distinct elements, this will require $O(n)$ memory. Instead, the following algorithm could be used. We find two distinct elements and eliminate them from the sequence. We repeat this process until only one distinct element remains in the sequence. It is easy to see that the remaining distinct element in the sequence is the only candidate for being the majority element. We can take another pass over the original sequence and check if the frequency of this element is greater than $N/2$.

This idea can be generalized to an arbitrary value of θ. We can proceed as follows. At any given time, we maintain a set K of frequently occurring items and their counts. Initially, this set is empty. As we read an element from the sequence, we either increment its count in the set K, or insert it in the set with a count of 1. Thus, the size of the set K can keep growing. To bound the memory requirements, we do a special processing when $|K| > 1/\theta$. We decrement the count of each element in the set K, and delete elements whose count has becomes zero.

Now, let us consider the set K after the entire sequence has been processed. The key property is that any element which occurs at least $N\theta$ times in the sequence is in the set K. Consider any element that occurs f times in the sequence, but is not in K. Each occurrence of this element is eliminated together with more than $1/\theta - 1$ occurrences of other elements. Thus, at least a total of f/θ elements are eliminated. However, since $f/\theta < N$, we have $f < N\theta$.

Note, however, that not all elements occurring in K need to have frequency greater than $N\theta$. Thus, the set K is a superset of the frequent items we are interested in. To find the precise set of frequent items, another pass can be taken on the sequence, and the frequency of all elements in the set K can be counted. The total memory requirements for the entire process is $O(1/\theta)$.

In this paper, we build a frequent itemset mining algorithm using the above basic idea. As we stated previously, we have two goals. First, we want to have a memory efficient two pass algorithm for frequent itemset mining. Second, we want a one pass algorithm with a provable bound on the accuracy.

Thus, we see the following two challenges in developing our algorithm:

- Computing k-itemsets approximately after the first pass and accurately after the second pass, without requiring any out-of-core or large summary structure. This can be particularly challenging, because the number of candidate k-itemsets can be very large. For example, if each transaction had the length l and we were interested in frequent 3-itemsets, a straightforward application of the above idea will require $(1/\theta) \times^{l} C_3$ space, which may not be very efficient.

- Ensuring a provable bound on the accuracy of the results after the first pass on the dataset. In streaming environments, second pass on the dataset is usually not feasible. Therefore, it is important that the set K computed above does not contain many false positives.

2.2 Algorithm Description

```
StreamMining(Stream $\mathcal{D}$)
 global $Lattice\ \mathcal{L}$;
 local $Buffer\ \mathcal{T}$;
 local $Transaction\ t$;
 $\mathcal{L} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$;
 $f \leftarrow 0$; $\{*\ Average\ 2-itemset\ per\ transaction\ *\}$
 $c \leftarrow 0$; $\{*\ NUmbers\ of\ calls \rightarrow\ crossOver\ *\}$
 foreach $(t \in \mathcal{D})$
     $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\}$;
     $Update(t, \mathcal{L}, 1)$;
     $Update(t, \mathcal{L}, 2)$;
     $f \leftarrow TwoItemsetPerTransaction(t)$;
     if $|\mathcal{L}_2| \geq \lceil 1/\theta\epsilon \rceil \cdot f$
         $CrossOver(\mathcal{L}, 2)$;
         $c++$;
         $i \leftarrow 2$;
         while $\mathcal{L}_i \neq \emptyset$
             $i++$;
             foreach $(t \in \mathcal{T})$
                 $Update(t, \mathcal{L}, i)$;
             $CrossOver(\mathcal{L}, i)$;
         $\mathcal{T} \leftarrow \emptyset$;
 while $c < \lfloor |\mathcal{D}| \cdot \theta \rfloor$
     $c++$;
     foreach $(\mathcal{L}_i \neq \emptyset)$
         $CrossOver(\mathcal{L}, i)$;
 $Output(\mathcal{L})$;
```

Figure 1: StreamMining Description

We now describe our algorithm for in-core frequent itemset mining on streaming data. Our algorithm is referred to as *StreamMining* and is outlined in Figure 1. The key subroutines used as part of this algorithm are shown in Figure 2. Here, we are focusing on the one pass algorithm which can be applied on streaming data. This algorithm can be trivially extended to a two pass accurate algorithm.

As we stated previously, one of the challenges in applying the idea from Karp *et al.* to frequent itemset mining is the potentially large number of frequent i-itemsets. Specifically, a direct application of the idea will require $(1/\theta) \times^{l} C_i$ space, where l is the length of each transaction. In comparison, most of the existing

```
Update(Transaction t, Lattice $\mathcal{L}$, i )
 for $all\ i\ subsets\ s\ of\ t$
     if $s \in \mathcal{L}_i$
         $s.count++$;
       else if $i \leq 2$
         $\mathcal{L}_i.insert(s)$;
       else if $all\ i-1\ subsets\ of\ s \in L_{i-1}$
         $\mathcal{L}_i.insert(s)$;

CrossOver(Lattice $\mathcal{L}$, i)
 foreach $i\ itemsets\ s \in \mathcal{L}_i$
     $s.count--$;
     if $s.count = 0$
         $\mathcal{L}_i.delete(s)$;

TwoItemsetPerTransaction(Transaction t)
 global $X$; {$* Number\ of\ 2\ Itemset *$}
 global $N$; {$* Number\ of\ Transactions *$}
 local $f$; {$* Average\ 2\ Itemset\ Per\ Transaction *$}
 $N++$;
 $X \leftarrow X + \binom{|t|}{2}$;
 $f \leftarrow \lceil X/N \rceil$;
 if $|\mathcal{L}_2| \geq \lceil 1/\theta\epsilon \rceil \cdot f$
     $N \leftarrow N - \lceil 1/\theta\epsilon \rceil$;
     $X \leftarrow X - \lceil 1/\theta\epsilon \rceil \cdot f$;
 $return\ f$;
```

Figure 2: Subroutines Description

work on frequent itemset mining uses the *apriori* property [1], i.e., an i-itemset can be frequent only if all subsets of this itemset are frequent. One of the drawbacks of this approach has been the large number of 2-itemsets, especially when the number of distinct items is large, and θ is small.

Therefore, our algorithm uses a *hybrid* approach. We use the idea from Karp *et al.* to reduce the memory requirements for determining the frequent 2-itemsets. Then, we use such a reduced set of frequent 2-itemsets and the apriori property to reduce the number of i-itemsets, for $i > 2$.

We initially introduce some terminology. We are mining a stream of transactions $\mathcal{D}$. Each transaction t in this stream comprises a set of *items*. The algorithm takes as input two parameters. θ is the support level, i.e. the minimum frequency with which an itemset should occur to be considered frequent. ϵ is a factor that determines the accuracy of the one pass algorithm. We have $0 < \theta < 1$ and $0 < \epsilon \leq 1$.

To store and manipulate the candidate frequent itemsets during any stage of the algorithm, a lattice $\mathcal{L}$ is maintained.

$$\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_k$$

where, k is largest frequent itemset, and $L_i, 1 \leq i \leq k$ comprises the potential frequent i-itemsets.

The algorithm maintains a buffer $\mathcal{T}$ which stores the recently received transactions. Initially, the buffer is empty. When a new transaction t arrives, we put it in $\mathcal{T}$. Next, we call the *Update* routine to increment counts in $\mathcal{L}_1$ and $\mathcal{L}_2$. This routine simply updates the count of 1-itemsets and 2-itemsets that are already in $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Other 1-itemsets and 2-itemsets that are in the transaction t are inserted in the sets $\mathcal{L}_1$ and $\mathcal{L}_2$.

The size of the set $\mathcal{L}_1$ is bound by the number of distinct items in the dataset, and is typically not very large. However, the size of $\mathcal{L}_2$ can get very large. Therefore, when the size of $\mathcal{L}_2$ is beyond a certain *threshold*, we call the procedure *CrossOver*. The value of this threshold is crucial for accuracy and memory efficiency of our algorithm, and is discussed later.

The function *CrossOver* can be invoked for any value of i. It examines all candidate i-itemsets in the set $\mathcal{L}_i$. The counts of all candidates k-itemsets are decremented, and the itemsets whose count becomes zero are deleted from the set.

After invoking *CrossOver* on $\mathcal{L}_2$, we have a reduced set of 2-itemsets. We use these for generating i-itemsets, for $i > 2$. This process is carried out level-wise, i.e, it proceeds from 3-itemsets to the largest potential frequent itemsets. For each transaction in the buffer T, we enumerate all i-subsets. For any i-subset that is already in $\mathcal{L}$, the process will be the same as for a 2-itemset, i.e, we will simply increment the count. However, an i-subset that is not in $\mathcal{L}$ will be inserted in L only if all of its $i-1$ subsets are in $\mathcal{L}$ as well. Thus, we use the *apriori* property.

After updating i-itemsets in $\mathcal{L}$, we will invoke the *CrossOver* routine. Thus, the itemsets whose count is only 1 will be deleted from the lattice. This procedure will continue until there are no frequent k-itemsets in $\mathcal{L}$. At the end of this, we clear the buffer, and start processing new transactions in the stream.

Now, we discuss how we choose the value of the threshold that determines how frequently *CrossOver* is called. Let $|\mathcal{D}|$ be the number of transactions in the stream when the results are output. Initially, suppose that we are not interested in having a bound on the false positives that may be output by the algorithm. In this case, we choose $\epsilon = 1$. If each transaction was a set of two items, only the count of a single 2-itemset could be updated by processing it. Then, the size $1/\theta$ would have been sufficient to ensure that no frequent 2-itemset is excluded from $\mathcal{L}_2$. However, the transactions could have an arbitrary length. A transaction of length l can have ${}^{l}C_2$ distinct 2-itemsets. Therefore, we maintain a factor f, which is the weighted average of the number of 2-itemsets that each transaction processed so far has. This weighted average is computed by giving higher weightage to recent transactions. The details are shown in the pseudocode for the routine *TwoItemsetPerTransaction*. A size f/θ can now ensure that $\mathcal{L}_2$ does not have any false negatives.

In a streaming environment, we are interested in having a bound on the accuracy of the results. In our case, if we choose $\epsilon < 1$, we guarantee that no itemset occurring with frequency less than $(1 - \epsilon)\theta|\mathcal{D}|$ will be in the set $\mathcal{L}$. In the process, however, the memory requirements can increase by a factor proportional to $1/\epsilon$. To ensure such a bound on the accuracy, we initially reduce the frequency of invocation of *CrossOver* routine by a factor proportional to $1/\epsilon$. As a result, any itemset occurring with the frequency greater than $\epsilon\theta|\mathcal{D}|$ will be included in $\mathcal{L}$. Then, before outputting the results, any itemset remaining in $\mathcal{L}$ with a count less than $(1-\epsilon)\theta|\mathcal{D}|$ is deleted. This is achieved through the last *while* loop in the algorithm shown in Figure 1. The net result is that 1) any itemset occurring with a frequency greater than $\theta|\mathcal{D}|$ is definitely included in $\mathcal{L}$, and 2) any itemset occurring with a frequency less than $(1-\epsilon)\theta|\mathcal{D}|$ is definitely not included.

3. THEORETICAL ANALYSIS

In this section, we establish a number of results on the correctness, accuracy, and memory efficiency of our algorithm.

As stated previously, we are mining a stream $\mathcal{D}$ and the number of transactions seen when the results are output is $|\mathcal{D}|$. θ and ϵ are as defined earlier. $\mathcal{L}_i$ is the set of frequent i-itemsets output by the one pass algorithm. Given any frequency $f|\mathcal{D}|$, let $\mathcal{L}_i^f$ be the actual set of i-itemsets occurring in $\mathcal{D}$.

3.1 Correctness

LEMMA 1. $\mathcal{L}_2^\theta \subseteq \mathcal{L}_2$.

Proof: The proof has two parts. First, we can see that *CrossOver* is called at most $\lfloor \theta\epsilon|\mathcal{D}| \rfloor$ times in the *foreach* loop. Thus, any 2-itemset that appears more than $\theta\epsilon|\mathcal{D}|$ times will stay in the set $\mathcal{L}_2$. Then, after the *while* loop, the total invocations of *CrossOver* will be at most $\theta|\mathcal{D}|$. Therefore, we have $\mathcal{L}_2^\theta \subseteq \mathcal{L}_2$.

LEMMA 2. *For any 2-itemset $s \in \mathcal{L}_2$, $s \in \mathcal{L}_2^{(1-\epsilon)\theta}$. In other words, s will appear at least $(1-\epsilon)\theta|\mathcal{D}|$ times in D.*

Proof: Before outputting the results, *CrossOver* is called at least $(1-\epsilon)\theta|\mathcal{D}|$ times in the *while* loop. Suppose there is an itemset appearing with a frequency less than $(1-\epsilon)\theta|\mathcal{D}|$ in the stream $|\mathcal{D}|$. The highest count it can have before the while loop is less than $(1-\epsilon)\theta|\mathcal{D}|$. Therefore, it will be removed from $\mathcal{L}_2$ before the results are output.

Putting Lemmas 1 and 2 together, we have following result.

LEMMA 3.

$$L_2^\theta \subseteq L_2 \subseteq L_2^{(1-\epsilon)\theta}$$

THEOREM 1. *For any $k \geq 2$, $L_k^\theta \subseteq L_k \subseteq L_k^{(1-\epsilon)\theta}$.*

Proof: We prove this inductively. The base case, $k = 2$, has been shown in the Lemma 3. Now, assume that the property holds true for L_{k-1}. To show that it is valid for L_k, we take two steps. First, assume there is no checking for subsets. Then, any k-itemsets which appears in the buffer will be inserted in the lattice. In this case, it is easy to see that the property holds for L_k.

Now let us consider why subset checking does not change the final results. Assume there is an k-itemset s appearing in a transaction t, and one of its $k-1$ subset is not in L_{k-1}. We can deduce that s must only appear once in the buffer, and it is not included in any other transactions besides t. Thus, we can see that s, if included in the lattice, would have been eliminated by the next invocation of *CrossOver*.

4. DATA STRUCTURES AND EFFICIENT IMPLEMENTATION

In this section, we discuss the data structure and other optimizations used for efficiently implementing our algorithm. Particularly, we address the challenges in efficient execution of *Update* and *CrossOver* routines.

4.1 Data Structure

An efficient data structure is required to maintain the lattice $\mathcal{L}$. Frequent itemset mining implementations often use a *prefix tree* for this purpose. However, our algorithm requires the ability to delete the itemsets efficiently, which is not possible using a prefix tree.

An obvious alternative is to use a hash table. Itemsets of different lengths can be mapped to hash buckets and stored there. However, this poses two problems. First, the total storage for large itemsets can be quite high. Second, comparing two itemsets of large length can be time consuming.

Thus, we need a data structure that is compact, and can allow the following operations efficiently: 1) insertion of a new itemset, 2) deletion of an itemset, 3) incrementing the count of an itemset, and 4) traversal of the lattice. We have developed a new data structure, which we refer to as *TreeHash*. Essentially, this data structure stores a prefix tree using hash tables. It has the benefit of easy deletion that a hash table allows, but it is also compact like a prefix tree.

To explain this data structure, we first review how a prefix tree is used to represent a lattice $\mathcal{L}$ of frequent itemsets. If an itemset R is in $\mathcal{L}$, then all subsets of R are also in $\mathcal{L}$. Thus, any *prefix subset* of R is also in L. Assume $R = \{r_1, r_2, \ldots, r_k\}$, its prefix subsets are $R_i = \{r_1, r_2, \cdots, r_i\}, i \leq k$. R_{k-1} is denoted as the *immediate prefix subset* of R. Clearly, $\emptyset$ is the immediate prefix subset of any singleton set. In a prefix tree, each node in the tree records an itemset, and its parent node records its immediate prefix subset.

Looking up a k-itemset in a prefix tree involves k operations, one for each of its prefix subsets. However, in a frequent itemset mining algorithm, if the count of an itemset needs to be incremented, count of all its prefix subsets need to be incremented as well. Thus, a prefix tree can allow compact storage and easy identification of an itemset, without requiring any additional work.

Our data structure TreeHash maintains a prefix tree using a hash table. A hashing function maps each node in the prefix tree to a hash bucket. For each itemset in a hash bucket, the following information is stored: 1) the length of the itemset, 2) hash address of its immediate prefix itemset, which is used for identifying this itemset, 3) A *serial number* to this itemset within this hash bucket, and 4) a count of the number of occurrences of the itemset.

Thus, the set of items is not explicitly maintained, allowing a compact representation. However, now two basic issues in using this data structure are: 1) given an itemset R, how we locate it in the hash table, and 2) given an entry in a hash bucket, how do we identify the itemset.

Both of the above problems are addressed inductively. We first consider the problem of mapping an itemset into a particular hash bucket. The empty set is trivially hashed into the hash table. Now, let R_{k-1} be hashed in the hash table, where its hash address (hash bucket number) is $H(R_{k-1})$ and its serial number is $s(R_{k-1})$. We use $H(R_{k-1})$, $s(R_{k-1})$, and r_k to determine the hash bucket for R. Within this bucket, the itemset is identified by the value $H(R_{k-1})$ that is stored there. Note that this requires that for two itemsets with the same $H(R_{k-1})$ (but different $s(R_{k-1})$), the hash bucket numbers output by the hash function are always different. This can be ensured by dividing the hash table into portions equal to the number of entries that can be stored in a bucket.

Now, we consider the problem of determining the itemset that a hash table entry represents. We have a hash element encoding R, which has the length k. Its identifier in the hash bucket is $H(R_{k-1})$, which points to the bucket where R's immediate prefix itemset R_{k-1} is stored. To identify R_{k-1}, we also need $s(R_{k-1})$. To reconstruct the itemset, we also need to determine r_k. Thus, if we have the hash function h such that

$$H(R_k) \ = \ h(H(R_{k-1}), s(R_{k-1}), r_k)$$

we need to have functions f and g such that

$$r_k \ = \ f(H(R_{k-1}), H(R_k))$$

and

$$s(R_{k-1}) \ = \ g(H(R_{k-1}), H(R_k))$$

4.2 Update and Delete Operations

One of the key operations in implementing our algorithm is deleting itemsets that do not occur frequently. Karp *et al.* [17] propose the following method. A linked list is maintained, where each entry represents a distinct count value of the itemsets. All of the elements which have the same count are organized as a double linked list and are attached to the first linked list. Thus, an increment operation will simply move the element from one double linked list to the next double linked list. The *CrossOver* routine will remove all of the elements in the double linked list which has the count value less than the desired value (typically 1). Theoretically, it provides an $O(1)$ computational cost to deal with each itemset and keeps the memory cost minimum. However, in practice, changing pointers while inserting and deleting elements from a double linked list results in high memory access costs.

We instead use the following approach. The counts of elements are not decremented during every invocation of *CrossOver*. Instead, they are decremented by p during every p^{th} invocation of *CrossOver*. Similarly, infrequent items are also typically removed after every p^{th} invocation of *CrossOver*. This, however, leads to the problem of determining when *CrossOver* must be invoked. In other words, we still need to know the number of 2-itemsets that would have been in the lattice even if deletion and decrement operations were performed during each of the previous invocations of *CrossOver*.

To address this problem, we maintain the following data structures. To count the correct number of 2-itemsets, we maintain a small array c which has the size $p+1$. Further, we keep the number of 2-itemsets which have the count $i, 1 \leq i \leq p$ in $c[i]$. We also record the total number of 2-itemsets that have count greater than p in $c[p+1]$.

Now, let us consider the *Update* operation. Assume that the count of the target 2-itemset is x. Besides incrementing the count by 1, we will also decrement $c[x]$ and increment $c[x+1]$, provided $x \leq p$. Let k be the number of times *CrossOver* has been invoked since the last time the counts were actually decremented. Here, $1 \leq k \leq p$. We can see that $c[i], k \leq i \leq p$, always has the correct number of 2-itemsets which would have the count $i-k$, if decrement was done during every invocation of *CrossOver*. Adding the values of $c[i]$, $k \leq i \leq p+1$ also gives the correct count of 2-itemsets in the lattice.

We also do not remove the infrequent itemsets every time *CrossOver* routine is invoked. Instead, the deletion happens in the following three ways. First, when we insert a new itemset in a bucket, if the bucket have some infrequent itemset present, it will be removed and the new itemset will be stored. Secondly, after every p invocations of *CrossOver*, we remove all infrequent itemsets in the lattice by examining the entire hash table. Finally, when the memory usage is beyond some threshold, infrequent itemsets are removed.

4.3 Other Optimizations

In the implementation of *StreamMining* used for our experiments, three additional optimizations are applied. The first is *online dataset trimming*. It was proposed by Park *et al.* [19] to reduce the transaction size. The basic idea is to use some statistics of each item in the transaction to determine if it can be part of a large frequent itemset. This method is only applicable if the dataset is in-core, but we found that it is very useful in the online context. The second is *reducing subset checking*. Recall that in our algorithm, the precondition for inserting a new itemset is that all of its subsets must be frequent. Our experience has shown that inserting with just testing for prefix subsets is more efficient. If an itemset is not frequent, it is typically deleted soon by the *CrossOver* routine.

Another optimization we implemented was based upon our experiences with the real dataset BMS-WebView-1 [24]. To motivate this, suppose we have two almost identical and large itemsets that appear physically very close to each other. Let s be the set of items in their intersection. Then, our algorithm will recognize any member of the power set of s to be a frequent itemset and insert it in the lattice. If s has 20 items, almost 1 million different itemsets will be added to the lattice, resulting in significant slow down to our algorithm. We believe that any one pass frequent itemset mining algorithm will have difficulty in handling such a case, whereas the performance of any multi-pass algorithm will not be impacted.

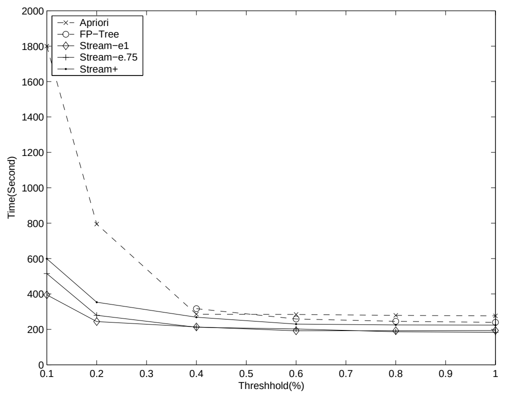

Figure 3: Execution Time with Changing Support Level (T10.I4.N10K Dataset)

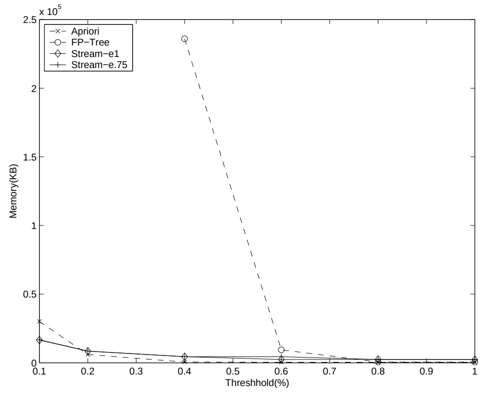

Figure 4: Memory Requirements with Changing Support Level (T10.I4.N10K Dataset)

We use a technique we call *online checking* to deal with such situations. When a new transaction arrives, we compare it with the transactions in the buffer $\mathcal{T}$. If two of them have a common and large subset, we put the new transaction in a *transaction pool*. After one pass of the data stream, we have a lattice as well as a separate pool of transactions. Now, we use the transaction pool to update counts of certain itemsets and insert new itemsets. Our implementation still ensures the accuracy properties we established earlier.

5. EXPERIMENTAL RESULTS

In this section, we evaluate our new algorithm using a number of synthetic and real datasets. We focus on a number of different aspects of our algorithm.

- Comparing the execution time and memory requirements of our one pass and two pass algorithm with those of apriori and fp-tree based algorithms.

- Evaluating the execution time and memory requirements of our new algorithms with increasing dataset size and decreasing support levels.

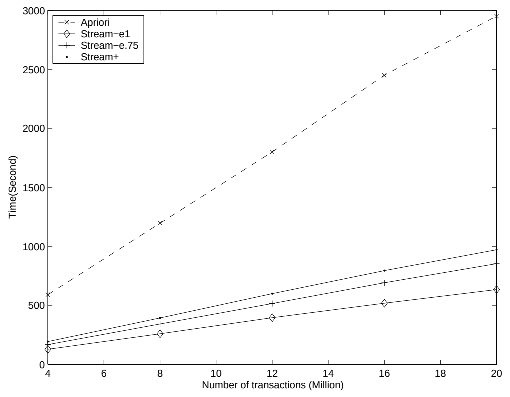

Figure 5: Execution Time with Increasing Dataset Size (threshold=0.1%, T10.I4.N10K Dataset)

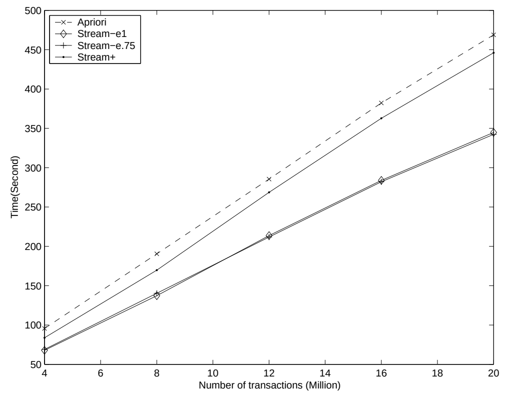

Figure 6: Execution Time with Increasing Dataset Size (threshold=0.4%, T10.I4.N10K Dataset)

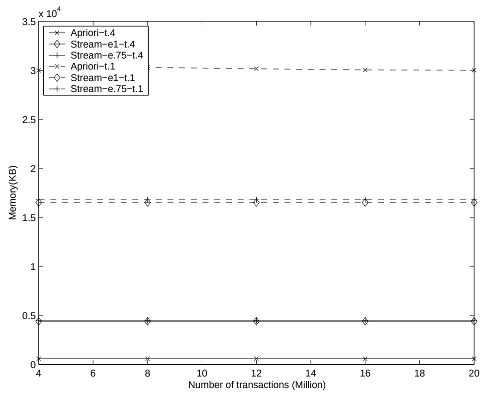

Figure 7: Memory Requirements with Increasing Dataset Size (T10.I4.N10K Dataset)

- Evaluating the accuracy of our algorithm with different levels of ϵ.

- Demonstrating the ability of our algorithm to handle very large number of distinct items and very low support levels.

For comparing our algorithm against the Apriori algorithm, we used a well-known public distribution from Borgelt [4]. Earlier versions of this code have been incorporated in a commercial data mining tool called Clementine. For comparisons with FP-tree based approach, the implementation we used is from Goethals [11]. All our experiments were conducted on a 933 MHz Pentium III machine with 512 MB main memory.

5.1 Synthetic Datasets

The synthetic datasets we used were generated using a tool from IBM [2]. Datasets generated from this tool have been widely used for evaluating frequent itemset and association mining implementations.

Initially, we focus on two datasets where conventional offline algorithm have performed well. We show that our algorithm can still be competitive, while allowing high accuracy on streaming data. Later, we show our algorithms ability to handle very large number of distinct itemsets and very low support levels.

The first dataset we used is T10I4.N10K. The number of distinct itemsets is 10,000, the average number of items per transaction is 10, and the average size of large itemsets is 4. We used three different versions of our algorithm. `Stream-e1` uses 1 as the value of ϵ and does not provide any theoretical bound on the accuracy. `Stream-e.75` uses .75 as the value of ϵ to provide a theoretical bound on the accuracy. `Stream+` is the two pass implementation that gives the accurate set of frequent itemsets and their frequency counts.

Figure 3 shows the execution times of apriori, fp-tree and our three versions as the support threshold is varied from 0.1% to 1.0%. The number of transactions is 12 million. Because of high memory requirements, fp-tree could not be executed with support levels lower than 0.4%. This limitation of the fp-tree appraoch has been identified by other experimental studies also [7]. Up to the support level of 0.4%, the execution times of all versions is quite similar. However, apriori's execution time increases rapidly when the support level is less than 0.4%. As expected, `Stream-e1` has the lowest execution time among all of our versions. The use of .75 as the value of ϵ increases the execution time by up to 25%. If a second pass is used, the total execution time is increased by up to 50%.

Figure 4 compares the memory requirements. Because the memory requirements of `Stream+` are identical to those of `Stream-e1`, this version is not shown separately in our memory requirements charts. The important property of our algorithm is that the memory requirements do not increase significantly as the support level is decreased.

Accuracy of an algorithm is defined as the fraction of reported frequent itemsets that are actually frequent. Obviously, the accuracy of apriori, fp-tree and `Stream+` is always 100%. With 12 million transactions, `Stream-e1` and `Stream-e.75` give accuracy of 100% with thresholds at 1%, .8%, .6%, and .4%. With thresholds of .2% amd .1%, `Stream-e1` has an accuracy of 95.8% and 97.8%, respectively. However, in both these cases, with .75 as the value of ϵ, the accuracy again becomes 100%.

Figures 5 and 6 examine the execution times as the dataset is increased. The threshold is kept at .4% and .1%, respectively. Because of the high memory requirements of fp-tree, our algorithm is only compared against apriori. When the support level is 0.1%,

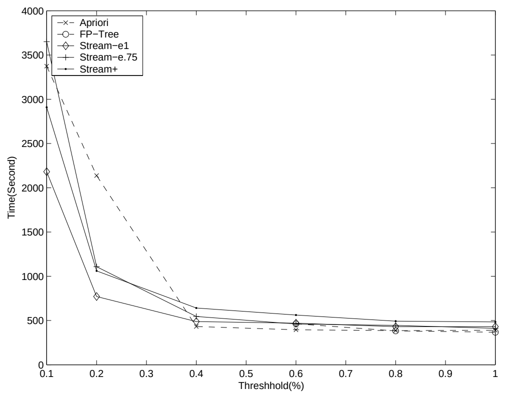

Figure 8: Execution Time with Changing Support Level (T15.I6.N10K Dataset)

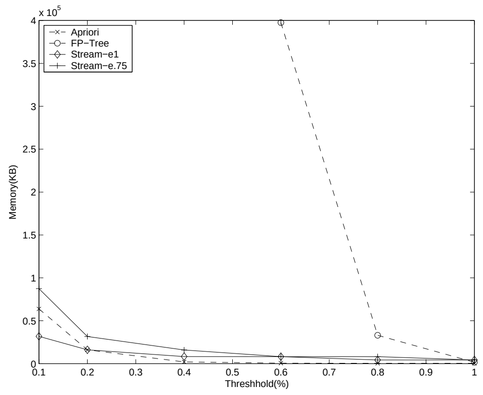

Figure 9: Memory Requirements with Changing Support Level (T15.I6.N10K Dataset)

our algorithm is up to an order of magnitude faster. The relative difference is smaller when the support level is 0.4%, but even our two pass version is faster than apriori. Even as the dataset size is varied, our one pass algorithms always give an accuracy of 100% when the threshold is .4%. With the threshold at .1%, the accuracy of `Stream-e.75` is again 100% in all cases. The accuracy of `Stream-e1` varies between 94.3% and 98.6%.

Figure 7 focuses on memory requirements with support levels of .4% and .1%. At the support level of .4%, apriori's memory requirements are lower than our versions. However, with threshold at .1%, our versions require less than half the memory. Moreover, it is important to note that with 10,000 distinct items and a support level of .1%, the total memory requirements are only around 17 MB. Thus, our algorithm is well suited for mining streaming data using a small device with only a limited memory.

The second dataset we use is T15.I6.N10K. We repeated the same set of experiments using this dataset. The results are shown in Figures 8, 9, 10, 11, and 12, respectively. The key difference between this dataset and the previous dataset is the the length of each transaction and each frequent itemset is higher. Because our algorithm needs to generate fairly accurate results after one pass, its ability to prune large itemsets is limited. As a result, our algorithm does not always outperform apriori with this dataset. However, our algorithm does maintain very high accuracy of results after one pass on the dataset. With 4 million, 8 million, 12 million, 16 million, or 20 million transactions, and with support levels of 1%, .8%, .6%,

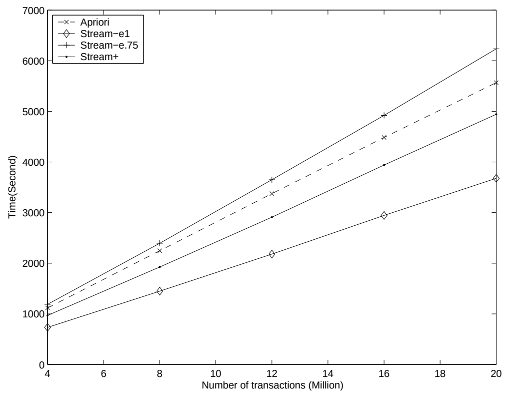

Figure 10: Execution Time with Increasing Dataset Size (threshold=0.1%, T15.I6.N10K Dataset)

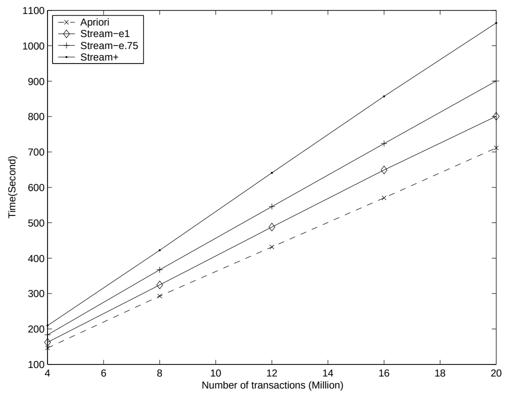

Figure 11: Execution Time with Increasing Dataset Size (threshold=0.4%, T15.I6.N10K) Dataset

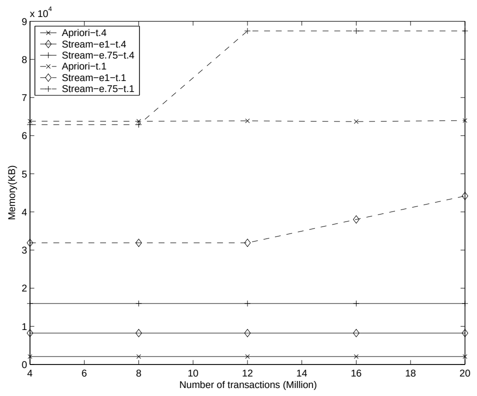

Figure 12: Memory Requirements with Increasing Dataset Size (T15.I6.N10K Dataset)

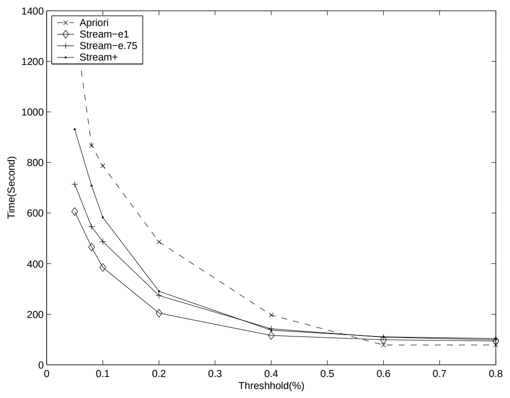

Figure 13: Execution Time with Changing Support Level (T25.I4.N100K Dataset)

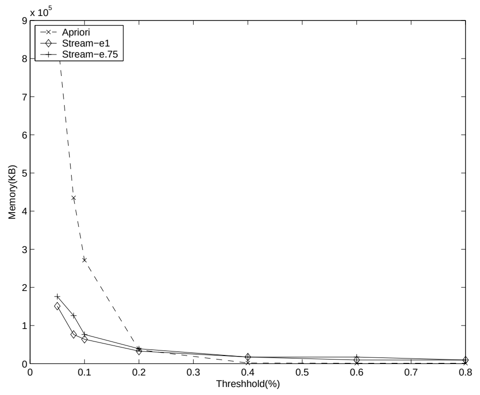

Figure 14: Memory Requirements with Changing Support Level (T25.I4.N100K Dataset)

or .4%, our `Stream-e1` always produces 100% accuracy. With support levels of .2% and .1%, the accuracy is still above 94%. The accuracy of Stream-e.75 is always above 99% with these support levels.

When the support level is .1%, `Stream-e1` is always significantly faster than apriori. `Stream-e.75` is also faster than apriori, but the difference is less significant. `Stream+` is actually slower than apriori. With the support level of .1%, `Stream-e1` also always requires less memory than apriori. When the threshold is .4%, apriori is faster than all of our versions.

As stated earlier, besides providing reasonably accurate results in one pass, the key benefit of our algorithm is its ability to handle very large number of distinct items and/or very low support levels. To demonstrate this, we first used the T25.I4.N100K dataset, which has 100,000 distinct items. The number of transactions was 12 million. Note that the size of each transaction is also quite large. Even in this case, the accuracy from `Stream-e1` is above 99.5% and the accuracy from `Stream-e.75` is above 99.8%. The execution times and memory requirements from this dataset are shown in Figures 13 and 14, respectively. With support levels below .4%, all of our versions are significantly faster than apriori. With support levels of .1% and .05%, the memory requirements are also drastically lower than those of apriori.

Next, we focus on the case when support levels are very low. The dataset we use is T10.I4.N10K. We consider support levels of .05% and .08%. The accuracy achieved is still very good. `Stream-`

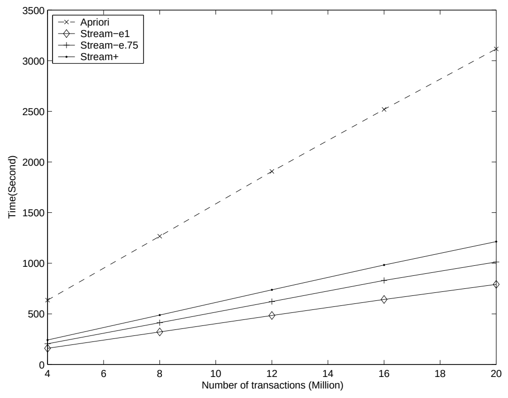

Figure 15: Execution Time with Increasing Dataset Size (threshold=0.08%, T10.I4.N10K Dataset)

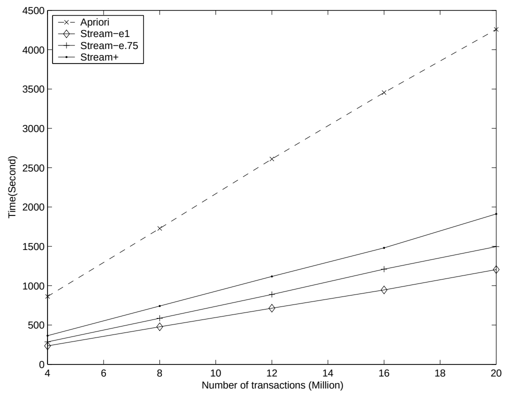

Figure 16: Execution Time with Increasing Dataset Size (threshold=0.05%, T10.I4.N10K Dataset)

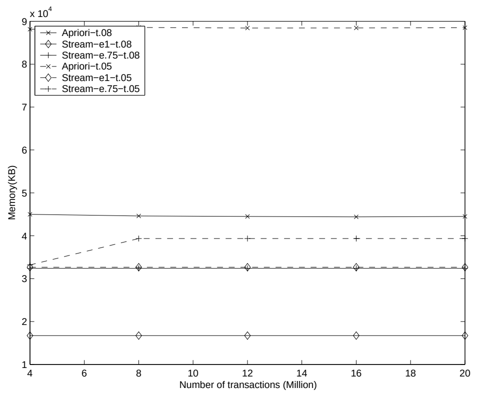

Figure 17: Memory Requirements with Increasing Dataset Size (T10.I4.N10K Dataset)

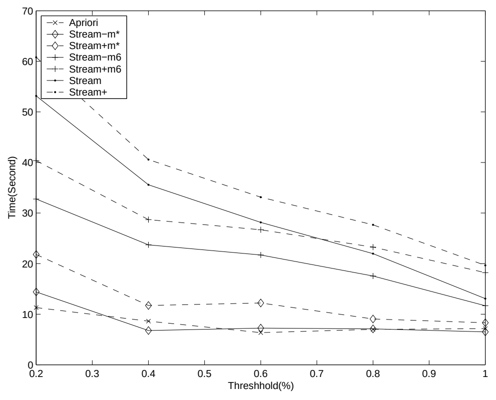

Figure 18: Execution Time with Changing Support Level (BMS-WebView-1 Dataset)

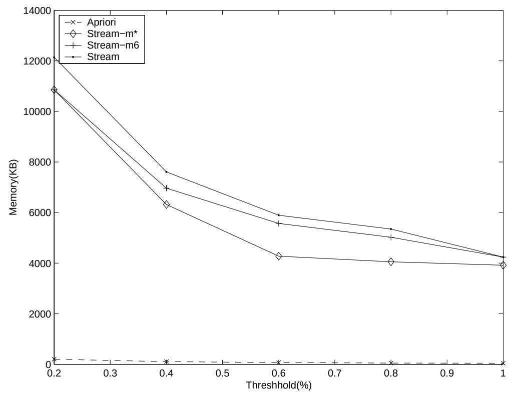

Figure 19: Memory Requirements with Changing Support Level (BMS-WebView-1 Dataset)

`e1` has an accuracy of 97% or better, and `Stream-e.75` has an accuracy of 99.8% or better. The execution times are presented in Figures 15 and 16 and the memory requirements are shown in Figure 17. All of our versions are significantly better both in terms of execution time and memory requirements.

5.2 Real Dataset

The real dataset we use is the BMS-WebView-1 dataset which contains several months of clickstream data from one e-commerce website. A portion of it has been used in the KDD-Cup 2000 competition and also used by Zhang *et al.* [24] to evaluate tranditional offline association mining algorithms.

The characteristics of the BMS-WebView-1 dataset are quite different from the IBM Quest synthetic datasets. The original dataset has 59,602 transactions and contains 497 distinct items. The maximum transaction size is 267, while the average transaction size is just 2.5. For our experiments, we duplicated and randomized the original dataset to obtain 1 million transactions.

Because of the small size of the dataset and the small number of distinct items, we did not expect to outperform apriori on this dataset. However, we have still compared the performance with apriori to show that the algorithm can give accurate results in one pass, and can still be competitive.

In our experiments, we use $\epsilon = 0.6$. Further, we provide another parameter m to represent the maximal frequent itemsets we are interested in. This is because if we have some additional knowledge about the length of the maximal frequent itemsets, the performance of our implementation can be improved. In this dataset, as the support level is 0.2%, 0.4%, 0.6%, 0.8% or 1%, the maximal frequent itemsets is 2, 3, 3, 4, and 6, repectively. For the online checking optimization we had described earlier, the threshhold we define is 10, i.e, two transactions in the buffer will not have a common subset which contains more than 10 items. Since we have only less than 500 distinct items, we maintain all of the 2-itemsets as an array in the main memory.

Figure 18 compares the execution time. `Stream-m*` refers to *StreamMining* with some knowedge of maximal frequent itemsets. For support level of 0.2%, we had $m = 4$, and for others, we had $m = 3$. `Stream-m6` refers to the version using $m = 6$ in all cases. `Stream` refers to *StreamMining* having no knowledge about the maximal frequent itemsets. `Stream+m*`, `Stream+m6` and `Stream+` refer to the corresponding two pass versions.

The three versions have very similar results for accuracy. For threshhold levels between 1% and 0.4%, they achieve 100% accuracy. For the threshold of 0.2%, the accuracy is nearly 99%.

We can see that the performance of `Stream-m*` is quite similar to apriori. For the `Stream-m6` and `Stream`, we can see as the additional information on maximal frequent itemsets is reduced, the algorithm peformance becomes less competitive. For the two-pass algorithm, we can see that the second pass just adds a faily small and constant time.

Figure 19 compares the memory cost of apriori and *StreamMining*. Because the number of frequent itemset is relatively small, the memory cost of apriori is very low. Although the cost of `Stream-Mining` is almost two orders higher than that of apriori, we can see the absolute memory cost is just 11MB. It comes mostly from the initial hash table and the 2-itemset array.

6. RELATED WORK

As stated through-out, our work has two implications. First, we have presented a one pass algorithm for approximate frequent itemset mining on streaming data. Second, we have presented a more memory efficient algorithm for two pass accurate frequent itemset mining. In this section, we compare our work with related research efforts in each of the areas.

Processing of streaming data has received a lot of attention within the last couple of years [3, 5, 8, 10]. Within the area of data mining, significant work has been done on the problem of classification [6, 16] and clustering [12]. More recently, attention has been paid to the area of frequent itemset mining [9, 18].

The work closest to our work on handling streaming data is by Manku and Motwani [18]. They have also presented a one pass algorithm that does not allow false negatives, and has a provable bound on false positives. They achieve this through a very different approach, called *lossy counting*. The differences in the two approaches are in space requirements. For finding frequent items, the approach we use takes $O(1/\theta)$ space. Their approach requires $O((1/\theta)log(\theta N))$ space, where θ is the desired support level and N is the length of the stream. Therefore, for frequent itemset mining, they require an out-of-core data structure. In comparison, we do not need any such structure. On the T10.I4.N10K dataset used in their paper as well, we see that with 1 million transactions and a support level of 1%, their algorithm requires an out-of-core data-structures on top of even a 44 MB buffer. For datasets ranging from 4 million to 20 million transactions, our algorithm only requires 2.5 MB main memory based summary. In addition, we believe that there a number of advantages of an algorithm that does not require an out-of-core summary structure. Mining on streaming data may often be performed in mobile, hand-held, or sensor devices, where processors do not have attached disks. It is also well known that additional disk activity increases the power requirements, and battery life is an important issue in mobile, hand-held, or sensor devices. Also, while their algorithm is shown to be currently computation-bound, the disparity between processor speeds and disk speeds continues to grow rapidly. Thus, we can expect a clear advantage from an algorithm that does not require frequent disk accesses.

Recently, Giannella *et al.* have developed a technique for dynamically updating frequent patterns on streaming data [9]. They create a variation of FP-tree, called FP-stream, for time-sensitive mining of frequent patterns. Because this approach gives additional weightage to recent transactions, it can efficiently answer time-sensitive queries, which we do not consider. However, for queries involving queries on an entire data stream, their approach is not efficient.

As our experimental results have shown, the memory requirements of our approach are significantly lower than those of FP-tree. However, we have not considered time-sensitive queries.

Our work directly builds on top of the recent work by Karp *et al.* on memory efficient frequent items analysis [17]. Our contributions are in extending the work to frequent itemset mining, establishing a bound on false positives, developing data-structures for efficient implementation, and a detailed evaluation and comparison with other frequent itemset mining algorithms.

Now, we compare our work with accurate frequent itemset mining algorithm, which require two or more passes. The classical work in this area is the Apriori algorithm [2, 1]. The basic idea in this algorithm has been extended by several others [22, 19]. Our experimental comparison has shown advantages of our approach when the number of distinct itemsets is large and/or the support level desired is very low. Several algorithms since then have required only two passes. This includes the FP-tree based approach by Han and co-workers [14]. Again, as our experimental results have shown, the memory requirements for maintaining the frequent patterns summary increase rapidly when the support levels are low. Other two pass algorithms for association mining include those from Savarese *et al.* [20] and Toivonen [21]. In each of these cases, the two pass algorithm does not extend to a one pass algorithm with any guarantees on accuracy. Hidber has developed a technique which guarantees that the results after the first pass do not include any false negatives, but produces a large number of false positives [15]. A detailed comparison of frequent itemset mining algorithms has been done by Zheng *et al.* [24].

7. CONCLUSIONS

In this paper, we have developed a new approach for frequent itemset mining. We have developed a new one pass algorithm for streaming environment, which has deterministic bounds on the accuracy. Particularly, it is the first such algorithm which does not require any out-of-core memory structure and is very memory efficient in practice. We have developed a new data structure and several other optimizations to support this algorithm.

Our detailed experimental evaluation has shown the following. First, our one pass algorithm is very accurate in practice. Though a tighter theoretical bound on accuracy can be achieved by increasing memory requirements, it was not really required in practice. Second, the memory efficiency of our one and two pass algorithms allowed us to deal with large number of distinct items and/or very low support levels. For other cases, where traditional multi-pass approaches have worked well in the past, our algorithms are still quite competitive. One exception is datasets with the average length of an itemset is quite large. In such case, some additional knowledge of maximal frequent itemsets helps efficiency of our algorithms.

8. REFERENCES

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonent, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. conf. Very Large DataBases (VLDB'94)*, pages 487–499, Santiago,Chile, September 1994.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 2002 ACM Symposium on Principles of Database Systems (PODS 2002) (Invited Paper)*. ACM Press, June 2002.

[4] Christan Borgelt. Apriori implementation. http://fuzzy.cs.Uni-Magdeburg.de/ borgelt/Software. Version 4.08.

[5] A. Dobra, J. Gehrke, M. Garofalakis, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[6] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the ACM Conference on Knowledge and Data Discovery (SIGKDD)*, 2000.

[7] Mohammad El-Haji and Osmar R. Zaiane. Inverted Matrix: Efficient Discovery of Frequent Items in Large Datasets in the Context of Interactive Mining. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM Press, 2003.

[8] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24. acmpress, June 2001.

[9] C. Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and P. S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In *Proceedings of the NSF Workshop on Next Generation Data Mining*, November 2002.

[10] Phillip B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proc. of the 2001 ACM Symp. on Parallel Algorithms and Architectures*, pages 281–291. ACM Press, August 2001.

[11] Bart Goethals. Fp-tree implementation. http://www.cs.helsinki.fi/u/goethals/software/index.html. Version Last Updated April 2003.

[12] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams. In *Proceedings of 2000 Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 359–366. ACM Press, 2000.

[13] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. *IEEE Transactions on Data and Knowledge Engineering*, 12(3), May / June 2000.

[14] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2000.

[15] C. Hidber. Online Association Rule Mining. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 145–156. ACM Press, 1999.

[16] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the ACM Conference on Knowledge and Data Discovery (SIGKDD)*, 2001.

[17] Richard M. Karp, Christos H. Papadimitrious, and Scott Shanker. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. Available from http://www.cs.berkeley.edu/ christos/iceberg.ps, 2002.

[18] G. S. Manku and R. Motwani. Approximate Frequency Counts Over Data Streams. In *Proceedings of Conference on Very Large DataBases (VLDB)*, pages 346 – 357, 2002.

[19] J. S. Park, M. Chen, and P. S. Yu. An effecitive hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[20] A. Savasere, E. Omiecinski, and S.Navathe. An efficient algorithm for mining association rules in large databases. In *21th VLDB Conf.*, 1995.

[21] H. Toivonen. Sampling large databases for association rules. In *Proc. of the 22nd VLDM Conference.*, 1996.

[22] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining.*, August 1997.

[23] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14 – 25, 1999.

[24] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 401–406. ACM Press, August 2001.