An Efficient Approach to Mining Frequent Itemsets on Data Streams

Sara Ansari, and Mohammad Hadi Sadreddini

***Abstract*—**The increasing importance of data stream arising in a wide range of advanced applications has led to the extensive study of mining frequent patterns. Mining data streams poses many new challenges amongst which are the one-scan nature, the unbounded memory requirement and the high arrival rate of data streams. In this paper, we propose a new approach for mining itemsets on data stream. Our approach SFIDS has been developed based on FIDS algorithm. The main attempts were to keep some advantages of the previous approach and resolve some of its drawbacks, and consequently to improve run time and memory consumption. Our approach has the following advantages: using a data structure similar to lattice for keeping frequent itemsets, separating regions from each other with deleting common nodes that results in a decrease in search space, memory consumption and run time; and Finally, considering CPU constraint, with increasing arrival rate of data that result in overloading system, SFIDS automatically detect this situation and discard some of unprocessing data. We guarantee that error of results is bounded to user pre-specified threshold, based on a probability technique. Final results show that SFIDS algorithm could attain about 50% run time improvement than FIDS approach.

***Keywords*—**Data stream, frequent itemset, stream mining.

I. Introduction

THE increasing importance of data stream arising in a wide range of advanced applications has led to the extensive study of mining frequent patterns. Mining data streams poses many new challenges amongst which are the one-scan nature, the unbounded memory requirement and the high arrival rate of data streams. Mining frequent sets over data streams presents interesting new challenges over traditional mining in static databases. Due to the speed of new arriving data, it is assumed that the history of the stream cannot be revisited, unless it is stored. Storing large parts of a stream, however, is impossible as the amount of data is typically huge. Most previous work on mining frequently occurring itemsets over data streams either focuses on (1) the sliding window model (2) the time-fading model or (3) the landmark model. Each of these models requires a fixed window length or decay factor, given by the user. In many applications however choosing such parameters that are most appropriate for every itemset at every time point in an evolving stream is almost impossible. For example, consider a large retail chain of which sales can be considered as a stream. Then, in order to find frequent sets to do market basket analysis, it is very difficult to choose in which period of the collected data you are interested. For many products, the amount of them sold depends highly on the period of the year. In summer time, e.g., sales of ice cream increase and during the soccer world cup, sales of beer increase. Such seasonal behavior of a specific item or combination of items can only be discovered when choosing the correct window size for that item. This size, however, can hide a similar behavior of other items in another window. The approach (SFIDS) has been developed based on FIDS algorithm. The main attempts were to keep some advantages of the previous approach and resolve some of its drawbacks, and consequently to improve run time and memory consumption. This approach has the following advantages: using a data structure similar to lattice for keeping frequent itemsets, separating regions from each other with deleting common nodes that results in a decrease in search space, memory consumption and run time; and Finally, considering CPU constraint, with increasing arrival rate of data that result in overloading system, SFIDS automatically detect this situation and discard some of unprocessing data. We guarantee that error of results is bounded to user pre-specified threshold, based on a probability technique.

The organization of the paper is as follows. In Section 2, the related works is defined and the central problem statement is formally introduced. Section 3 gives preliminaries for main theorem, on which the SFIDS algorithm in Section 4 is based. Experimental results in Section 5 show that the memory requirements and execution time for the algorithm are extremely small for many real-life data applications and the relation between our measure and existing related work is explored, and Section 6 concludes the paper.

II. Related works

Frequent-pattern mining has been studied extensively in data mining, with many algorithms Proposed and implemented for example, Apriori[17], FP-growth [18], CLOSET [19], and CHARM [20]. Frequent pattern mining and its associated methods have been popularly used in association rule mining [17], sequential pattern mining [21], structured pattern mining [22], iceberg cube computation [23], cube gradient analysis [24], associative classification [25], frequent pattern-based clustering [26], and so on.

Sara Ansari is with the Islamic Azad University of Nourabad Mamasani (corresponding author to provide phone: 917-722-0314; fax: 722-423-2311; e-mail: sansari62@ gmail.com).

Mohammad Hadi Sadreddini is now with the Department of Computer Science, Shiraz University (e-mail: sadreddin@shirazu.ac.ir).

Recent emerging applications, such as network traffic analysis, Web click stream mining, power consumption measurement, sensor network data analysis, and dynamic tracing of stock fluctuation, call for study of a new kind of data, called *stream data*, where data takes the form of continuous, potentially infinite data streams, as opposed to finite, statically stored data sets. Stream data management systems and continuous stream query processors are under popular investigation and development. Due to the large volume of data, data streams can hardly be stored in main memory for on-line processing. A crucial issue in data streaming that has recently attracted significant attention is thus to maintain the most frequent items encountered ([29], [27]). Furthermore, since data-streams are continuous, high speed and unbounded, it is impossible to mine frequent itemsets by using algorithms that require multiple scans. As a consequence new approaches were proposed to maintain itemsets rather than items [1], [2], [7], [32]; [28]; [3]; [30]; [31]; [33]).

In our design, we actively maintain frequent patterns under a tilted-time window framework in order to answer time-sensitive queries. The frequent patterns are compressed and stored using a structure similar to lattice [1], and updated incrementally with incoming transactions. Our time-sensitive stream mining model, SFIDS, includes two major components: (1) SFKI (structure for keeping itemset), and (2) tilted-time window.

III. Preliminaries

Let $I = \{i_1, i_2 \ldots i_m\}$ be a set of literals, called items. Let database *DB* be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Associated with each transaction is a unique identifier, called its *TID*. A set $X \subseteq I$ is also called an *itemset*, where items within are kept in lexicographic order. A k-itemset is represented by $(x_1, x_2 \ldots, x_k)$ where $x_1 < x_2 <\ldots < x_n$. The support of an itemset X, denoted *support(X)* is the number of transactions in which that itemset occurs as a subset. An itemset is called a frequent itemset if $support(X) \geq \sigma \times |DB|$ where $\sigma \in (0, 1)$ is a user-specified minimum support threshold and $|DB|$ stands for the size of the database. The problem of mining frequent itemsets is to mine all itemsets whose support is greater or equal than $\sigma \times |DB|$ in *DB*. The previous definitions consider that the database is static. Let us now assume that data arrives sequentially in the form of continuous rapid streams. Let *data stream* $DS = B_{at}^{bt}, B_{at+1}^{bt+1} \ldots B_{an}^{bn}$ be an infinite sequence of batches, where each batch is associated with a time period $[a_k, b_k]$, let B_{an}^{bn} be the most recent batch. Each batch B_{ak}^{bk} consists of a set of transactions; that is, each batch $B_{ak}^{bk} = [T_1, T_2, T_3 \ldots T_k]$. We also assume that batches do not have necessarily the same size. Hence, the length (L) of the data stream is defined as $L = |B_{at}^{bt}| + |B_{at+1}^{bt+1}| + \cdots + |B_{an}^{bn}|$ Where $|B_{at}^{bt}|$ stands for the cardinality of the set B_{at}^{bt} .The support of an itemset X at a specific time interval $[a_i, b_i]$ is now denoted by the ratio of the number of customers having X in the current time window to the total number of customers.

TABLE II
Updating Items After Processing All Batches in FIDS

Items	Tilted-TW	{Region,Root}
1	$\{[T_0^1=1],[T_1^1=1],[T_2^1=2]\}$	$\{[T_0^1=1],[T_1^1=2]\}$
2	$\{[T_0^1=1],[T_1^1=1],[T_2^1=2]\}$	$\{[T_0^1=1],[T_1^1=2]\}$
3	$\{[T_0^1=1],[T_1^1=2]\}$	$\{(1,T_a)\}$
8	$\{[T_0^1=1],[T_1^1=2]\}$	$\{(2,T_a)\}$
9	$\{[T_0^1=1],[T_1^1=2]\}$	$\{(2,T_a)\}$

Therefore, given a user-defined minimum support, the problem of mining itemsets on a data stream is thus to find all frequent itemsets X over an arbitrary time period $[ai, bi]$, i.e. verifying:

$$\sum_{t=at}^{bt} support(X) \geq \sigma \times |B_{at}^{bt}|$$

Of the streaming data using as little main memory as possible.

IV. SFIDS Algorithm

As we mentioned, SFIDS developed based on FIDS algorithm [1], FIDS with use of region concept and lattice data structure, first process transactions of each batch, each itemset in transaction compare with root of each available regions and compute common subsets then insert this subset in both of regions, in following example we illustrate how FIDS algorithm process each incoming batch, consider Table 1, We will now focus on how each new batch is processed. From the batches from Table I our algorithm performs as follows: we process the first transaction T_a in TableI by first storing T_a into our lattice (*Latticereg*). This lattice has the following characteristics: each path in $Lattice_{reg}$ is provided with a *region* and itemsets in a path are ordered according to the inclusion property. By construction, all subsets of an itemset are in the same region. This lattice is used in order to reduce the search space when comparing and pruning itemsets. When the processing of T_a completes, we are provided with a set of items {1, 2, 3, 4, 5} and $Lattice_{reg}$ updated. Items are stored in ItemTable as illustrated in TableII. The *Tilted-T W* attribute is the number of occurrences of the corresponding item in the batch. The $Root_{reg}$ attribute stands for the root of the corresponding region in $Lattice_{reg}$. Of course, for one region we only have one $Root_{reg}$ and we also can have several regions for one item. For itemsets, we store both the size of the itemset and the associated tilted-time window Table IV.

Let us now process the second transaction T_b of B_0^1. Since T_b is not included in T_a, it is inserted in *Latticereg* in a new region. Let us now consider the batch B_1^2 merely reduced to *Tc*. Since items 1 and 2 already exist in the set of itemsets, their tilted-time windows must be updated. (Table II, Table IV)

Furthermore, items 1 and 2 are in the same region: 1 and the longest itemset for these items is (1 2 3 4 5), i.e. T_c is

included in T_a. We thus have to insert T_c in $Lattice_{reg}$ in the region 1. Nevertheless as T_c is a subset of T_a that means that when T_a occurs in previous batch it also occurs for T_c. So the tilted-time windows of T_c must also be updated. The transaction T_d is considered in the same way as T_c. for transaction T_e; items 1 and 2 are in region 1 while items 8 and 9 are in region 2; it means that we are provided with a new region. Nevertheless, we can notice that the itemset (8 9) already exist in $Lattice_{reg}$ and is a subset of T_e. The longest itemset of T_e in the region 1 is {1, 2}. In the same way, the longest subset of T_e for region 2 is {8, 9}. As we are provided with two different regions and {8, 9} is the root of the region 2, we do not create a new region but we insert T_e as a root of region for 2 and we insert the subset {1, 2} both on lattice for region 1 and 2. (Fig. 1)

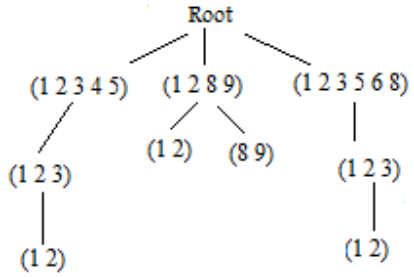

Fig. 1The region lattice after processing all batches (FIDS)

As we seen itemset (1 2) are common in two regions 1 and 2, also in Fig. 2 each item belong to more than one regions i.e. item 1exist in regions 1 and 2, In SFIDS algorithm we keep for each item in ITEMS only first region that item insert into it regardless of other regions it belong to. Let us process previous example with new algorithm, when the processing of T_a completes, we are provided with a set of items {1, 2, 3, 4, 5} and SFKI updated. Items are stored illustrated in (Fig. 2). Let us now process the second transaction T_b of B_0^1. Since T_b is not included in T_a, it is inserted in SFKI in a new region. Sub tree (8 9). Let us now consider the batch B_1^2 merely reduced to T_c. Since items 1 and 2 already exist in the set of itemsets, their tilted-time windows must be updated (Table III). Now we want to process B_2^3, first transaction T_d, since (1 2 3) are subset of root of region 1, insert it in region 1, and update TTW, for transaction T_e, check for each item of it to see regions of each item, since we consider only one region for each item, retrieve their regions, for itemset (1 2 8 9) we compute common subset with (1 2 3 4 5) as root of region 1, since(1 2) already exist in structure we only update occurrence of it, then compute common subset with region 2, (8 9) also exist in region 2 ,we only update occurrence of it, since (1 2 8 9) is superset (8 9) ,we insert it in this region Fig. 2 but don't insert(1 2) in this region, since it exist in structure in region 1, now suppose new batch coming, and include T_f (1 2 3 5 6 8),we refer to item table and see region 1 for items 1 and 2 and 5,and region 2 for item 8 , since itemsets(1 2 3) now exist in structure we only update TTW(Table IV), and insert T_f in new region 3(Fig. 2).

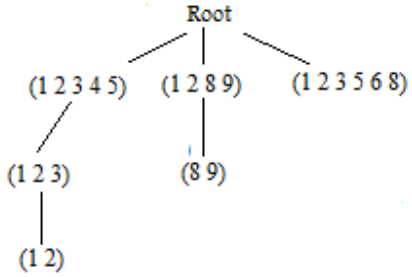

Fig. 2 SKFI after processing all batches

To only store frequent maximal itemsets, let us now discuss how infrequent itemsets are pruned. While pruning in [3] is done in two distinct operations, our algorithm prunes infrequent itemsets in a single operation which is in fact a dropping of the tail itemsets of tilted-time windows support $_{k}^{k+1}$, support $_{k+1}^{k+2}$ … support $_{n-1}^{n}$ when the following condition holds:

$$\forall i, k \le i \le n, Support\ {}_{a_i}^{b_i}(\mathrm{X}) \prec \varepsilon_f \left| B_{a_i}^{b_i} \right| \qquad (1)$$

By navigating into $Lattice_{reg}$, and by using the region indexes, we can directly and rapidly prune irrelevant itemsets without further computations. This process is repeated after each new batch in order to use as little main memory as possible. During the pruning phase, titled-time windows are merged in the same way as in [3].

A. An Efficient Representation for Itemsets

According to the overview, one crucial problem is to efficiently compute the inclusion between two itemsets. This costly operation could easily be performed when considering a new representation for items in transactions. From now, each item is represented by a unique prime number. According to this definition, each transaction could be represented by the product of the corresponding prime numbers of individual items into the transaction. As the product of the prime number is unique we can easily check the inclusion of two itemsets by performing a modulo division on itemsets (Y MOD X). If the remainder is 0 then $X \subseteq Y$, otherwise X is not included in Y.

B. Workload Estimation

The second of SFIDS considering CPU constraint[34], with increasing arrival rate of data that result in overloading system, SFIDS automatically detect this situation and discard some of unprocessing data, in [15] with use of Probabilistic technique could reach to interesting results, this technique is used in SFIDS to shed load. Since the behavior of data streams often changes over time, detecting overload situations is an important step in our algorithm. For the frequent set mining problem, the system workload may not be simply estimated by regularly checking the rate of transactions arriving in one time unit. Rather, it is essentially dependent on the number of itemsets containing in each transaction whose frequencies must be updated. Certainly, an accurate method to evaluate the system workload is to have an exact count of this number In each transaction. Unfortunately, this task is generally impossible since the system may not be able to fully process all incoming transactions under overload situations.

Therefore, it is necessary to find a technique that is able to approximately estimate this number meanwhile it is efficient to compute. We pro pose such an estimate based on a small set of maximal

Frequent itemsets(MFIs). The key intuition behind using MFIs for this task is that the number of MFIs is exponentially smaller than the number of frequent itemsets. Mean while, such a compact set completely captures the entire set of frequent itemsets. To give an explicit formula for this estimate, let us denote the number of MFIs in a transaction by m and let Xi be a MFI, $1 \le i \le m$. We assume the following estimated time (or load coefficient) to process one transaction:

$$L = \sum_{i=1}^{m} 2^{|x_i|} - \sum_{i,j=1}^{m} 2^{|x_i \cap x_j|} \qquad (2)$$

In this equation, the first summation estimates the number of frequent itemsets within each MFI. The second one estimates the number of itemsets overlapping between MFIs. Apparently, one may suppose that computing L is expensive and eventually defeats the purpose of quickly detecting CPU overload. Nevertheless, we do not explicitly compute L by finding all MFIs and the overlapping subsets among them.
Instead, the set of MFIs is maintained in a prefix tree and the equation is computed by matching transactions to this structure to derive the number of distinct frequent sets. As we shall see from the empirical results in Section 4, this approach yields very good approximation while the computational time is negligible. Given this computed statistics for each transaction, we measure it for n transactions over one time unit and let r be the current rate of the data stream. The following inequality imposes a constraint on load shedding decisions:

$$P \times r \times \frac{\sum_{i=1}^{n} L_i}{n} \le C \qquad (3)$$

In this inequality, $P \in$ (0; 1] is a parameter adjusted adaptively to make the inequality hold. It also expresses the fraction of transactions that should be discarded. The expression $r \times \frac{\sum_{i=1}^{n} L_i}{n}$ gives the estimated system workload to process transactions arriving in one time unit. L_i is described in Equation 1; C, as formulated above, is the processing capacity of the mining system.

C. Probabilistic Technique to Shed Load

As we have analyzed above, when the system is overloaded, an immediate approach is to drop a fraction of the stream to reduce workload. Certainly, when all incoming data is not entirely processed (and dropped transactions are lost forever), one can expect some errors in the results. Our algorithm is designed to approximate this error in a precise manner. In other words, the error is guaranteed within some specific lue. To achieve this, we approach the problem by utilizing a technique from probability, the Chernoff bound [16]. Such a theoretically sound tool allows us to obtain a more accurate estimate on the mining error. To apply the Chernoff bound in our frequent set approximation, we clarify

TABLE III
UPDATING ITEMS AFTER PROCESSING ALL BATCHES IN SFIDS

Items	Tilted-TW	{region, root}
1	$\{[T_0^1=1],[T_1^0=1],[T_2^1=2],[T_3^2=1]\}$	{1,T_a}
2	$\{[T_0^1=1],[T_1^0=1],[T_2^1=2],[T_3^2=1]\}$	{1,T_a}
3	$\{[T_0^1=1],[T_2^1=2],[T_3^2=1]\}$	{1,T_a}
5	$\{[T_3^1=1]\}$	{3,T_f}
6	$\{[T_3^1=1]\}$	{3,T_f}

the following concepts. Let P be a value smaller than 1, then each coming transaction is chosen with probability P. For a set of N transactions arriving in the stream, n transactions are chosen randomly. Given an itemset X, we want to approximate how

Close is its computed frequency in n sampling transactions, as compared to its actual frequency p in N transactions. Note that event X appearing in a transaction can be seen as a Bernoulli trial and thus, we denote random variable $Ai = 1$ if X appears in the ith transaction and $Ai = 0$ otherwise. Obviously, Pr ($Ai = 1$) = p and Pr ($Ai = 0$) = 1- p. Hence, n randomly drawn transactions are viewed as n independent Bernoulli trials. Let r be the number of times that $Ai = 1$ occurs in these n transactions; r is called a *binomial* random variable and thus, its expectation is np. Then, the Chernoff bound states that given an error bound ε; $0 < \varepsilon < 1$:

TABLE IV
UPDATING ITEMSETS AFTER PROCESSING ALL BATCHES

Itemsets	Tilted-TW	Size
(1 2 3 4 5)	$[T_6^1=1]$	5
(8 9)	$\{[T_0^1=1],[T_2^1=2]\}$	2
(1 2)	$\{[T_0^1=1],[T_1^0=1],[T_2^1=2]\}$	2
(1 2 3)	$[T_2^1=1]$	3

$$\Pr\left\{|r - np| \ge np\varepsilon\right\} \le 2e^{-np\varepsilon^2/2} \qquad (4)$$

Let us call $supp_E(X) = r/n$ the estimated support of X computed from n sampling transactions, then this equation gives us the probability that the true support $supp_E(X)$ deviates from its estimated support $supp_E(X)$ by an amount ±ε. If we want this probability to be no more than δ, then the required number of sampling transactions is at least by setting

σ= $2e^{-np\varepsilon^2/2}$

$$n_0 = 2p\ln(2/\delta)/\varepsilon^2 \qquad (5)$$

For example, consider p = 0.002, δ = 0.05 and ε = 0.001, then $n_0 \approx 15\ 000$. This means that for itemset *X*, if we sample 15,000 transactions from a partition of the stream, its true support $supp_E(X)$ in this partition is beyond the range of $[supp_E(X) - 0.001; supp_E(X) + 0.001]$ with probability 0.05. In other words, $supp_T(X)$ is within ±ε of $supp_E(X)$ with a high confidence of 95%.

D. Details about SFIDS

We describe in more detail the Fids algorithm (c.f. Algorithm SFIDS). While batches are available, we consider itemsets embedded into batches in order to update our structures (Update). Then we prune unfrequent itemsets in order to maintain our structures in main memory (Prune). In the following, we consider that we are provided with the three next structures. Each value of *ITEMS* is a tuple (*labelitem*,{*time, occ*}, {(*regions*, *rootreg*)}) where *labelitem* stands for the considered item, {*time, occ*} is used in order to store the number of occurrences of the item for different time of batches and for each region in {*regions*} we store its associated itemsets (*rootreg*) in the *Latticereg* structure. The *ISETS* structure is used to store itemsets. Each value of *ISETS* is a tuple (*itemset*, *size(itemset)*, {*time, occ*}) where *size(itemset)* stands for the number of items embedded in *s*. Finally, the *Latticereg* structure is a lattice where each node is an itemset stored in *ISETS* and where vertices correspond to the associated region (according to the previous overview). Let us now examine the Update algorithm (c.f. Update method) which is the main core of our approach. We consider each transaction embedded in the batch. The system workload is periodically estimated to identify overloading. If such a situation occurs, the maximal value of P is computed via equation 3. Otherwise, P is set to 1. If transaction T is chosen in this phase, first get regions of all its items is compted from GetRegions method. If items were not already considered we only have to insert *T* in a new region. Otherwise, we extract all different regions associated on items of *T* (one region for each item;it's different from FIDS approach). For each region, the GetRoot*reg* function returns the corresponding root of the region, *FirstItemset*, i.e. the maximal itemset of the region *reg*. Since we represent items by prime numbers, we can then compute the greatest common factor of *T* in *FirstItemset* by applying the GCF function. This usual function was extended in order to return an empty set both when there are no maximal itemsets or if itemsets are merely reduced to one item. If there is only one itemset, i.e. cardinality of NewIts is 1; we know that the itemset is either a root of region or *T* itself. We thus store it into a temporary array (*LatticeMerge*) in order to avoid creating a new useless region. Otherwise we know that we are provided with a subset and then we insert it into *Latticereg* (Insert) and propagate the tilted-time window (UpdateTTW). Itemsets
are also stored into a temporary array (*DelayedInsert*). If there exist more than one sub itemset (from GCF), then we insert all these subsets on the corresponding region. We also store them with *T* on *DelayedInsert* in order to delay their insertion as a New region. If *LatticeMerge* is empty we know that it does not exist any subset of *T* already included on itemsets of *Latticereg* and then we can directly insert *T* into *Latticereg* with a new region. If the cardinality of *LatticeMerge* is greater than one, we are provided with an itemset which will be a new root of region and then we insert it.

SFIDS ALGORITHM

Batch

Data: an infinite set of batches B= B_0^1 , B_1^2 … B_{n-1}^n ; a user defined threshold σ; an error rate ε.
Result: A set of frequent items and itemsets
//initialize phase
$Lattice_{reg}$=0; ITEMS=0; ISETS=0;
Repeat
  For each $B_a^b \in$ B do
  Update (B_a^b , $Lattice_{reg}$, ITEMS, ISETS, ,);
  Prune ($Lattice_{reg}$, ITEMS, ISETS,,);
Until no more batch;

SFIDS ALGORITHM UPDATE METHOD

Data: a batch B_a^b : a user- defined thresholdσ; an error rate ε , three structures and a processing capacity C of the mining system.
Result: Latticereg, ITEMS, ISETS updated.
Periodically identify sampling rate P;
For each transaction T∈ B_a^b , sampling it with probability of P and if T is chosen
  LatticeMerge=0; Candidates=0;
  If (candidates==0) then
    Insert (T, New Region);
  Else
    For each region ∈ Candidates do
      FirstItemset= GetRootreg(reg);
      //compute all the longest common subset
      NewIts=GCF (T, FirstItemset);
      If (NewIts==T)|| (NewIts==FirstItemset) then
        LatticeMerge=reg;
      Else
        //A new itemset has to be considerd
      Insert (NewIts, reg);UpdateTTW (NewIts);
If(|LatticeMerge|==0) then
  Insert (T,New Region); UpdateTTW(T);
  Else
  If(| LatticeMerge|==1) then
    Insert (T, LatticeMerge[0]);UpdateTTW(T);
    Else
    Merge (LatticeMerge, T);

Maintaining all the data streams in the main memory requires too much space. So we have to store only relevant itemsets and drop itemsets when the tail-dropping condition holds. When all the tilted-time windows of the itemsets are dropped the entire itemset is dropped from *Latticereg*. As the result of the tail-dropping we no longer have an exact support over *L*, rather an approximate support. Now let us denote $support_L(X)$ the frequency of the itemset *X* in all batches and $support_{L'}(X)$ the approximate frequency. With $\varepsilon << \sigma$ this approximation is assured to be less than the actual frequency according to the following inequalityas in [3]:

$$Supprt_L(X) - \varepsilon|L| \leq Support_{L'}(X) \leq Support_L(X) \quad (6)$$

V. Experimental Result

All experiments were performed on a PC with a CPU 1.8GHz running Windows with 1GB memory. All programs were implemented in VC++. The stream data was generated from dataset *T*10*I*4*D*100*K* is created by IBM synthetic data generator. The stream was broken into batches of 30s duration which enables the possibility for different batch sizes depending on the distribution of the data. The number of items per batch was nearly 5,000. We have fixed the error threshold (ε) at 0.1% and we compared our method with FIDS.

A. Comparison with Algorithm FIDS

In this experiment, we examine the execution time and memory usage between SFIDS and FIDS by dataset *T*10*I*4*D*100*K*. In Fig. 3, we can see that the execution time incurred by SFIDS is quite steady and is shorter than that of FIDS. The experiment shows that SFIDS performs more efficiently than FIDS. In Fig. 4, the memory usage of SFIDS is more stable and smaller than that of FIDS. This is because SFIDS delete a lot of common nodes between various regions does not need to search these spaces for some computations and enumerate all subsets of each incoming transaction. The amount of all subsets is an enormous exponential number for long transaction. Hence, it shows that SFIDS is more suitable for mining frequent itemsets in data streams.

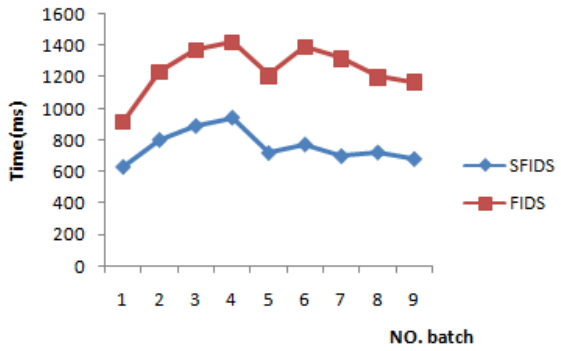

Fig.3 Compare execution time between FIDS and SFIDS

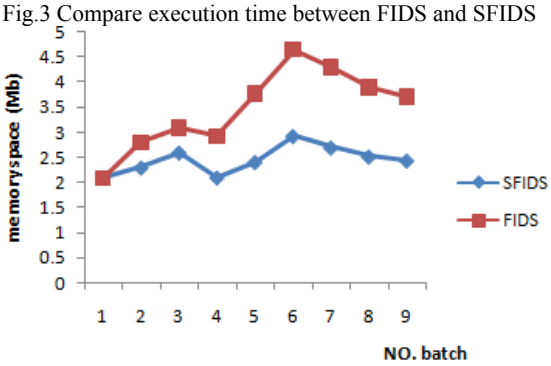

Fig. 4 Compare space consumption between FIDS and SFIDS

VI. Conclusion

In this paper, an SFKI structure was designed to dynamically maintain the up to date contents of an online data stream by scanning it only once, and a new method SFIDS was proposed to mine the frequent patterns in a tilted time window. This method could answer a request with no false negative. Extensive Experimental results show that SFIDS decrease required time for processing batches and amount of memory for storing history of data. We compare our algorithm with FIDS algorithm and show that SFID perform better than FIDS in various conditions.

References

[1] C.Raissi, P. Poncelet, Teisseire, "Towards a new approach for mining frequent itemsets on data stream", J Intell Inf Syst , 2007, vol. 28, pp. 23–36, 2007.

[2] J,.Xu Yu , Z.Chong , H. Lu, Z. Zhang , A. Zhou b," A false negative approach to mining frequent itemsets from high speed transactional data streams", Information Sciences 176, 1986–2015,2006.

[3] G. Giannella, J. Han, J. Pei, X. Yan, P.Yu," Mining frequent patterns in data streams at multiple time granularities", In Next generation data mining. New York: MIT, 2003.

[4] H. Li, F. Lee, S.Y., M. Shan," An efficient algorithm for mining frequent itemsets over the entire history of data streams".,In Proceedings of the 1st InternationalWorkshop on Knowledge Discovery in Data streams,2004.

[5] R. Jin, G. Agrawal," An Algorithm for In-Core Frequent Itemset Mining on Streaming Data"

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

[7] A. Cheng, Y. Ke,W. Ng, "A survey on algorithms for mining frequent itemsets over data streams", Knowl Inf Syst, 2007.

[8] M. Charikar, K. Chen, M. Farach," Finding frequent items in data streams". Theor Comput Sci, vol. 312, pp.3–15, 2004.

[9] J.Cheng,Y. Ke,W. Ng," Maintaining frequent itemsets over high-speed data streams", In Proceedings of the 10th Pacific-asia Conference on knowledge discovery and data mining, Singapore, pp. 462–467, April 2006.

[10] R.Agrawal,T. Imielinski,A. Swami , "Mining association rules between sets of items in large databases", In Proceedings of the ACM SIGMOD international conference on management of data, Washington DC, pp 207–216,1993.

[11] 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, 1953.

[12] M. Charikar, K. Chen, M. Farach," Finding frequent items in data streams", in Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), pp. 693–703,2002.

[13] T. Calders , N. Dexters , B. Goethals ,"Mining Frequent Items in a Stream Using Flexible Windows"

[14] X. Sun Maria, E. Orlowska , X. Li, "Finding Frequent Itemsets in High-Speed Data Streams".

[15] X. Han Dong, W. Ng, K. Wong,V. Lee, "Discovering Frequent Sets from Data Streams with CPU Constraint", This paper appeared at the AusDM 2007, Gold Coast, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol.70

[16] H. Chernoff, "A measure of asymptotic efficiency for thests of a hypothesis based on the sum of observations", in Annals of Mathematical Statistics, pp. 493, 1952.

[17] R. Agrawal, R. Srikant, " Fast algorithms for mining association rules", In Proc. Int. Conf. Very Large Data Bases (VLDB'94), 487.499, 1994

[18] J. Han, J. Pei, Y. Yin," Mining frequent patterns without candidate generation", In Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00), 1.12, 2000.

[19] J. Pei, J. Han, R. Mao," CLOSET: An efficient algorithm for mining frequent closed itemsets", In Proc. ACM-SIGMOD Int.Workshop Data Mining and Knowledge Discovery (DMKD'00), 11.20,2000.

[20] M. Zaki, C. Hsiao," CHARM: An efficient algorithm for closed itemset mining", In Proc. SIAM Int. Conf. Data Mining, 457.473, 2002.

[21] R. Agrawal, R. Srikant," Mining sequential patterns", In Proc. Int. Conf. Data Engineering (ICDE'95), 3.14, 1995.

[22] M. Kuramochi, G. Karypis, "Frequent subgraph discovery", In Proc.Int. Conf. Data Mining (ICDM'01), 313.320, 2001.

[23] K. Beyer, R. Ramakrishnan," Bottom-up computation of sparse and iceberg cubes", In Proc. ACM-SIGMOD Int. Conf.Management of Data (SIGMOD'99), 359.370,1999.
[24] T. Imielinski, L. Khachiyan,A. Abdulghani,"Cubegrades: Generalizing association rules", Data Mining and knowledge Discovery 6:219.258,2002.
[25] B. Liu, W. Hsu, Y. Ma," Integrating classification and association rule mining", In Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD'98), 80.86,1998.
[26] H. Wang, J. Yang, W. Wang, P. Yu," Clustering by pattern similarity in large data sets", In Proc. ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'02), 418.427,2002.
[27] R. Karp, C. Papadimitriou, S. Shenker, "A simple algorithm for finding frequent elements in streams and bags", ACM Trans. Database Systems 2003.
[28] Y. Chi, H. Wang, P. Yu, R. Muntz," Moment: Maintaining closed frequent itemsets over a stream sliding window", In Proceedings of International Conference on Data Missing Conference, pp. 59–66, 2004.
[29] C.Jin, W. Qian, C. Sha, J.Yu, A. Zhou," Dynamically maintaining frequent items over a data stream", In Proceedings of International Conference on Information and Knowledge Management Conference, pp. 287–29, Washington, District of Columbia, 2004.
[30] H. Li, S. Lee, M. Shan, "An efficient algorithm for mining frequent itemsets over the entire history of data streams", In Proceedings of the 1st International Workshop on Knowledge Discovery in Data streams, 2003.
[31] G. Manku,R. Motwani, " Approximate frequency counts over data streams. In Proceedings of very Large Databases Conference, pp. 346–357, Hong Kong, China, 2002.
[32] Y. Chen, G. Dong, J. Han, B. Wah, J. Wang, ," Multidimensional regression analysis of time-series data streams" In VLDB Conference,2002.
[33] W. Teng, M. Chen, P. Yu, "A regression-based temporal patterns mining schema for data streams", In Proceedings of very large Databases Conference, pp. 93–104, Berlin, 2003.
[34] X. Hong , W. Keong , K. Leong Ong ,v. C S Lee , "Discovering Frequent Sets from Data Streams with CPU Constraint", Sixth Australasian Data Mining Conference, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 70, 2007.