# An Efficient Sliding Window Based Algorithm for Adaptive Frequent Itemset Mining over Data Streams

MHMOOD DEYPIR[1] AND MOHAMMAD HADI SADREDDINI[2], MEHRAN TARAHOMI[3]
[1,2,3]*Computer Science and Engineering Department, School of Engineering, Shiraz University, Shiraz, Iran*

Mining frequent itemsets over high speed, continuous and infinite data streams is a challenging problem due to changing nature of data and limited memory and processing capacities of computing systems. Sliding window is an interesting model to solve this problem since it does not need the entire history of received transactions and can handle concept change by considering only a limited range of recent transactions. However, previous sliding window algorithms require a large amount of memory and processing time. This paper, introduces a new algorithm based on a prefix tree data structure to find and update frequent itemsets of the window. In order to enhance the performance, instead of a single transaction, a batch of transactions is used as the unit of insertion and deletion within the window. Moreover, by using an effective traversal strategy for the prefix tree and suitable representation for each batch of transactions, both updating of current itemsets and inserting of newly emerged itemsets are performed together, thus improving the performance even further. Additionally, in the proposed algorithm by storing required information in each node of the prefix tree, deleting old batch of transactions from the window as well as pruning infrequent itemsets are efficiently accomplished. Although, with respect to previous algorithms, our algorithm maintains more information in the prefix tree, it does not require storing the set of transactions of the window, thus reducing the memory usage. Empirical evaluations on both real and synthetic datasets show the superiority of the proposed algorithm in terms of runtime and memory requirement. Moreover, it produces mining results with higher quality.

***Keywords:*** Frequent itemset mining, Sliding window, Data stream, Data Mining, Prefix tree

## 1. INTRODUCTION

A data stream represents an input data that arrives at a rapid rate and is probabably infinite. Examples for the sources of data streams include customer click streams, network monitoring data, telephone record calls, large sets of web pages, sensor networks, scientific data, retail chain transactions and etc. Due to massive amount of incoming data in data stream mining, data elements must be scanned only once using a limited amount of main memory during the mining process [1]. Among all functions of data mining, frequent itemset mining [2] over data streams attracts a great deal of interest in data mining community since it can reveal valuable knowledge in various applications. Sliding window is an interesting model for frequent itemset mining over data streams [3-11]. In this model, only recently arrived transactions are considered for the mining purpose. Sliding window model handles concept drift within an input data stream by considering only recent transactions. Therefore, the window always contains recent concept. In frequent pattern mining problem, the concept referes to the set of frequent patterns. There are two different approaches in data stream mining, for handling concept change within a window. First, detecting concept change and recovering from it and, second, adapting the mining result to the new concept as time progresses. The second approach is more efficient and widely used in pattern mining over data streams since it does not require an independent process of change detection.

Another property of sliding window is its limited amount of memory usage and processing power. In a transaction sensitive window, each sliding is performed by inserting a new transaction and deleting the oldest one. In this way, performing one sliding dose not significantly affect the window content and mining results whereas takes a considerable time to update both of them. By enlarging the unit of insertion and deletion to a batch of transactions as well as dividing overlappling windows into disjouint panes, the window sliding process is performed by pane addition and deletion. This deviding, enhances the process of sliding as long as there are enough transactions per pane. This is due to applying the same process on the window content and mining results for a couple of transactions instead of single one. Moreover, we can design a data structure to efficiently store panes of a window.

In this study, a novel pane based algorithm is proposed to adaptively maintain the set of frequent itemset over sliding window. This algorithm not only updates the current set of frequent itemsets but also adapts itself to recent changes of the input data stream. That is, it can identify newly appeared patterns and remove infrequent ones as new tansactions arrive. The main contributions of this paper are summarized as follows:

- A new approximate algorithm for sliding window frequent itemset mining is introduced which is more efficient and more accurate than previously proposed algorithms.
- A novel traversal strategy of the prefix tree is proposed which enhances the process of updating set of frequent itemsets and insertion of newly emerged ones.
- A new approach to remove expired transactions without physically storing transactions of the window is devised.

The rest of the paper is organized as follows. In the next section some previous related works are reviewed. Section 3 states the problem. In Section 4, the proposed algorithm is introduced. Some experimental evaluations of the proposed algorithm and comparison with pervious methods are presented in Section 5. Finally, Section 6 concludes the paper.

## 2. RELATED WORKS

The problem of frequent itemset mining over data stream was introduced by Manku et al [12] where the authors proposed algorithms for frequency counts of items over data streams. Subsequently, they extend their work to a frequent itemsets generation and test approach within data streams. Over the last decade, a number of exact and approximate algorithms have been proposed for mining frequent patterns on data streams. Based on data stream processing model, they can be categorized to three different categories, landmark based [13, 14, 15], damped or time decay based [16, 17] and sliding window based [3-11] algorithms. In the landmark model, incoming data from a point of time called landmark until the current time is considered. The landmark can be starting or restarting time of the system. In this model, previous and recent transactions of the input stream are regarded as being the same. DSM-FI [13] is a landmark based algorithm. In this algorithm every transaction is converted into smaller transactions and inserted into a summary data structure called *item-suffix frequent itemset forest* which is based on prefix-tree. In [14] the authors used the *Chernoff Bound* to produce an approximate result of frequent patterns over the landmark window. Zhi-Jun et al. [15] used a lattice structure, referred to as a *frequent enumerate tree*, which is divided into several equivalent classes of stored patterns with the same transaction-ids in a single class. Frequent patterns are divided into equivalent classes, and only those frequent patterns that represent the two borders of each class are maintained; other frequent pat-

terns are pruned. In time decay model, the importance of data elements varies based on their arrival order to emphasize recently arrived data. Chang and Lee proposed an algorithm called *estDec* based on *time decay* model in which each transaction has a weight decreasing with age [16]. In this method in order to reduce the effect of old transaction in the set of frequent patterns a *decay rate* is defined. In [17] an algorithm similar to the *estDec* is proposed for mining maximal frequent itemsets instantly over data stream based on damped model.

There are a lot of sliding window based algorithms proposed for frequent itemset mining over data streams. Since our proposed algorithm operates in the sliding window model and this model is a widely used model for data stream mining, here significant algorithms proposed in this model are reviewed. *DSTree* [3] and *CPS-Tree* [4] are two algorithms that use the prefix tree to store raw transactions of sliding window. *DSTree* uses a fixed tree structure in canonical order of branches while in *CPS-Tree* the prefix tree structure is reconstructed to control the amount of memory usage. Both of [3] and [4] perform the mining task using *FP-Growth* [18] algorithm that was proposed for static databases. In [5] an algorithm namely *MFI-TransSW* was proposed which is based on the *Apriori* algorithm [2]. This algorithm mines all frequent itemsets over recent window of transactions. It uses a bit string for every item to store its occurrence information within the window. In [6] a sliding window based algorithm has been proposed in which the window content is dynamically maintained using a set of simple lists.

Lin et al. [7] proposed a new method for mining frequent patterns over time sensitive sliding window. In their method, the window is divided into a number of batches for which itemset mining is performed separately. The *Moment* algorithm [8] finds closed frequent itemsets by maintaining a boundary between frequent closed itemset and other itemsets. The *SWIM* [9] is a pane based algorithm in which frequent itemsets in one pane of the window are considered for further analysis to find frequent itemsets in whole of the window. It keeps the union of frequent patterns of all panes and incrementally updates their supports and prunes infrequent ones. It stores transactions of the window in form of the prefix tree of each pane. In [10] the authors devised an algorithm for mining non-derivable frequent itemsets over data streams. This algorithm continuously maintains non-derivable frequent itemsets of the sliding window. Non-derivable and closed frequent itemsets are special types of frequent itemsets which can be viewed as a summary of all frequent itemsets.

Chang and Lee proposed the *estWin* algorithm [11] that finds recent frequent patterns adaptively over transactional data streams using sliding window model. Since our algorithm is closely related to the *estWin* algorithm, it is described in more details. This algorithm uses a monitoring lattice in the form of a prefix tree to monitor the set of frequent itemsets over a data stream. Each node of the lattice represents an itemset which can be constructed using items stored in the nodes in the path from the root to the node. Last node of this path stores the information related to the itemset, e.g., support. The algorithm uses a reduced minimum support named minimum significant to early identify new frequent itemset and to better estimate their support. When a new transaction is arrived from the input data stream, current set of frequent itemset is updated by visiting related itemset in the monitoring lattice. Then new significant itemsets are identified by second traversal of the related paths of the monitoring lattice using the transaction. This algorithm stores the set of transactions of the window in the memory to eliminate their effect when they become expired. For removing oldest transaction from the window, related paths of the monitoring lattice must be visited again. The algorithm, prunes insignificant itemset from the tree after passing a constant number of transactions using an independent process named the force pruning. An itemset is inserted to the prefix tree when all of its subsets become significant and are stored

in the tree. The algorithm estimates the support of a new significant itemset within previous transactions of the window. For an itemset having length *k* (k-itemset), its estimated support is equal to lowest support among all of its subsets with length *k-1*. Each node of the prefix tree contains information including possible count (*pcnt*), actual count (*acnt*), error (*err*) and first transaction (*mid*). *Pcnt* referrers to the estimated frequency of the itemset among the previous transactions of the window before the itemset is inserted to the prefix tree. *Acnt* is the monitored frequency of the itemset after it is inserted to the tree. *Err* is error of the estimated support of the itemset calculated using its subsets. *Mtid* is the *id* of transaction that causes the itemset to be inserted to the prefix tree. The support of an itemset in the prefix tree is computed using the summation of its *acnt* and *pcnt*. For more information about computing the possible count and the error in the support of significant itemsets and other aspect of the *estWin* algorithm, the interested readers can refer to [11]. After insertion of an itemset, as new transactions arrive, old transactions are removed from the window and the error in the support of the itemset is reduced. This reduction of the error is due to removing the effect of old transactions and reduction in the value of possible count.

Sliding window based frequent itemsets mining over data streams can be categorized into two main groups. In the first group [3-6], a window is maintained over recent transactions and the mining process is started to extract frequent patterns within the current window when the user submits a request. Therefore, the aim is to store transactions using low memory usage and performing the mining process efficiently. In the second group [7-11], the mining results are continuously updated by inserting new transactions to the window and removing old transactions from the window. Therefore, fast updating of the mining result and low memory usage for the mining result and transactions of the window are desirable. Algorithms of the second group are more useful for the user since they can see the mining result immediately when it is required. For fast processing, an approximate algorithm which can identify set of frequent itemsets within sliding window with high quality of the result is acceptable. Algorithms of the first group [3-6], do not adaptively maintain and update the mining result. Therefore, after the mining, when new transactions are arrived from the stream, obtained result becomes invalid for the user and thus the mining task need to be re-executed. On the other hand, applying a mining algorithm on the whole of the window needs considerable processing and memory requirements especially in large window size with respect to algorithms of second group [6-10] for which the mining results are updated adaptively. However, the performance of these algorithms is relatively low as compared to high arrival rate of input streams. Therefore, in this study, an efficient approximate algorithm belonging to the second group for adaptive maintenance of frequent itemsets over sliding window is proposed.

## 3. PROBLEM STATEMENT

Let $I=\{i_1,i_2,\ldots,i_m\}$ be a set of items. Suppose that *DS* be a stream of transactions received in sequential order. Each transaction of *DS* is a subset of *I*. For an itemset *X*, which is also a subset of *I*, a transaction *T* in *DS* is said to contain the itemset *X* if $X \subseteq T$. Each transaction has an identical number named *Tid*. A transactional sliding window *W* over data stream *DS* contains $|W|$ recent transactions in the stream, where $|W|$ is the size of the window. The window slides forward by inserting a new transaction into the window and deleting the oldest transaction from the window. Due to efficiency issues, instead of a single transaction, the unit of insertion and deletion can be a pane (or batch) of transactions. In fact the window contains the *n* most recent panes of transactions of the input stream. The first transaction id (*Tid*) of each pane is regarded as pane id

(*Pid*) of that pane and first *Pid* of the window is named window id (*Wid*).

**Example 1:** Consider a data stream shown in Table 1 including a window with 2 panes. First, second, third and fourth columns show window id pane id, transaction id and items, respectively. Each pane contains 4 transactions. That is, pane size is 4. We use this sample data stream to describe the proposed algorithm.

**Table 1. An example of a data stream containing two panes of transactions.**

| Wid | Pid | Tid | Items |
|---|---|---|---|
| **1** | **1** | 1 | a, b, c |
| | | 2 | a, b, d |
| | | 3 | b, d |
| | | 4 | a, b, c |
| | **5** | 5 | b, d |
| | | 6 | a, c |
| | | 7 | a, b, c |
| | | 8 | a, b, d |

An itemset $X$ is said to be frequent in $W$ if $Freq(X) \geq n \times |P| \times s$, where $Freq(X)$, $n$, $|P|$ and $s$ are frequency of $X$ in $W$, number of the pane in the window, pane size and the minimum support threshold, respectively. The pane size and number of panes in each window are fixed during a data stream mining and are the parameters of the mining algorithm. Thus, having a pane based transactional window $W$ and a minimum support threshold $s$ specified by the user, the problem is defined as mining all frequent itemsets that exists in window $W$. The results should be continuously updated when the window advances. For storing and updating the set of frequent itemsets an ordered prefix tree is used. In this prefix tree, each node contains a single item. However, each node in this prefix tree represents an itemset which can be induced by item *id*s of nodes in the path from the root to the node. By prefix sharing among itemsets of the tree, a smaller amount of memory is required. All items resided in each path and all items of each level are ordered based on a canonical order. That is, from the root to a node of each path and from the left to the right of each depth an ordered set of items is visited. Due to unbounded and rapid rate of incoming transactions, maintaining a high quality approximate set of frequent itemsets belonging to the active window at each moment is acceptable.

## 4. THE PROPOSED ALGORITHM

Pane based windows were shown to be useful for querying and mining data streams [9, 19]. In transaction based sliding window algorithm like *estWin* [11], the unit of insertion and deletion is a single transaction, therefore, for a newly inserted itemset the actual count of only one transaction is available in time of insertion. This transaction is the one which causes the insertion of the itemset. However, this itemset might have occurred in previous transactions of the current window. Therefore, within these transactions, the support of the new itemset should be estimated or exactly calculated using stored transactions. However, in a pane based window, the support of a new itemset within the new pane is available and its remaining support from previous panes can be estimated or calculated. This leads to better and more accurate identification of new itemsets especially if the pane size is considerable with respect to the window size since the actual support

of newly emerged itemset in the new pane is available.

On the other hand, insertion and deletion of panes are more efficient with respect to transaction based processing since transactions are batch-processed together and an identical task is performed on the set of transactions. However, a pane based adaptation of the *estWin* algorithm raises some challenges. The first challenge is concerned with the process of updating frequent itemsets and insertion of new itemsets when a new pane arrives. The support of an itemset must be computed by searching the new pane. Searching for itemsets within a set of transactions is a time consuming task. Moreover, the prefix tree must be visited twice using the new pane, first for updating current itemsets and second for insertion of new itemsets. The visiting process takes too much time especially when there are large number of itemset for updating and insertion. The second challenge occurs during the pane deletion process where the effect of deleting old transactions has to be removed from supports of frequent itemsets. A similar time consuming process must be also performed on the transactions of the old pane. Third, the panes of the current window must be stored to use them for updating when they become expired.

To cope with these drawbacks, in this study, a new pane based online algorithm named *pWin* is proposed which overcomes all mentioned problems and has good runtime and memory usage. Additionally, its result is more accurate with respect to transaction based window algorithms like *estWin*. Our proposed algorithm is composed of two main phases, window initialization and window sliding. The novel algorithm, benefits from new techniques to enhance both runtime and memory usage. Similar to the *estWin* method, the *pWin* uses a prefix tree to store and maintain frequent itemsets of sliding window.

### 4.1 Window Initialization

In our algorithm, the incoming data stream is fed into the algorithm using equal sized panes of transactions. Each pane of transaction is stored in form of *Tidset*s.

**Definition 1 [20]**. The *Tidset* of each item is a set containing the *Tid*s of transactions in which the item has appeared. If a transaction contains an item, its *Tid* will appear in that item's *Tidset*.

After arrival of the first pane of transactions, the set of frequent itemsets of this pane is mined using the minimum support threshold. Eclat [20] algorithm is used for this purpose. The set of frequent itemsets are stored in the prefix tree structure. In the remaining process this prefix tree is used to update the mining result. Different elements of each node which represent an itemset in this prefix tree are described in Table 2. Similar to the *estWin* algorithm, a reduced minimum support named *minimum significant* is used for early monitoring of the support of the itemsets. After the first pane, by arrival of subsequent panes, the mining result is updated and new significant itemsets are identified and inserted to the prefix tree. This process continues until the window becomes complete. Subsequently, the window sliding phase starts.

**Table 2. Information contained in each node of the prefix tree**

| Element | Purpose |
|---|---|
| ID | Item ID |
| AC | Actual Count of the Itemset |
| PC | Potential Count |
| ACs | Array of actual counts of panes of the window |
| FPid | Pid of the pane that causes the node to be inserted to the prefix |
| Pids | Pane IDs of panes after the insertion of the node which contain the itemset |
| ER | Computed Error for the support of the itemset represented by the node |
| Children | Set of pointers to the children of the node |

### 4.2 Window Sliding

The window sliding phase consists of new pane addition and old pane deletion. A new pane, composed of *Tidset*s of items, is constructed and updated by arrival of a new transaction. After arrival of a number of transactions equal to the pane size, the set of *Tidset*s is ready for updating the prefix tree. This process includes updating of old itemset stored in the prefix tree, removing of insignificant itemsets from the tree and also insertion of newly identified significant itemsets to the prefix tree.

#### 4.2.1 Prefix Tree Updating

The *Tidset*s of newly arrived pane is used to perform a depth first addition process. In our algorithm conditional panes are used to update prefix tree structure at different depths. A conditional pane can be generated for a node of the prefix tree.

**Definition 2.** Conditional pane of each node contains some *Tidset*s. Each *Tidset* contains *Tid*s of transactions that contain both the item and the itemset represented by that node.

Conditional pane of the root node of the prefix tree is the complete set of *Tidset*s of the newly arrived pane. Since all single items having at least one occurrence are maintained in *Tidset*s, each *Tidset* of this pane can be used to update the nodes of the first depth in the prefix tree. However, a depth first traversal of the prefix tree is used in which conditional panes are generated after visiting and updating each node to process nodes at lower depth. *Tidset* intersections are used to construct a conditional pane. The process of updating is illustrated using an example of a prefix tree shown in Fig. 1 for its left most branch.

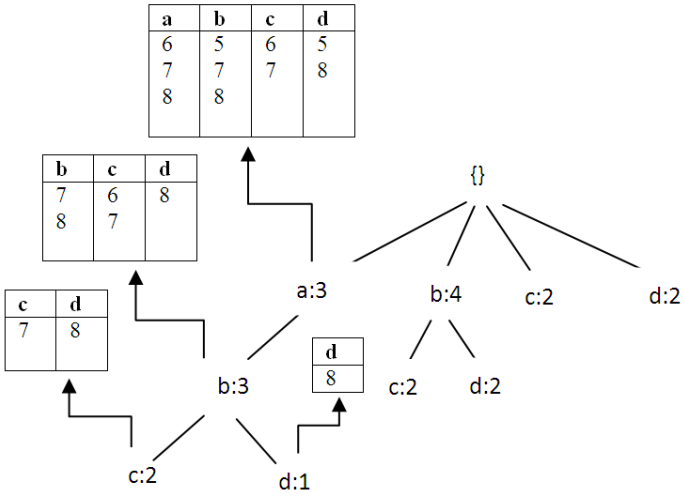


Fig. 1. Updating prefix tree using conditional panes.

This figure shows the state of prefix tree during insertion of the second pane of Example 1. For sake of presentation, details of the nodes in the monitoring tree are not depicted. Moreover, the count values of the nodes after updating are not shown. The full content of the pane from the root

of tree is used to update all corresponding branches by constructing conditional pane of each node. For each node of the left most branch, the conditional pane that is used to update its actual count is shown in Fig. 1. During the traversal process, for each item that resides in a node, the size of corresponding *Tidset* is added to the count value of the node. In Fig. 1, for the node containing item "a", the value of 3 is added to its support and thus its value becomes 6 since the size of *Tidset* of item "a" in the corresponding conditional pane is 3. Subsequently, before processing its first child, the corresponding conditional pane of that child must be constructed. For this purpose, *Tidset* of item "a" is intersected by *Tidset*s of child node's item and its subsequent *Tidset*s. Since node containing item "b" is first visited in the second depth, after generating conditional pane for this node, its count is added by the size of *Tidset* "b" in conditional pane, i.e., 2 and thus its count becomes 4. This is due to the number of transactions of the complete pane that contains both items "a" and "b", i.e., count of itemset "ab" is two. For the next depth, other conditional panes are generated using set intersection of *Tidset* of item "b" and other *Tidset*s. At node c, its count is added by size of c's *Tidset* and the new value becomes 4. In fact, the number of transactions that contain "a", "b" and "c" is two. That is, itemset "abc" has actual count 2 in the complete pane. For node representing itemset "abd", the conditional pane only contains one *Tidset* of item "d" whose size is 1. This conditional pane is constructed by interesting *tidset*s of "b" and "d". Therefore, the count of the node "d" at third depth becomes 3.

Conditional panes are generated irrespective to existing of the corresponding node. For a branch of the tree, the process continues until an empty conditional pane is constructed or if there is not any child node corresponded to a generated conditional pane. In the second case, remaining items of the conditional pane are tested for possible insertion of new node, i.e. adding a new itemset into the prefix tree. If an itemset represented by the current node and an item remaining in the conditional pane has a significant greater than the minimum significant, a new node corresponded to the item is inserted to the tree. The value of significance is composed of two other values, the actual count of the itemset in the conditional pane and the estimated count using the subsets of the itemset. The estimated count and error in this estimated count are calculated by the approach proposed in [11]. Therefore, for an itemset which corresponds to a node, all of its subsets must be visited first since the estimated support is calculated based on them. On the other hand, due to *Apriori* principle a new itemset is inserted once all of its subsets become significant and inserted. In order to achieve this goal, we use a reverse order depth first traversal of the prefix tree which is described later. For an existing node, its actual support is updated. For a newly inserted node its actual count is set to the size of its corresponding *Tidset* in the conditional pane and its potential count is set to the estimated support computed using the subsets of the itemset. Moreover, the *Pid* of inserted pane is stored in a newly inserted node as its *FPid*. This value is used to update potential count of the itemset in the pane removal process. Furthermore, for both updated node and newly inserted node, the *Pid* is also appended in the array of the *Pid*s of that node. Values stored in this array are used to update the actual count of the node again in the pane removal process.

Another task performed by pane addition process is pruning of insignificant itemsets. Although, the support of an itemset increases by the updating process, however, its significant may become lower than minimum significant since it is a relative value with respect to the window size. Therefore, such nodes should be erased from the prefix tree. Additionally, based on the *Apriori* property, all of its descendents are also removed from the prefix tree and it is not required to visit them during the pane addition process.

### 4.2.2 Reverse Order Depth First Traversal

As mentioned previously, updated subsets of an itemset is required for adding the corresponding node. Therefore, the pane addition process must visit all subsets of an itemset before the itemset itself. In an ordered prefix tree, some subsets of an itemset are placed after the itemset itself, according to the order of prefix tree nodes when a depth first traversal is used. That is, if we perform a depth first traversal, some subsets of a itemsets reside in other nodes of the tree are visited after the node containing the itemset. The *estWin* algorithm overcomes this problem, by traversing the prefix tree twice, once for updating of the itemsets and the other for insertion process. However, this violates the single processing of elements rule in data stream mining and diminishes the performance. Moreover, for a pane based window, it has a higher cost since the new pane containing a couple of transactions must visits related nodes twice. Therefore, we have proposed a new traversal strategy of the prefix tree during the pane addition process for both nodes' update and new nodes insertion processes by only one traversal. In this strategy, the prefix tree is traversed in a depth first reverse order manner. That is, visiting nodes are carried out in the reverse order of prefix tree nodes arrangement. In this way, for an itemset, all of its subsets are visited first. Therefore, for estimating the support of a new significant itemset, all updated support values of its subsets are available. Moreover, some subsets of the itemset might be inserted to the prefix tree before the itemset itself. This facilitates early insertion of the itemset.

The reverse order depth first traversal strategy is explained using a prefix tree depicted in Fig. 2. In this figure, alphabetical ordering of items is used to arrange the prefix tree. In fact, this figure is the complete lattice for set of items {a,b,c,d} in the form of a prefix tree. For sake of presentation, itemset induced by the path from the root to each node is shown in that node. The order for visiting branches are shown using numbers. The branches are visited from right to left. Each branch is processed from top to down. In other words, it is a depth first traversal performed in a reverse order of nodes.

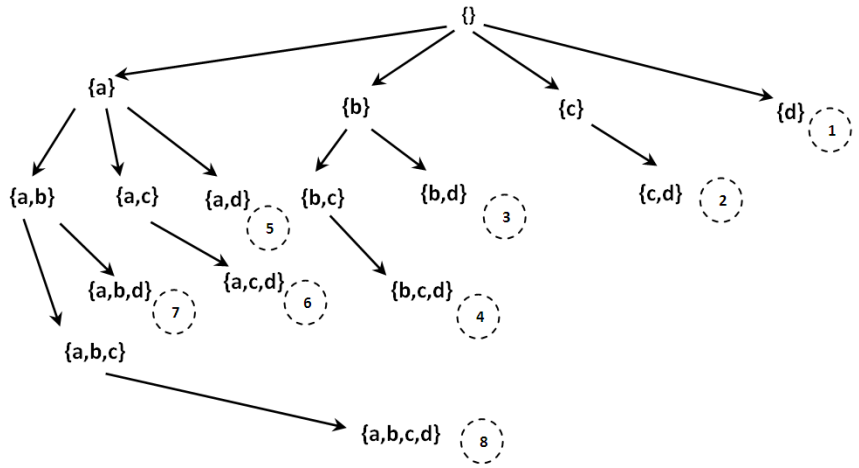


Fig. 2. A reverse order depth first traversal of a prefix tree

For example, as indicated in Fig. 2, set of visited itemsets before “a,b,c,d” is {{d},{c}, {c,d},{b}, {b,d},{b,c}, {b,c,d}, {a},{a,d}, {a,c}, {a,c,d}, {a,b}, {a,b,d}, {a,b,c}}, all of which are subsets of “abcd”. For inserting a new itemset all of its supersets must be included in the tree. For support estimation of a k-itemset, as described in [11], only (k-1)-itemsets are required which are

obviously visited before the itemset. Therefore, by using this strategy both updating and inserting itemsets cause by new pane addition are performed together in one traversal of the prefix tree. It is important to note that, those nodes in the monitoring prefix tree are visited that the corresponding itemsets are included in the newly inserted pane. Therefore, whole of the tree nodes are not visited in a pane addition process.

### 4. 2. 3 The Pane Addition Algorithm

Based on the above descriptions, Fig. 3 shows the pseudo-code of the pane addition procedure. This procedure is used in the window initialization (after inserting the first pane) and window sliding phases to insert a new pane of transactions. The procedure is called for each related node of prefix tree using a conditional pane generated for that node, *Pid* of inserted pane of the window, minimum significant, current depth and an itemset corresponding to the node. All symbols used in this procedure are summarized in Table 3.

**Table 3. All symbols and parameters used in the Pane Addition procedure and their meaning**

| Symbol | Meaning |
|---|---|
| Node | Current node of the tree to be processed |
| Pane | A new pane of transactions containing tidsets of items |
| Pid | The id of the new pane inserted to current window |
| Sig | Reduced minimum support used to early insertion of itemsets |
| depth | The current depth of recursion |
| Itemset | The current Itemset to be processed |
| D | Current number of transactions of the window |
| Generate | A function to generate a conditional pane |
| Compute | A function to compute PC and ER using the prefix tree |

This procedure is first called for the root node, complete pane and depth of one. The procedure subsequently is called by a conditional pane and nodes in different depths of the tree recursively. *Tidset*s of the complete pane are ordered according to the canonical orders of their items. Generated conditional panes also preserve this ordering. *Tidset*s of each conditional pane is processed in the reverse order of items as shown in the first line in the *for* statement. For each item of a pane, if corresponding node exists, its support is updated. If the updated support is lower than minimum significant, the node and all of its descendents are removed from the tree. Otherwise, as shown in lines 7 through 10, a new conditional pane is generated for updating children at lower depth and subsequently a function is called recursively for updating their actual counts. In Fig. 3, *D* is the current number of transactions within the window. In the window initialization phase, this number is smaller than window size and in the window sliding phase it is equal to the window size. In lines 11 through 21, for an item which does not exist as a child node, it must be checked for possible insertion as a new child. If we are in the first depth, the item is inserted as a new child of the root node irrespective to its count value since we monitor all singletons. Otherwise, its *PC* and *ER* are estimated using the subsets of corresponding itemset. After the estimation, if the actual and potential counts together exceed the minimum significant value, it is inserted as a new child of the processed node. The actual count of the new node is equal to the size of corresponding item in the conditional pane. Subsequently, the Addition procedure is called for newly inserted node after generating corresponding conditional pane. For creating a new node, its actual

count, *Pid* of current pane as its *FPid*, estimated *PC* and *ER* are given. For a single item, both *PC* and *ER* are equal to zero since its support is monitored from scratch and has not potential count and error.

```
Addition(Node, Pane, Pid, Sig, depth, Itemset)
{
 1)   For( it = Pane.end to Pane.start)
 2)     If( it ∈ Node.children)
 3)           Node = Node.children(it);
 4)           Node.AC = Node.AC + it.listsize;
 5)             If(Node.AC / D < sig)
 6)                   remove Node and all of its descendent
 7)             Else
 8)                   Pane = Generate(it, Pane)
 9)                   itemset = itemset + it.item
 10)                  Addition(Node, Pane, Pid, Sig, depth+1, itemset);
 11)  Else
 12)    If( depth ==1)
 13)          Node.children [it] = new Node(it.listsize , Pid , 0, 0)
 14)    Else
 15)          itemset = itemset + it.item
 16)          (PC, ER) =Compute(root, itemset)
 17)          If((it.listsize+PC) / D > Sig)
 18)                Node.children [it] = new Node(it.listsize, Pid , PC, ER) ;
 19)                Pane = Generate(it, Pane)
 20)                itemset = itemset + it.item
 21)                Addition(Node.children [it], Pane, Pid, Sig, depth+1, itemset);
}
```

Fig. 3. Pane Addition

After new pane addition, a removal process is required to delete oldest pane of the window. In this way, fixed size window is preserved. In contrast to the *Addition* process, deletion does not require using the oldest pane of transactions and the action is performed only on the prefix tree using stored pane information.

### 4.2.4 Oldest Pane Deletion

When the window advances, the oldest pane of the window should be removed. As mentioned previously, when a pane causes a node to be inserted to the tree, its *Pid* and *Pid*s of subsequent panes, when they will arrive, are stored in the node. In another array (*ACs*), for each of these panes, corresponding actual counts are also stored. Due to the stored pane information, it is not required to store transactions of the window in order to remove their effects when they become obsolete. For oldest pane, its contribution should be removed from the set of significant itemset represented as the nodes of the prefix tree. Removing the oldest pane from the window is performed by visiting the nodes of the prefix tree. For each node of the tree if the list of *Pid*s contains the oldest pane id, it is removed. Moreover, the actual count of the pane is removed from

*ACs* list and it is also subtracted from the actual count of the itemset in the window (*AC*).

On the other hand, if the list of *Pid*s does not contain the removed pane id and *FPid* of the node is greater than the removed *Pid*, only the value of potential count is corrected. The *PC* is replaced by the value of formula 1, if the value is smaller than the current *PC* value.

$$(FPid - Wid) \times S_{sig}$$

Where *FPid* and *Wid* are the *Pid* causes the itemset to be inserted to the tree and *Pid* of the first pane of the window, respectively. If the *FPid* of the node becomes equal to the *Wid* due to the pane removal process, the value of the error (*ER*) and potential count (*PC*) of the node become zero since we have the actual count of the itemset in whole window. Another task performed during the pane deletion process is erasing the nodes which become insignificant. For each insignificant node, all descendants are also removed since they belong to supersets of corresponding itemset of the node. Similarly, this is performed according to the *Apriori* property since any superset of an insignificant itemset is also insignificant.

### 4.2.5 Computation Complexity

In this subsection, we compare computation complexity of *pWin* with respect to the *estWin* algorithm. For this purpose, suppose that, there are b items within |W| transactions of the window. In the worst case scenario, all items are frequent. Based on the form of the prefix tree used by the *estWin* and *pWin* algorithms, its greatest branching degree is b (i.e., for the root node) and its smallest branching degree is 1 (i.e., for the right most node of the first depth, see Fig. 2). Branching degree for other nodes are between these two values. However, we regard b/2 for average branching factor. As can be inferred from figure Fig. 2, if all items are stored in the tree, depth of the prefix tree is b (consider the left most branch). On the other hand, right most branch is shallowest one having depth 1. Depths of other branches are within these two values. Similarly, we regard b/2 for average branching factor. Based on these assumptions, we analysis computation complexity of *estWin* and *pWin* algorithms for transactions of one window. For *estWin* algorithm, a window sliding requires three prefix tree traversal, two traversals for inserting a new transaction and one traversal for deleting the oldest one. Suppose that, in the worst case, a transaction causes all nodes of the prefix tree to be observed. Based on above assumptions about the prefix tree, a cost of one traversal is $O((b/2)^{b/2})$. Thus, for |W| transactions of a window, the computation cost is:

$$3\times|W|\times(b/2)^{b/2} \in O(|W|\times(b/2)^{b/2})$$

On the other hand, for each sliding, *pWin* observes related nodes of the tree two times. Moreover, in this algorithm, the number of transactions |W| divided by pane size |P| is equal to the number of sliding. We regard a constant number *K* for |W|/|P| where the pane size is considerably large with respect to the window size.

$$2\times|(W|/|P|)\times(b/2)^{b/2} = K\times(b/2)^{b/2} \in O((b/2)^{b/2})$$

Therefore, traversal cost of *pWin* is lower than the *estWin* algorithm. The above cost analysis is with the assumption of all items to be frequent which is the rare case. In fact, usually a considerable number of items are pruned away even in low values of minimum support thresholds.

## 5. EXPERIMENTAL RESULTS

In this section, the proposed algorithm is experimentally evaluated and compared to the *estWin* and *SWIM* algorithms. These algorithms are selected since the first algorithm is an approximate algorithm and second is a pane based method. We have implemented *estWin*, *SWIM* and the *pWin* algorithms using C++ language and *Dev C++* IDE. The *STL* template library of C++ is used to implement required data structures. All experiments are executed on an Intel Pentium 4 CPU 3.0-GHz machine with 2-GB RAM running on Windows XP. In the pane addition of the *pWin*, to enhance the runtime, a heuristic is used. In this heuristic, for small depths of the prefix tree, *diffset*s [21] of conditional panes are used and in the subsequent depths we switch to *Tidsets*. This is due to using shorter lists in corresponding situations.

Since the memory and processing time are two main factors of every data stream mining algorithm, we compare the algorithms in terms of these factors. Similar to the *estWin* algorithm, our algorithm is an approximate algorithm. That is, it might miss some frequent itemsets or identify some infrequent patterns as frequent. Therefore, in the last experiment, the quality of the result is evaluated and compared to the *estWin* method in terms of false negatives and false positives. In order to show the effectiveness of the *pWin* algorithm, various real and synthetic datasets with different data sizes have been selected for experimentations. We use three real datasets *BMS-POS, Kosarak* and *BMS-Webview-2* and one synthetic dataset *T40I10D100K* to accomplish empirical evaluations. The latter is generated using synthetic data generator as described in [2]. Specifications of all datasets are summarized in Table 4.

**Table 4. Datasets specifications**

| Dataset | #trans | #items | Max. length | Avg. length |
|---|---|---|---|---|
| BMS-POS | 515,597 | 1657 | 164 | 6.53 |
| T40I10D100K | 100,000 | 1000 | 77 | 40 |
| Kosarak | 990002 | 41270 | 2498 | 8.1 |
| BMS-Webview-2 | 77512 | 3340 | 161 | 5 |

### 5.1 Runtime

The parameters that impact the runtime on *estWin* and *pWin* algorithms are minimum support and minimum significant thresholds. However, *SWIM* does not require minimum significant threshold. In this experiment for *BMS-POS* dataset, algorithms have window size equal to 200K transactions. The pane size for our algorithm and *SWIM* and also the force pruning for the *estWin* are set to 50K transactions. For *T40I10D100K* dataset, window size is set to 40K, pane size is set to 10K for *pWin* and *SWIM,* and force pruning is set to 10K for the *estWin* algorithm. For Kosarak and BMS-Webview-2 datasets window sizes are set to 200K and 20K, respectively. Pane sizes used for these datasets are 50K and 5K. We use the minimum significant 0.5 for *pWin* and *estWin* and vary the value of minimum support threshold for all algorithms to see the impact of minimum support threshold on the runtime. For each algorithm, total required time to process

each dataset is measured. The results are shown in Fig. 4. Vertical and horizontal axises show runtime and minimum support respectively.

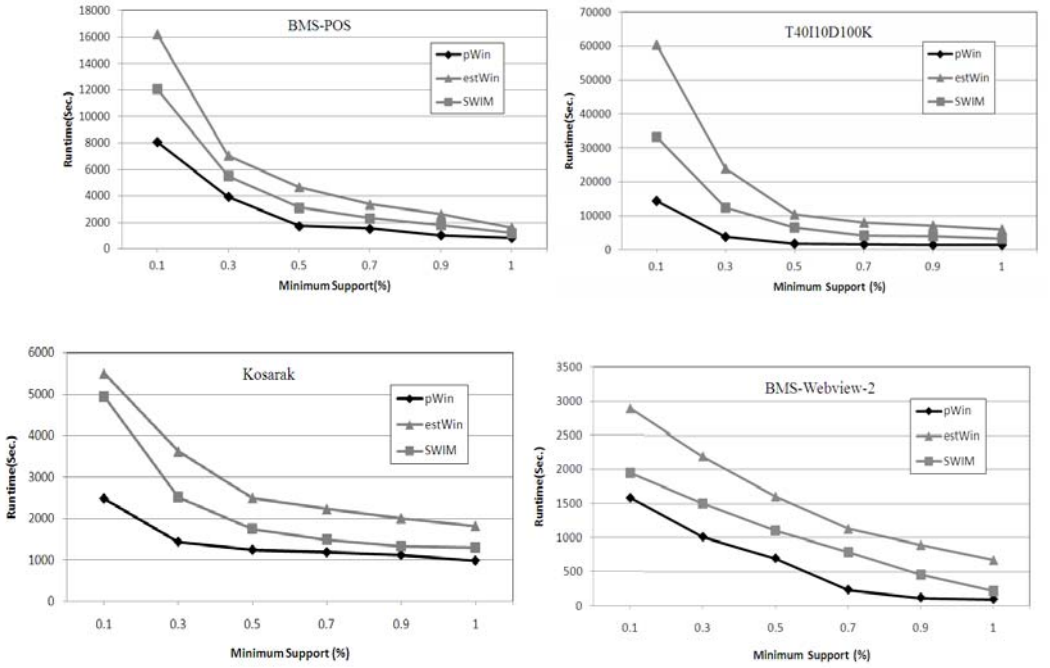


Fig. 4. Runtime comparisons for different minimum support threshold values (Significant= 0.5).

As shown in Fig. 4 for all datasets, the proposed algorithm is faster than the *estWin* and *SWIM* algorithms. The performance gain of our algorithm is considerable for almost all values of minimum support thresholds. However, performance gain of the *pWin* increases as minimum support threshold decreases. The superiority is due to batch processing of transactions and also performing fast removal of old information belonging to the removed pane from the set of frequent patterns. Moreover, in *pWin* both of the pane removal and pruning of insignificant itemsets are performed simultaneously while the *estWin* algorithm has a separate phase of pruning named force pruning which is performed periodically. Furthermore, by using the reverse order depth first traversal, both updating of current itemset and insertion of new patterns are performed simultaneously. On the other hand, in the *estWin* algorithm, related paths for every incoming transaction are visited twice, one for itemset updating and the other for possible itemset insertion to the tree. In our algorithm, by arrival of a pane, new itemset identification and support updating for previously found frequent itemsets are performed together while the *SWIM* algorithm applies the *FP-Growth* algorithm on new pane to detect frequent itemset of the pane and then separately verifies the support of previous frequent itemsets. Our algorithm computes the support of new itemset within new pane and estimates the support on previous pane of the window while in *SWIM*, the support of new itemset is verified in all panes of the window which is a time consuming process.

### 5.2 Memory

In this subsection the memory usage of all three algorithms are compared together. For each

algorithm, average memory usage of all windows on each dataset is measured. Memory required by the *estWin* algorithm is composed of memory of the prefix tree and the memory required for storing raw transactions of windows. In *SWIM*, for storing transactions of each pane, a *FP-Tree* is used. Moreover, the union of sets of frequent itemsets of all panes is maintained separately. On the other hand, the *pWin* algorithm only maintains the prefix tree structure for itemsets and does not physically store transactions of the window neither in the form of *FP-Tree* nor in the form of raw transactions. Therefore, *pWin* should have lowest memory usage. For different value of minimum supports and fixed minimum significant of 0.5 (for *estWin* and *pWin*), average memory required for all active windows are computed and the results are plotted in Fig. 5. This figure shows that, for all datasets, the *pWin* has lowest memory requirement.

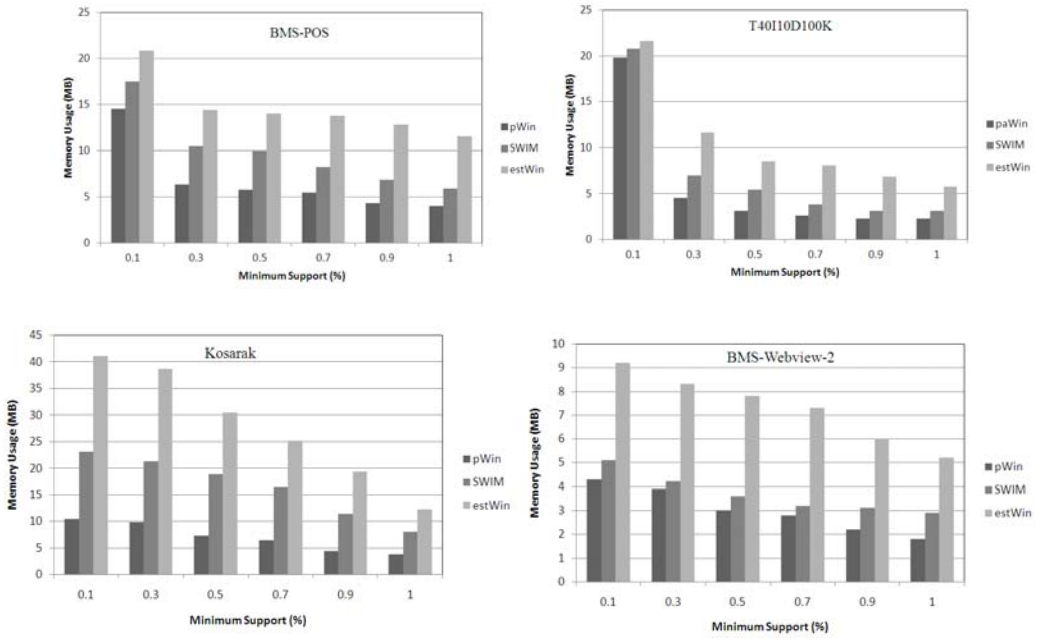


Fig. 5. Memory usage comparison with respect to minimum support threshold.

Although, the *pWin* stores extra information in each prefix tree node (including pane id and support information of each pane), it does not maintain raw transactions of the window. This is the reason of memory usage superiority of the *pWin* algorithm with respect to the *estWin* and *SWIM*.

In order to show the effect of window sizes on the memory usage of all algorithms, we measured memory of algorithms with respect to various window sizes. In this experiment, for each window size, average memory occupied by each algorithm during a data stream mining is measured and the result is depicted in Fig.6.

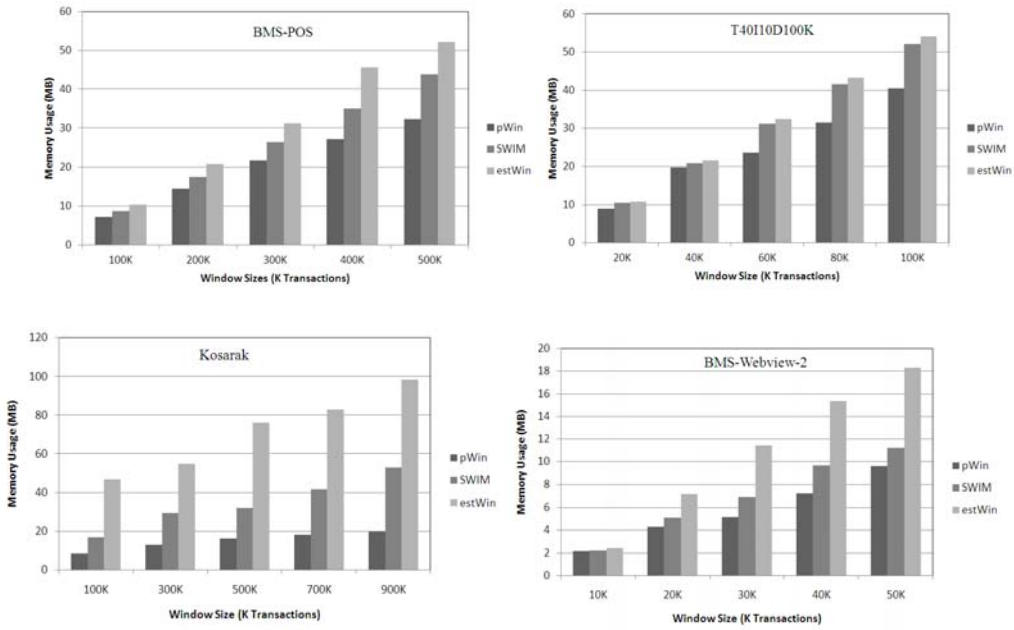


Fig. 6. Memory usage comparison with respect to different window sizes.

As shown in this figure, for all dataset, as the window size increases, each algorithm requires more space. However, for all window sizes, the *pWin* algorithm occupies lower memory. Besides the *pWin*, *SWIM* and *estWin* require larger amount of memory for storing raw transactions in large window sizes.

### 5.3 Accuracy Analysis

Since the new algorithm produces an approximate result for an input data stream, the quality of the result must be measured. The number of false positives and false negatives are important factors for approximate frequent itemset mining algorithms. False positives waste memory and processing time. On the other hand false negatives reduce the quality of the mining results. Vast numbers of false positives and negatives cause the results to be undependable. In [11], it is noted that the *estWin* dose not has any false negative in the mining results. However, in this algorithm, an itemset can not be identified as significant before all of its subsets are significant and are inserted to the prefix tree. Therefore as our experimentation also confirms, the *estWin* algorithm misses some frequent itemsets during the mining process. We report the total number of false positives and false negatives on *BMS-POS* and *T40I10D100K* datasets for three different minimum support thresholds and fixed minimum significance 0.5 for *estWin* and *pWin* algorithms. The *SWIM* is an exact algorithm and does not produce any false positive or negative. For computing the accuracy, set of frequent patterns of each window (extracted by the Apriori algorithm [2]) is compared by the result of each algorithm for that window. Total number of false positives and false negatives of all windows are enumerated and reported in Table 5. As shown in this table, for all minimum support values of both datasets, our algorithm has smaller number of false positives and false negatives. For *BMS-POS* dataset, numbers of false positives and negatives of our algo-

rithm are far less than the *estWin*. For *T40ID10D100K* dataset, our algorithm does not have any false negative and the number of false positives for each minimum support value is negligible. Therefore, the devised algorithm is more accurate and its mining result is more dependable for the user.

**Table 5. Number of false positives and false negatives (Minimum Significance = 0.5)**

| | BMS-POS | | | | T40I10D100K | | | |
|---|---|---|---|---|---|---|---|---|
| | *estWin* | | *pWin* | | *estWin* | | *pWin* | |
| MinSup(%) | FP | FN | FP | FN | FP | FN | FP | FN |
| 1 | 7157 | 8564 | 1907 | 178 | 19 | 260719 | 9 | 0 |
| 5 | 519 | 460 | 106 | 20 | 5820 | 25 | 1 | 0 |
| 9 | 69 | 218 | 24 | 6 | 916 | 0 | 0 | 0 |

The reason of this supremacy is that in our algorithm a new candidate itemset is evaluated, for possible insertion to the prefix tree, based on their actual support in the new pane and estimated support in the previous panes of the window. The new pane constitutes a considerable number of transactions of the window. On the other hand, in the *estWin* algorithm, actual support of a new itemset is restricted to the newly arrived transaction and its support is mainly computed based on estimation using old transactions of the window. In our algorithm, having a wider range of actual support for an itemset, the computed support becomes more realistic and the probabilities of being false positive or false negative are decreased.

## 5. CONCLUSION

In this study, an efficient algorithm for mining set of frequent itemsets over data streams has been proposed. Although, this algorithm operates under sliding window model, it does not store sliding window transaction. Moreover, by batch processing of transactions and using new techniques it becomes a fast algorithm which is suitable for high speed data streams. These techniques facilitate updating and insertion of the itemsets by arriving a new batch of transactions and removing the effect of the oldest batch from the mining results. In this algorithm, always the mining result of the active window is available for the user and is continuously updated. Experimental evaluations show that, this algorithm has better runtime and memory usage with respect to previously proposed *estWin* and *SWIM* methods. Moreover, it has better mining result quality with respect to the *estWin* as an approximate algorithm. An advantage of the *estWin* over the *pWin* and *SWIM* is that it is more online since its result is updated by every insertion or deletion of a single transaction. However, our proposed traversing strategy is general and can be also exploited in the *estWin* algorithm to enhance its runtime. In large minimum support thresholds, the runtime of *SWIM* is close to the *pWin*. Therefore, since *SWIM* generates the complete set of frequent patterns, its execution time could be lower than that of the proposed algorithm under the accuracy consideration. The proposed algorithm can be operate also in time sensitive sliding window in which at each time a different number of transactions are received from a stream. Moreover, if the pane deletion becomes removed from the algorithm it can also work in the landmark model

## REFERENCES


[1] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions", *Data Mining and Knowledge Discovery,* Vol. 15, 2007, pp. 55–86.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules" in *Proceedings of International Conference on Very Large Databases*, 1994, pp. 487–499.

[3] C. K.S Leung and Q. I. Khan, "DSTree: a tree structure for the mining of frequent sets from data streams", in *Proceedings of IEEE International Conference on Data Mining*, 2006, pp. 928–932.

[4] S. K. Tanbeer, C. F. Ahmed, B. S. Jeong, and Y. K. Lee, "Sliding window-based frequent pattern mining over data streams", *Information Sciences,* Vol. 179, 2009, pp. 3843-3865.

[5] H. F. Li and S. Y. Lee, "Mining frequent itemsets over data streams using efficient window sliding techniques", *Expert Systems with Applications,* Vol. 36, 2009, pp. 1466–1477.

[6] M. Deypir and M. H. Sadreddini, "EclatDS: An Efficient Sliding Window Based Frequent Pattern Mining Method for Data Streams", *Intelligent Data Analysis,* Vol. 15(4), 2011, pp. 571-587.

[7] C. H. Lin, D. Y. Chiu, Y. H. Wu, and A. L. P. Chen, "Mining frequent itemsets from data streams with a time-sensitive sliding window", in *Proceedings of SDM International Conference on Data Mining,* 2005.

[8] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, "Catch the moment: maintaining closed frequent itemsets over a data stream sliding window", *Knowledge and Information Systems,* Vol. 10, 2006, pp. 265–294.

[9] B. Mozafari, H. Thakkar, and C. Zaniolo, "Verifying and mining frequent patterns from large windows over data streams", in *Proceedings of International Conference on Data Engineering,* 2008, pp. 179–188.

[10] H. Li and H. Chen, "Mining non-derivable frequent itemsets over data stream", *Data & Knowledge Engineering,* Vol. 68, 2009, pp. 481-498.

[11] J. H. Chang and W. S. Lee, "estWin: Online data stream mining of recent frequent itemsets by sliding window method", *Journal of Information Science,* Vol. 31, 2005, pp. 76–90.

[12] G.S. Manku and R. Motwani, "Approximate frequency counts over data streams", *Proc. VLDB Int. Conf. Very Large Databases*, 2002, pp. 346–357.

[13] H.-F. Li, S.-Y. Lee, and M.-K. Shan, "An efficient algorithm for mining frequent itemsets over the entire history of data streams", *Proc. Int. Workshop on Knowledge Discovery in Data Streams*, 2004.

[14] J.X. Yu, Z. Chong, H. Lu, Z. Zhang, and A. Zhou, "A false negative approach to mining frequent itemsets from high speed transactional data streams", *Information Sciences,* 176(14), 2006, pp. 1986–2015.

[15] X. Zhi-Jun, C. Hong, and C. Li, "An efficient algorithm for frequent itemset mining on data streams", *Proc. ICDM*, 2006, pp. 474–491.

[16] J. Chang, W. Lee, "Finding recently frequent itemsets adaptively over online transactional data streams", *Information Systems*, 31 (8), 2006, pp. 849-869.

[17] J.H. Chang, W.S. Lee, "estMax: Tracing Maximal Frequent Itemsets Instantly over Online Transactional Data Streams", IEEE Transactions on Knowledge and Data Engineering, 21 (10), 2009, pp. 1418-1431.

[18] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach", *Data Mining and Knowledge Discovery,* Vol. 8, 2004, pp. 53-87.

[19] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker, “No pane, no gain: efficient evaluation of sliding-window aggregates over data streams.” SIGMOD Record, 2005.

[20] M. Zaki, “Scalable algorithms for association mining”, *IEEE Transactions on Knowledge and Data Engineering,* Vol. 12, 2000, pp. 372-390.

[21] M. Zaki, and K. Gouda, “Fast vertical mining using Diffsets” in *Proceedings of 8 st International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 326-335.