Computers and Mathematics with Applications 64 (2012) 1706–1718

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

Max-FISM: Mining (recently) maximal frequent itemsets over data streams using the sliding window model

Zahra Farzanyar [a,*], Mohammadreza Kangavari [a], Nick Cercone [b]

[a] Department of Computer Engineering, Iran University of Science & Technology,Tehran, Iran

[b] Department of Computer Science and Engineering, York University, Toronto, Canada

ARTICLE INFO

Article history:
Received 24 November 2011
Accepted 11 January 2012

Keywords:
Data mining
Data stream
Maximal frequent itemset (MFI)
Single-pass algorithms
Sliding windows

ABSTRACT

Frequent itemset mining from data streams is an important data mining problem with broad applications such as retail market data analysis, network monitoring, web usage mining, and stock market prediction. However, it is also a difficult problem due to the unbounded, high-speed and continuous characteristics of streaming data. Therefore, extracting frequent itemsets from more recent data can enhance the analysis of stream data. In this paper, we propose an efficient algorithm, called *Max-FISM* (**Max**imal-**F**requent **I**temsets **M**ining), for mining recent maximal frequent itemsets from a high-speed stream of transactions within a sliding window. According to our algorithm, whenever a new transaction is inserted in the current window only its maximum itemset should be inserted into a prefix tree-based summary data structure called *Max-Set* for maintaining the number of independent appearance of each transaction in the current window. Finally, the set of recent maximal frequent itemsets is obtained from the current *Max-Set*. Experimental studies show that the proposed *Max-FISM* algorithm is highly efficient in terms of memory and time complexity for mining recent maximal frequent itemsets over high-speed data streams.

1. Introduction

A data stream is an unbounded sequence of data elements continuously generated at a rapid rate and have a data distribution that often changes with time.

Mining frequent patterns from data streams has become one of the most challenging problems in a wide range of applications including market basket analysis, traffic signals analysis, web click-stream mining, ATM transactions analysis, sensor network data analysis, etc.

There are some inherent challenges for data stream mining [1]. First, each data element can be examined at most once. Second, although the data elements are continuously generated, the consumption of memory space should be limited. Third, every incoming data element should be processed as fast as possible. Fourth, the analytical result of data stream should be available with an acceptable quality when users request results. Due to the characteristics of data streams, traditional frequent pattern mining algorithms cannot be directly applied.

In general, the mining result set includes a large number of frequent item sets. Therefore, closed or maximal frequent item sets are often used to represent them in a more compact notation but finding such item sets over online transactional data streams is not easy due to the requirements of a data stream [2].

* Corresponding author.
E-mail addresses: farzanyar@iust.ac.ir, z_farzanyar@iust.ac.ir (Z. Farzanyar).

0898-1221/$ – see front matter
doi:10.1016/j.camwa.2012.01.045

Depending on the stream processing model [3], the research of frequent patterns mining over data streams can be divided into three categories: the landmark window model, the damped window model and the sliding window model. In the landmark window model, the range of mining includes all the data between a specific timestamp, called the landmark, and the current time. In the damped window model (also referred to as the time fading window approach), each transaction is associated with a weight depending on the order of its appearance. In other words, the new transactions of data receive higher weights than the older ones. The last one is the sliding-window model. In this model, the range of mining is the length of the most recent transactions within a window. In other words, the basic processing unit of window sliding is either a time unit or an expired transaction.

Processing the recent data is usually important for the applications that handle stream oriented data. Therefore, the sliding window model is widely used to find recent frequent patterns in data streams.

To discover frequent itemsets on a data stream, we cannot afford to keep all itemsets or even frequent itemsets in the streaming environment, because of space and time constraints. On the other hand, any deletion may prevent us from discovering future frequent itemsets [4]. Therefore, the challenge resides in organizing a compact data structure which does not miss information of any frequent itemset over a sliding window and should be built with only one scan over the stream.

Motivated from these requirements, in this paper, we propose an efficient algorithm, called *Max-FISM* (**Max**imal-**F**requent **I**temsets **M**ining), for mining of recent maximal frequent itemsets from a high-speed stream of transactions within a sliding window. According to our algorithm, whenever a new transaction is inserted in the current window only its maximum itemset should be inserted into a prefix tree-based summary data structure called *Max-Set* for maintaining the number of independent appearance of each transaction in the current window. Finally, the set of recent maximal frequent itemsets is obtained from the current *Max-Set*.

Our comprehensive experimental results for both real and synthetic datasets show that the proposed *Max-FISM* algorithm runs significantly faster and consumes less memory than previous well-known algorithms such as Moment [4] and MFI-TransSW [5] when discovering recent frequent itemsets from a high-speed data stream.

The remainder of the paper is organized as follows. In Section 2, we briefly describe the existing algorithms for frequent pattern mining from a data stream. The problem is defined in Section 3. Section 4 presents the proposed algorithm. Experiments are discussed in Section 5. Finally, we conclude this work in Section 6.

2. Related works

Mining frequent itemsets from databases was first introduced by Agrawal et al. [6] in 1993. This technique produces the candidate itemsets of length k from the set of frequent itemsets of length $k-1$. The main performance limitations of Apriori-like approaches result from the requirement for multiple database scans and a large number of candidate itemsets. In 1999, Pasquier et al. [7] provided an improved theory that all nonempty closed sub-itemsets of a frequent closed itemset must be frequent as well. Following this theory, many other algorithms for closed itemset mining have been proposed [8,9]. Han et al. [10] proposed the frequent pattern tree (FP-tree) in 2000 which was the first attempt in mining frequent itemsets without candidate generation and reduces the number of database scans by two. Introduction of the data structure adopted in *FP-Tree* led to many other areas including mining frequent patterns with a prefix-tree structure from data streams [11].

Many algorithms have been proposed for finding frequent itemsets over data streams. However, because the scope of this work includes frequent itemsets mining using a sliding window model, we focus primarily on studies related to window-based algorithms.

The first effort to mine frequent itemsets over the entire history of data stream was proposed by Manku and Motwani [12]. They developed a single-pass algorithm, Lossy Counting based three module method BTS (Buffer-Trie-SetGen). The incoming stream is divided into segments and it processes a number of segments in every batch; this algorithm provides approximate results with an error bound.

The FDPM algorithm proposed in [13] is based on the Chernoff bound for false negative or false positive mining of frequent itemsets from high speed transactional data streams. It utilizes a running error parameter to prune itemsets and a reliability parameter to control memory. Like Lossy Counting, FDPM processes a data stream in the segment-based manner, while it is a false-negative oriented approach.

DSM-FI [11] is another algorithm that was developed to mine frequent itemsets over the entire historical stream data using landmark window method. DSM-FI extends prefix trees [10] for compact pattern representation, and adopts a typical top-down frequent itemset discovery scheme. Every transaction is converted into k (the total number of items in the transaction) small transactions and inserted into an extended prefix-tree-based summary data structure called the item-suffix frequent itemset forest.

Mining recent frequent patterns using the sliding window technique has also been studied in the literature. Lee et al. [14] proposed the SWF algorithm for mining of frequent itemsets within a sliding window, which is composed of a sequence of partitions. All candidate 2-itemsets are maintained independently. The candidate 2-itemsets of the new incoming partition are changed with every window sliding. Consequently, all candidate itemsets are generated from these candidate 2-itemsets. The set of frequent itemsets is generated by scanning the entire window. In this algorithm, to update the mining result, all the transactions within the current window should be re-scanned.

Chang and Lee [15] proposed the SWFI algorithm for finding frequent itemsets within a transaction-sensitive sliding window. A prefix tree lattice is constructed to store the current candidate itemsets and their frequencies. Another data

structure called current transaction list (CTL) maintains all the transactions in the range of current sliding window. For each incoming transaction, the algorithm inserts it into the CTL, counts the set of itemsets in the transaction and inserts the itemsets into the current monitoring lattice. During window sliding, every itemset embedded in the oldest transaction is deleted from the current monitoring lattice. At last, by traversing all the paths of the monitoring lattice, the set of frequent itemsets in the monitoring lattice are found.

The methods discussed above find approximate frequent itemsets with an error bound. Very few techniques [4,5] find the exact set of recent frequent itemsets from a data stream.

Chi et al. [4] proposed Moment algorithm to mine closed frequent itemsets within a transaction-sensitive sliding window from continuous streaming data. They presented an in-memory data structure, the *closed enumeration tree* (CET), which maintains a dynamical set of itemsets which contains (i) closed frequent itemsets, and (ii) itemsets that form a *boundary* between closed frequent itemsets and the rest of the itemsets. They further divide itemsets on the boundary into two categories, which correspond to the boundary between frequent and non-frequent itemsets, and the boundary between closed and non-closed itemsets, respectively. When a new transaction enters, it traverses the related parts of the CET and updates them. When a transaction is deleted from the current window, Moment also traverses the related parts of the CET and changes them. As long as the window size is not large, and the concept drifts in the stream are too dramatic, the effects of transactions moving in and out of a window cause change of status of many involved nodes. However, the exploration and node type checking are time consuming.

Li et al. [5] proposed an Apriori-based algorithm, called MFI-Trans-SW to mine the complete set of recent frequent itemsets from data streams. This algorithm uses the bit-sequence representation to keep track of the occurrence of all items in the transactions of the current sliding window. The MFI-TransSW algorithm consists of three phases: window initialization, window sliding and pattern generation. The window initialization phase is activated while the number of transactions generated so far in a transaction data stream is less than or equal to a user-predefined sliding window size w. In this phase, each item of the new incoming transaction is transformed into its bit-sequence representation. In MFI-TransSW algorithm, for each item X in the current transaction-sensitive sliding window TransSW, a bit sequence with w bits, denoted as Bit(X), is constructed. If an item X is in the ith transaction of current TransSW, the ith bit of Bit(X) is set to be 1; otherwise, it is set to be 0. The window sliding phase is activated after the current sliding window TransSW becomes full. A new incoming transaction is appended to the current sliding window, and the oldest transaction is removed from the window. To remove oldest information and to reflect the inclusion of new data it performs a bit-wise left-shift operation for all bit-sequences. When the up-to-date set of frequent itemsets within the current sliding window is requested, MFI-TransSW algorithm uses a level-wise method to generate the set of candidate itemsets with k items from the pre-known frequent itemsets with $k-1$ items according to the Apriori property [6]. Then, the algorithm uses the bitwise AND operation to compute the support (the number of bit 1) of these candidates in order to find the frequent k-itemsets. The candidate-generation-then-testing process is stopped until no new candidates with $k+1$ item are generated. Therefore, it suffers from the Apriori [6] limitation of vast candidate itemset generation, especially when mining stream data that include large number of long frequent itemsets with lower support thresholds. Moreover, since the algorithm maintains the bit-sequence information entirely for all items in the current window, it is not efficient in term of memory usage when the window includes large number of transactions and distinct items, which is very common in data stream environment.

Because the focus of the paper is on frequent itemsets mining over data streams within a sliding window, we compare with the algorithms Moment [4] and MFI-Trans-SW [5].

Comprehensive experiments show that the proposed algorithm not only acquires highly accurate mining results, but also runs significantly faster and consumes less memory than well-known algorithms Moment [4] and MFI-Trans-SW [5] for mining frequent itemsets over sliding windows.

3. Problem statement

We formalize our research problem as follows to explain the concepts of frequent itemsets mining over a data stream more formally. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n distinct literals, called *items* which are units of information in an application domain and let $D = [T_1, T_2, \ldots, T_m)$ be an infinite data stream, where each transaction $T_i \in D$, $i \in [1, m]$, is a subset of I and has a *unique Transaction IDentifier (TID)*. A window W consists a set of w recent transactions and slide transaction-by-transaction i.e., each slide of the window introduces a new transaction and removes the oldest transaction from the current window W.

Therefore, given a continuous data stream $D = [T_1, T_2, \ldots, T_m)$ and the user-specified minimum support threshold s in the range of [0, 1], mining of itemsets in W that have support no less than a minimum support threshold s is the problem of frequent itemsets mining in the data stream using the sliding window mechanism. Our objective is to develop a single-pass algorithm that discovers maximal frequent itemsets using a sliding window mechanism. To achieve this objective, we have designed *Max-FISM algorithm*, which finds maximal frequent itemsets in a sliding window manner.

4. The Max-FISM algorithm

We present the algorithm *Max-FISM* (**Max**imal-**F**requent **I**temsets **M**ining) for online mining of frequent itemsets in a sliding window of a continuous data stream. There are two phases in the proposed *sliding window* method. One is a *window*

initialization phase. This phase is activated while the number of transactions entered in the current window W is less than or equal to w. Therefore, a new transaction is entered to the current window and the total number of transactions in W is increased by one. The other is a *window sliding phase*. This phase is activated after the current window becomes full. A new transaction is entered to the current window and the oldest transaction is removed from the current window. Therefore, the *Max-FISM* algorithm consists of the following five steps.

- *Step* 1. Read and sort a transaction.
- *Step* 2. Insert new Max-itemset in the proposed summary data structure, called *Max-Set*.
- *Step* 3. Eliminate the effect of the oldest transaction in the current window from the *Max-Set*.
- *Step* 4. Find the maximal frequent itemsets from the current *Max-Set*.
- *Step* 5: Prune *Max-Set*.

In reality, a data stream can be too large to be fully sorted in memory, but it is always possible to sort the items in each transaction due to the limited number of items in transaction, and sorted transactions can be processed much more efficiently than the unsorted itemsets [16].

Steps 1 and 2 are performed in sequence for a new transaction. Step 3 is performed only in the window sliding phase. Step 4 is performed only when the up-to-date set of recent frequent itemsets is requested. Step 5 can take place periodically or when it is needed.

Since the reading of a basic window of transactions from the buffer in main-memory is straightforward, we shall henceforth focus on Steps 2 and 3 (discussed in Section 4.1), step 4 (discussed in Section 4.2), and step 5 (discussed in Section 4.3).

4.1. Effective construction and maintenance of Max-Set

We describe the method which constructs and maintains the proposed in-memory prefix-tree based data structure, called *Max-Set*. Before discussing the construction process for the *Max-Set*, we provide a brief description of its structure. A *Max-Set* consists of one root node referred to as "null", a set of nodes in the form (*itemset, I.Cnt*) representing Max-itemset of a transaction and the total number of its independent appearance in the current window, and a pointer pointing to the right most leaf in the *Max-Set*. In the *Max-Set*, the leaves point to each other from right to left order as shown in Fig. 1. The necessity of using these pointers will be discussed later in Section 4.2. Like a prefix-tree, each node in a *Max-Set* explicitly maintains parent, children, the itemset name and a counter to record the independent appearance count of the corresponding itemset in the current window.

We propose the novel concept of maintaining only the count of Max-itemset for a transaction, instead of maintaining all subsets of transaction in the tree. We call such a tree the *Max-Set*.

Based on the above discussion, a Max-Itemset can be defined as follows.

Definition 1. Let $T = \{A, C, D\}$ be a transaction in a data stream with items sorted according to a predefined sort order. The largest subset among the subsets of a transaction is called Max-itemset. For example, the itemset *ACD* is the Max-itemset of this transaction.

In the proposed method, initially, the *Max-Set* is empty, whenever a new transaction $T_i, i \in [1, m]$, is inserted in the current window W, only its max-itemset in a predefined item order should be inserted in the *Max-Set*. If Max-Itemset of the new transaction is inside the *Max-Set*, its counter is increased by 1. If it was not available in the *Max-Set*, the corresponding node is added to *Max-Set* and its counter is initialized at 1. For example, if itemset *ACD* first appears as Max-itemset induced by a transaction of the current window, its corresponding node should be created in the *Max-Set* and its counter should be initialized by 1 i.e., $I.Cnt_{ACD} = 1$.

A parent node in the *Max-Set* does not inherit the support count of its children. It only keeps the independent support value of the corresponding itemset in the current window.

Upon entrance of each transaction, each leaf node that is created in *Max-Set* should identify its previous and next leaf node in the *Max-Set*. For this purpose, the *Max-FISM* obtains a bit representation of the leaf itemset and then changes it to a decimal form. Note that itemsets with higher decimal numbers are arranged at the left side of the tree. As an example, consider the *Max-Set* in Fig. 1. We assign a decimal number for each of the leaf nodes as follows.

$$AB = 1100 = 12$$
$$ACD = 1011 = 11$$
$$BCD = 0111 = 7$$
$$BD = 0101 = 5$$
$$D = 0001 = 1$$
$$12 < 11 < 7 < 5 < 1.$$

The correct position of each newly arrived leaf node can be found by using the obtained order between decimal forms of leaf nodes. Then the relevant pointers should be updated.

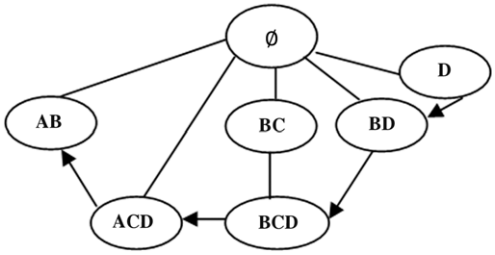


Fig. 1. Nodes of a *Max-Set*.

Algorithm 1 (*Max-Set* construction)
INPUT: (1) A transaction data stream $\boldsymbol{D} = [T_1, T_2, \ldots T_m)$ and (2) the user-specified sliding window size w.
OUTPUT: *A Max-Set tree for the current window: M.*
Method:
Begin
1. $w \leftarrow \emptyset$;
2. $M \leftarrow a\ Max - Set\ with\ null\ initialization$;
   /* **For the first window***/
3. **While** (W ≠ w) **do**
4. Call Insert_Transaction(T_i);
5. $w = w + 1$;
6. **End While**
   /* **At each slide of window***/
7. **Repeat**
8. Call Delete_Transaction(T_1);
9. Call Insert_Transaction(T_w);
10. **End**

End

Insert_Transaction()
Begin
1. Sort incoming transaction $T_i \in D$ in W
2. **If** $T_i \notin M$ **then**
3. Create a new entry of form (*itemset, I.Cnt*) into the M; /* *Insert the Max-Itemset of transaction* T_i *into Max-Set* */
4. **Else** T_i*. I.Cnt* = T_i*. I.Cnt t* + *1*; /**increment the count of max-Itemset* T_i*/
5. **End if**

End

Delete_Transaction ()
Begin
1. T_1*. I.Cnt* = T_1*. I.Cnt* – *1; /* Reduce the count of Max-Itemset in the oldest transaction* T_1 *in W from M* */
2. **If** T_1*. I.Cnt =0* **then**
3. Prune the corresponding node T_1 from M; /* *Delete the node having count zero from M**/
4. **End if**

End

Fig. 2. The *Max-Set* construction algorithm.

After sliding the current window in the window sliding phase, the *Max-Set* is updated by deducting the oldest transaction information and inserting a new transaction. Therefore, a newly generated transaction should be appended and the corresponding node of max-itemset in the *Max-Set* that is induced by the oldest transaction is decreased by 1. If its count is zero, that node will be pruned from *Max-Set*. Through the construction mechanism, we observe the following property of a *Max-Set*.

Property 1. The total frequency count of any node in a *Max-Set* is equal to the independent appearance count of the corresponding itemset in the current window.

Therefore, based on the *Max-Set* construction technique discussed above, the following lemma holds for a *Max-Set*.

Lemma 1. *Given a window size* w *on a data stream D, without considering the root node, the size of a Max-Set is bounded by* w.

Proof. Based on the *Max-Set* construction process, each transaction T_i, $i \in [1, m]$, in W contributes at one node *Max-Set*. Therefore, the total size contribution of all transactions in W is at the most w. However, because there are usually many common transactions in a window, the size of a *Max-Set* is normally much smaller than w. □

Lemma 1 highlights how compactly and completely the *Max-Set* captures the stream data in the current window. The compactness and completeness of the *Max-Set* enables mining of frequent itemsets without any loss, because there is no requirement for any approximation or error bound. Moreover, mining the *Max-Set* can be delayed until needed, because *Max-Set* constantly maintains a ready-to-mine platform at each window with exact recent information. Since in this method, from each transaction only *Max-Itemset* is considered, the processing time for each transaction is very low as compared to the case which all subsets should be monitored. Therefore, when transactions in data streams arrive in large volumes at an unanticipated rate, most transactions can be processed by using this method. So, this method increases the accuracy of results. Fig. 2 outlines the algorithm of *Max-Set* construction and maintenance in the *Max-FISM* algorithm.

In the next subsection, we focus on the technique used to discover Maximal frequent itemsets from the tree structure of the *Max-Set* during stream flow.

4.2. Determining maximal frequent itemsets from the Max-Set

Thus far, we have shown how a *Max-Set* can be constructed and maintain a ready-to-mine platform at each window, for the mining of exact frequent itemsets in the current window without any approximations or error bounds. The frequent itemsets selection phase is performed only when the mining result of the current window is requested. The *Max-FISM* can find the set of all currently frequent itemsets by upward and left to right traverse of the *Max-Set*.

In the *Max-Set* traverse step, to record the essential information about the previous itemsets, *Max-FISM* uses two list which are called *MFI list* and *MIFI list*.

Definition 2. *MFI list* contains all *Maximal Frequent Itemsets (MFI)* that have been found so far by traversing *Max-Set*.

Definition 3. MIFI List contains all **M***inimal* **I***n***F***requent* **I***temsets (MIFI)* that have been identified so far in this traversing.

Definition 4. **M***inimal* **I***n***F***requent* **I***temsets* are the set of itemsets that are not deemed frequent, but all their immediate subsets are frequent. They are denoted by *MIFIs*.

When *Max-FISM* reaches a node, *MFI list* and *MIFI list* should be searched before investigating the corresponding itemset.

MFI list should be searched because if the corresponding itemset or its superset is available in the *MFI list*, the itemset will be left without investigation. Then, *Max-FISM* considers the leaf node in the next branch of *Max-set*. In a prefix-tree based structure, a branch is represented by a path from root to a leaf. *MIFI List* also should be searched for the purpose that if the corresponding itemset or its subset is available in the list, the itemset will be left without investigation. Then *Max-FISM* considers the $(n-1)$-*subsets* of the *n-itemset*, that do not contain any *MIFI*. In this algorithm, most of the itemsets are considered as the subset of the previous itemsets in the *Max-Set* before *Max-FISM* reaches their position in the *Max-Set* traversing. Therefore, those itemsets that are specified as an infrequent itemset, should be kept in a list to avoid reconsideration.

If the subset of the intended itemset is not in the *MIFI list* and also its superset is not in the *MFI list*, the real count of the corresponding itemset in the current window should be obtained from total initial count and its supersets count in the *Max-Set*. Since this itemset has a node on *Max-Set*, its supersets are located in its child nodes and previous nodes in the Max-Set. Child nodes can be accessed more easily. In order to find superset nodes which are in the *Max-Set* before this node, only the leaf nodes should be searched. Because each node in the prefix tree based structure, as shown in Fig. 1, contains the parent itemset as well. Since the route between *Max-Set* leaf nodes is kept by using pointers, this search is done quickly.

So *Max-FISM* begins with child nodes and searches the leaves one by one respectively until it comes to the last leaf node in the left side of the *Max-Set*. In case a leaf node contains that itemset, *Max-FISM* continues in that path and goes up as the nodes containing that itemset are available and adds the count of them to the itemset count.

In this stage, the search space is very small, since most branches are not included in the search space because their leaf node does not contain the intended itemset. Furthermore, since in this method, from each transaction only the Max-Itemset is monitored, the number of nodes in each branch is not high and therefore, the supersets of an itemset will be found immediately.

Now *Max-FISM* has obtained the real count of the itemset that is total count of the itemset in the current window. If the real count of the itemset is more than the minimum support threshold s, it is added to the MFI list and the MFI list will be updated by removing its subsets. Then *Max-FISM* investigates the leaf node in the next branch. If it is not frequent, the $(n-1)$-*subsets* of *n-itemset* should be investigated on the *Max-Set*. Among its $(n-1)$-*subsets*, the position of which does not include the last item will be located in that branch and the position of the rest of $(n-1)$-*subsets* in the *Max-Set* will be after this branch. In order to obtain the real count of any of $(n-1)$-*subsets*, if $(n-1)$-*subsets* have no node on the *Max-Set*, the supersets search space is begun from leaf node that has higher decimal number from intended itemset to the left most leaf in the *Max-Set*.

In case none of $(n-1)$-*subsets* is frequent, $(n-1)$-*subsets* will be added to MIFI list by removing their supersets from that list. Then *Max-FISM* investigates the $(n-2)$-*subsets* of this *n-itemset*. This loop continues until a MFI is found in that path. Then *Max-FISM* considers the leaf node in the next branch. Fig. 3 shows the pseudo code of *Max-FISM*.

In this method, since most itemsets are considered earlier before *Max-FISM* reaches their position in the *Max-set* traversing, the whole computational load occurs in the first branches. Unless an itemset has only a few independent appearances in a window and is not among subsets of other itemsets, that itemset will not be considered until its position is traversed.

We use an example to illustrate the construction of the summary data structure *Max-Set* from stream data and demonstrate the major steps of *Max-FISM* in mining maximal frequent itemsets.

Example 1. Assume that the current window W contains eight transactions and the minimum support threshold s is 0.25. Fig. 4 shows W in a lexicographical item order with corresponding transaction *TIDs* and the *Max-Set* constructed after processing all the transactions of window W by *Max-FISM* algorithm. Note that each node of the form (*itemset: I.Cnt*) consists of two fields: itemset name and independent *support*. For example, (*ABC*: 4) indicates that, itemset *ABC* appeared independently four times in the current window. The Maximal frequent itemsets mining in the current window W, by *Max-FISM* algorithm is described as follows.

```
Algorithm 2(Max-FISM)
INPUT: (1) Max − set M that maintains entries e(itemset, I.Cnt) and (2) a user-defined minimum support threshold s ∈ [0, 1].
OUTPUT: The maximal frequent itemsets, MFI-list.
Method:
Begin
 1.  MFI-list = {};  /*initialize the MFI-list to empty*/
 2.  MIFI-list= {};  /*initialize theIMFI-list to empty*/
 3.  Call select_MFI(M);
End

Select_MFI()
Begin
 1.  v ← the left most leaf;
 2.  While (v ≠ ∅) do
 3.    Visit n-itemset e in node v;  /* visit the nodes by upward and left to right traverse of the Max-Set*/
 4.    If Superset of e ∉ MFI-list and subset of e ∉ MIFI-list then
 5.      Call supp_Calculate (e);  /* obtain real count of the itemset from total initial count and its supersets count in the Max-Set*/
 6.      If e. Supp ≥ s then
 7.        Insert e into MFI_list ;
 8.        Remove the subsets of e from MFI_list;
 9.      Else
 10.       Insert e into MIFI_list;  /* should be kept in a list to avoid reconsideration*/
 11.       Remove the supersets of e from MIFI_list;
 12.       Call subsets_investigate(e);  /*Consider the (n-1)-subsets of n-itemset, that do not contain any minimal infrequent itemset*/
 13.     Endif
 14.   Else if the subset of e ∈ MIFI-list then
 15.     For each the subset d from e  not including infrequent subset; /*Consider the subsets of n-itemset e, that do not contain any minimal infrequent itemset*/
 16.       Call supp_Calculate (d);
 17.       Go to 6;  /* to investigate itemset d
 18.   Endfor
 19. End if
 20. v ← the next right leaf;
 21. End while
End

Supp_Calculate ()
Begin
 1.  If e ∈ M then  /* the supersets are located in the child nodes and previous nodes in the Max-Set*/
 2.    e.Cnt = e.I.Cnt + count of all children;
 3.  endif
 4.  foreach previous leaf node do  /* visit the leaf nodes with higher decimal number*/
 5.    L ← previous leaf node; /* route between leaf nodes is kept by using pointers */
 6.    If L includes e then
 7.      e. Cnt = e. Cnt + L. I.Cnt;
 8.      foreach node n containing e in branch L do
 9.        e. Cnt = e. Cnt + n. I.Cnt;
 10.     Endfor
 11.   End if
 12. EndFor
 13. e. Supp = e.Cnt/w
End

Subsets_investigate()
Begin
 1.  foreach (n-1)-subset d of n-itemset e do
 2.    Call supp_Calculate (d)
 3.    If d. Supp ≥ s then
 4.      Insert d into MFI_list ;
 5.      Remove the subsets of d from MFI_list;
 6.    Else
 7.      Insert d into MIFI_list;  /* should be kept in a list to avoid reconsideration*/
 8.      Remove the supersets of d from MIFI_list;
 9.      Call subsets_investigate(d);  /* This loop continues recursively until an MFI is found in this branch */
 10.   Endif
 11. endfor
End
```

Fig. 3. The main framework of *Max-FISM*.

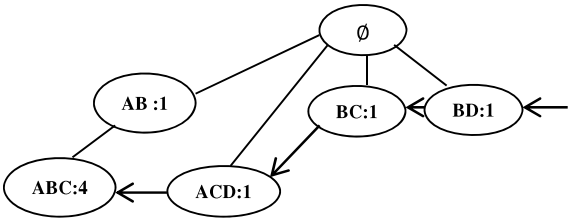
Upon a user request the *Max-FISM* algorithm runs the maximal frequent itemsets mining phase. First of all, the *Max-FISM* algorithm reads the left most leaf node, i.e., *ABC*.

ABC with the current count is frequent and therefore, *Max-FISM* inserts *ABC* into the *MFI-list* as shown in Fig. 4(c). Then *Max-FISM* goes to a leaf node in the next branch, i.e., *ACD*.

In order to obtain the real count of *ACD*, *Max-FISM* should obtain the count of its supersets. For this purpose, the algorithm should begin the search from children of *ACD*. *ACD* node is the leaf node and has no child. Then *Max-FISM* goes to the next leaf node, i.e., *ABC* which does not include *ACD* and is the last leaf node in the search space. Therefore, real count of *ACD* is 1 which is not frequent in the current window. Therefore, *ACD* is stored in the *MIFI-list* as shown in Fig. 4(c). Then *Max-FISM* considers its (2)-subsets, i.e., *AC, AD, CD*. Since the superset of *AC* is in the *MFI-list*, *Max-FISM* leaves it without further investigation. Then the algorithm considers *AD*. *AD* has no node on the *Max-Set*. In order to obtain its real count, the supersets search space is begun from the leaf node that has the higher decimal number from *AD* to the left most leaf in the *Max-Set*.

TID	Item set
1	B,D
2	A,B,C
3	A,B,C
4	B,C
5	A,B,C
6	A,C,D
7	A,B,C
8	A,B

a



b

MFI–LIST1 = {ABC}
MIFI–LIST1 = {ACD}
MIFI–LIST2 = {AD}
MFI–LIST2 = {ABC,D}
MIFI–LIST3 = {AD,CD,BD}

c

Fig. 4. A running example.

Therefore, *Max-FISM* obtains a bit representation of *AD*, i.e., 1001 and then changes it to a decimal form, i.e., 9. Note that leaf node with higher decimal numbers are in the search space, i.e., *ACD* and *ABC*. The result of this search is to inherit the count of *ACD*, i.e., 1. Therefore, *AD* is not frequent and is added to the *MIFI-list* and its superset, i.e., *ACD* is deleted from the *MIFI-list* as shown in Fig. 4(c). Then *Max-FISM* considers the (1)-subset of *AD*, i.e., *D*. *D* has no node on the *Max-set*. Therefore, *Max-FISM* obtains a bit representation of *D*, i.e., 0001 and then changes it to a decimal form, i.e., 1. Therefore, *Max-FISM* begins searching to find supersets from the leaf node with higher decimal number from *D* to the left most leaf in the *Max-Set*, i.e., *BD*, *BC*, *ACD* and *ABC* respectively. The result of this search is total count of *ACD* and *BD*, i.e., 2. Therefore *D* is frequent and is added to *MFI-list*. Then *Max-FISM* goes to *CD* node.

Since, its subset in the *MIFI-list* and its superset in the *MFI-list* are not available, it investigates *CD*.*CD* has no node on the *Max-Set*. So, *Max-FISM* obtains a bit representation of *CD*, i.e., 0011 and then a decimal form, i.e., 3. Therefore, *Max-FISM* begins searching to find supersets from the leaf node with higher decimal number from *CD* to the left most leaf in the *Max-Set*, i.e., *BD, BC*, *ACD* and *ABC* respectively. The result of this search is count *ACD*, i.e., 1. Therefore, *CD* is an infrequent itemset in the current window and is added to the *MIFI-list*. Since, *D* and the superset of *C* are available in the *MFI-list*, they are not investigated. Then *Max-FISM* goes to the leaf node in the next branch, i.e., *BC*. Since, the superset of *BC* in the *MFI-list* is available, it will be left without investigation. Then *Max-FISM* goes to the leaf node in the next branch, i.e., *BD*. Since, *BD* has no superset on the *Max-Set*, its real count is 1 which is not frequent. Therefore, *BD* is added to the *MIFI-list*. Since its (1)-subsets, i.e., *B* and *D* are both in the *MFI-list*, they are not considered.

At this stage, after processing all the nodes in the *Max-Set*, the *MFI list* generated by *Max-FISM* algorithm contains the set of current maximal frequent itemsets: {*ABC, D*} in the current window. Therefore, the set of all frequent itemsets can be generated by enumerating the set: {*ABC, D*}. The result is shown in Fig. 4(c).

In the next section, we describe the steps of pruning some frequent itemsets from *Max-Set*.

4.3. Pruning some frequent itemsets from the current Max-Set

The Max-Set may lose its property of compactness due to insertion of new transactions in the *Max-Set* construction phase. In the proposed method, in order to reduce the number of itemsets that should be maintained in the *Max-Set*, some itemsets can be pruned from the *Max-Set* periodically or when it is needed without any decrease in the results accuracy.

The proposed technique prunes the corresponding node of subsets of those MFIs which are valid for a definite period of time in the future. For this purpose, we define *Maximum Validity Time (MVT)* for MFIs.

Definition 5 (*Maximum Validity Time (MVT)*)**.** Given a frequent item set *e* in the current window *W*, *MVT(e)* is a measure to indicate how long the itemset *e* will remain as a frequent itemset without any additional occurrence in the future transactions of the window.

Algorithm 3(pruning from *Max-Set*)
INPUT: (1) Max – set M that maintans entries e(itemset, $I.Cnt$) and (2) a system-defined minimum MVT threshold Tr_{min}
OUTPUT: The reduced M, M'.
Method: */* it can run periodically or when it is needed*/*
Begin
1. **Foreach** maximal frequent itemset e ∈ M
2. **If**($MVT(e) \geq Tr_{min}$) **then**
3. *Eliminate all ascendant nodes of e from M;*
4. **Endif**
5. **Endfor**

End

Fig. 5. The pruning algorithm from *Max-Set*.

In other words, the value of *MVT(e)* is the identifier of the last future transaction that will regard the itemset e as a frequent itemset by assuming that the itemset e will not occur in the future transactions any more. We need to identify the boundary transaction in order to measure *MVT (e)*.

Definition 6 (*BOundary Transaction (BOT)*)**.** Given a frequent item set e in the current window W, from among the transactions which include itemset e in the current window W by assuming that the item set e will not occur in the future transactions any more, *BOT* is the oldest transaction that by exiting from this window, itemset e will no longer be frequent.

Therefore, during the *Max-Set* construction phase, we must record in the node containing each itemset the *TIDs* of all transactions whose max-itemsets in the current window are according to this itemset. The situation of *BOT* can be calculated by using Theorem 1.

Theorem 1. *Given an itemset e in the current window W, where* $Supp(e) \geq s$, $BOT\ (e)$ *is* $(Supp(e) - s) * w$ *transaction after the first transaction which includes the item set e.*

Proof. Assuming that the itemset e does not occur in the future transactions that insert into the window, the transactions including the itemset e exit from the window one by one by sliding the window. Therefore, this itemset will no longer be frequent. The number of these transactions can be calculated by using $(Supp(e) - s) * w$. Therefore, $BOT\ (e)$ can be obtained by having the number of transactions including the itemset e.

For the current window W, the $MVT\ (e)$ can be obtained by Theorem 2. □

Theorem 2. *Given a frequent item set e in the current window W, the MVT (e) can be obtained as follows:*

$$MVT(e) = BOT(e) - 1. \tag{1}$$

Proof. Suppose that subsequent transactions will not contain the itemset e. So, $n = (Supp(e) - s) * w$ means by exiting n transaction including itemset e from this window, the itemset e also will be frequent. But by existing $n + 1$ transaction including itemset e, it will no longer be frequent. *BOT(e)* denotes *TID* of transaction $n + 1$. Consequently, *MVT(e)* is obtained as follows:

$$MVT(e) = BOT(e) - 1. \quad \square$$

As proved in Theorem 2, $MVT(e)$ depends on two factors:

1. $Supp(e) - s$;
2. the situation of transactions containing the itemset e in the current window.

Fig. 5 shows the pseudo code of the pruning algorithm from *Max-Set*

Example 2. In Fig. 6, *ABC* has occurred in the transactions 2, 3, 5 and 7. If the minimum support threshold s is 0.25, according to the following relation, *BOT (ABC)* is the second transaction after the first transaction that includes this itemset, i.e., the fifth transaction.

$$Supp(ABC) - s = \left(\frac{4}{6} - \frac{2}{6}\right) * 6 = 2.$$

Therefore, *MVT (ABC)* is calculated by using the following formula:

$$MVT(ABC) = BOT(ABC) - 1 = 5 - 1 = 4.$$

If we assume the itemset *ABC* does not appear in the next transactions, we remove the first four transactions from the window, and then the itemset *ABC* will be frequent again. But upon deletion of the fifth transaction which is *BOT (ABC)*, the itemset *ABC* will become infrequent.

Therefore, to prune the subsets of a MFI from the *Max-Set*, its *MVT* should be greater or equal to the threshold defined by the system. This threshold is measured based on the average of the intervals between previous requests of user for MFIs.

TID	Itemset
1	B,D
2	A,B,C
3	A,B,C
4	B,C
5	A,B,C
6	A,C,D
7	A,B,C
8	A,B

Fig. 6. Current window.

For example, in a system, users may request new mining results every 100 transactions. On this basis, when pruning, the system calculates the threshold based on the number of remaining transactions until the user's next request.

This threshold can be updated by the system continuously since in a time interval, the system may face a large volume of user's requests. Therefore, the intervals between requests will become shorter and the threshold will come down. While the interval of the requests is short, the *Max-set* leads to inconsistent structures as the stream flows. Therefore, in this case when deleting the subsets of one MFI from the *Max-Set*, some information about the subsets must be recorded on the corresponding node of MFI. So that upon exiting *BOT* from the window, a part of its subsets can enter Max-Set with the frequency of MFI.

This method has no false negative error when most parts of the window are refreshed during further request of the user. This is because, if after announcement of previous results to the user, a MFI becomes infrequent immediately by exiting *BOT*, the system will have enough time to correct the results until the next request of user and the results will contain no false negative error. In this paper, we assumed that most part of the window is refreshed until further request.

This technique focuses on the point at which the user will be provided with the accurate mining results. It ensures that the *Max-Set* structure represent the exact content of the current window of stream and has the highest possible compactness. In the proposed method contrary to previous methods, memory consumption decreases as the *s* decreases.

5. Experimental results

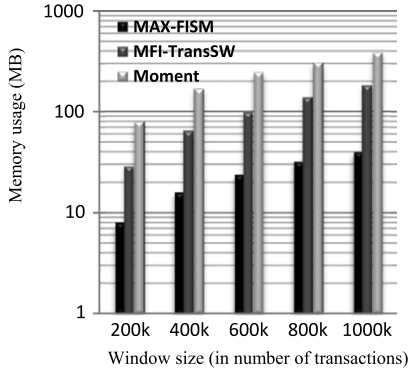
In this section, we report our experimental results of the performance analyses of the *Max-FISM* algorithm. To evaluate the performance of *Max-FISM* algorithm, we conduct empirical studies based on the IBM synthetic datasets [17]: T10.I5.D1000K, T30.I20.D1000K and real dataset BMS-POS [18]. The parameters of synthetic data generated by IBM synthetic data generator [2] denote the average transaction size (T), the average potentially maximal frequent item set size (I), and the total number of transactions (D), respectively. The first synthetic dataset T10.I5 is a *sparse* dataset. The second synthetic dataset T30.I20 is a *dense* dataset. The real dataset BMS-POS has average transaction size 6.53 items and the total number of transactions 515,597 transactions. It contains several years' worth of point-of-sale data from a large electronics retailer a *dense* dataset. In all experiments, the transactions of each data set are looked up one by one in sequence to simulate the continuous characteristic of an online data stream. A force-pruning operation is performed whenever 10,000 new transactions are generated only after the mining operation. All algorithms are implemented in java on a 3 GHz Intel Core 2 Duo PC with 2 GB main memory running Windows XP operating system.

In this experiment of study, we examine the two primary factors, *execution time* and *memory usage*, for mining frequent itemsets in a data stream environment, since both should be bounded online as time advances. To evaluate the performance of *Max-FISM*, we compare it with sliding-window based algorithms MFI-Trans-SW and Moment in terms of the memory and runtime requirement.

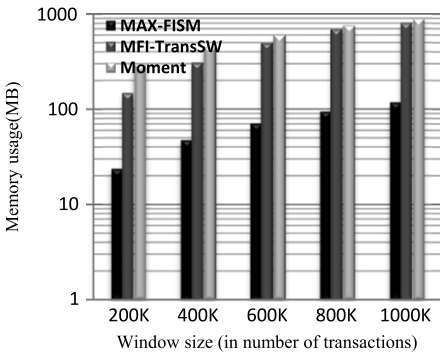
Experiment 1. In the first experiment, the performance of the *Max-FISM* method is compared with that of Moment, and MFI-TransSW in term of memory usage. Fig. 7 shows the results of the experiment in several datasets under different window sizes. The *x*-axes of the graphs represent the variation of window size in number of transactions for each dataset. While the *y*-axes show the memory usage. The large window-size is different for different data sets, because the number of available transactions in each data set is different.

For example, Fig. 7(a) illustrates the results on *T10I5D1000K* when the window size w grows from 200 K to 1000 K transactions, respectively. The memory usage for *T30I20D1000K* also is illustrated in Fig. 7(b) while that for the data set BMS-POS is shown in Fig. 7(c).

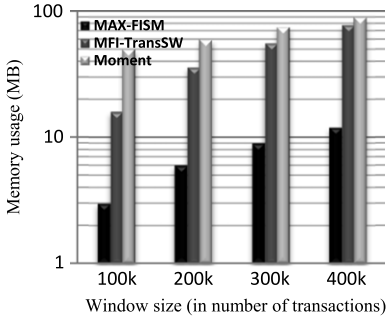
Since in Moment, MFI-TransSW and *Max-FISM*, one update consists of adding a new transaction to and deleting an old transaction from a sliding window, each sliding window differs from the previous one by exactly one transaction. That is,



(a) On dataset *T10I5D1000K*.



(b) On data set *T30I20D1000k*.



(c) On data set *BMS-POS*.

Fig. 7. Memory comparison.

for example when the sliding window size is 100,000, the first sliding window contains transactions 1–100,000; the second sliding window contains transactions 2–100,001, and so on. Therefore, to reduce variation, for each experiment we have executed over some consecutive sliding windows for each window size and reported the average performance over these sliding windows.

While comparing *Max-FISM* with Moment and MFI-TransSW, Fig. 7 demonstrates that for all window sizes *Max-FISM* requires less memory; the especially significant improvement in the dense datasets (e.g., *T30I20D1000K*), due to the high level of association among patterns in such datasets. For example, size improvements of *Max-FISM* in *T30I20D1000K* (Fig. 7(b)) for all window sizes are remarkable compared with other datasets.

The overall memory requirement of *Max-FISM* is reduced by maintaining only a few nodes (the only nodes keeping maximal itemset of each transaction in the data stream) to ensure search efficiency and a low memory usage. The number of items recorded in *MomentT* is larger than the number of closed itemsets in a given data stream and also much larger than its maximal itemsets. Therefore, it is understandable that *Max-FISM* consumes less memory than *Moment*.

For MFI-TransSW we calculate the memory for a window by accumulating the bit-sequence sizes of all items in the window and space to maintain other parameters such as item names and frequency count of each item. The bit sequence size for an item in a window is exactly the same as the number of transactions in the window. Therefore, *Max-FISM*'s memory gain over the MFI-TransSW is more prominent when the window size is larger.

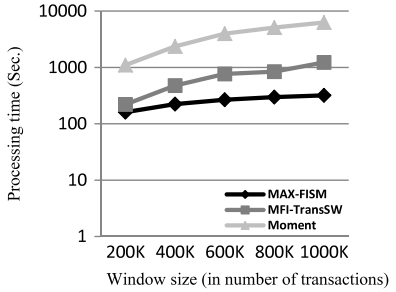
Therefore, Fig. 7 supports our claim that the physical memory requirement of *Max-FISM* for datasets with different characteristics is less than that of MFI-TransSW and Moment.

The experimental results presented in the next subsection highlight the runtime performance gain of *Max-FISM* due to its compact tree structure and efficient search mechanism.

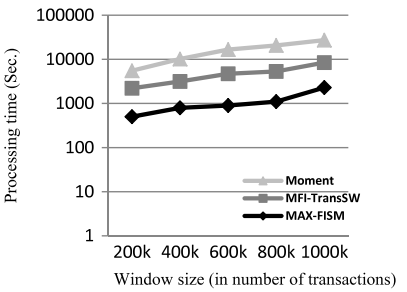
Experiment 2. In this experiment, we compare the execution time of *Max-FISM* with Moment and MFI-TransSW under different sliding window sizes in number of transactions over different datasets. We set the minimum support threshold s to 0.3%.

We also analysed *Max-FISM*'s performance by varying the min_sup values, as well over different datasets while the window size was kept fixed at reasonably high values (the results are provided in the next subsection).

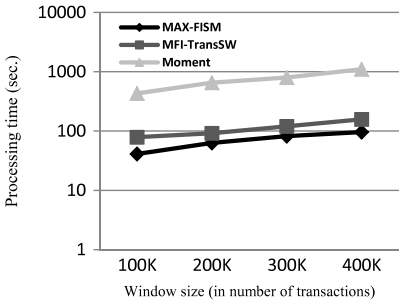
The time shown on the y-axis of each graph in Fig. 8 is the average total time required in all windows where the time includes for updating tree for Moment and *Max-FISM* and bit-sequence representation for MFI-TransSW (for adding each new transaction and deleting each old transaction), and that for mining frequent itemsets. While the x-axes show the variation in window size. Therefore, each graph illustrates the trends in execution time with the variation of window size. The experiments were performed with a mining request in each window.



(a) On dataset *T10I5D1000k*.



(b) On dataset *T30I20D1000k*.



(c) On dataset *BMS-POS*.

Fig. 8. Runtime comparison by varying the window size.

As shown in the graphs in Fig. 8, *Max-FISM*'s overall runtime for the above datasets does not change significantly with changes in the sliding window size. Mining time has a far greater impact on the total runtime than *Max-Set* updating time. The stable running time of *Max-FISM* comes from its efficient search and mining mechanism. The *Max-FISM* mining mechanism avoids traversing the whole *Max-Set* for the support estimation. One design consideration for *Max-FISM* is to maintain pointers between leaves in *Max-Set*. The overall runtime required by *Max-FISM* is therefore small enough to handle larger windows in different datasets. This result demonstrates an advantage of the *Max-FISM* algorithm that it is not sensitive to the sliding window size.

As shown in Fig. 8, for MFI-TransSW and Moment, in contrast, a considerable increase in the running time with increasing window size was observed. Therefore, in most cases, MFI-TransSW and Moment were not able to produce the frequent patterns within a reasonable amount of time when window size was large.

In dense datasets, the number and average size of frequent patterns may vary substantially for different window sizes.

Nevertheless, MFI-TransSW and Moment will require large amount of execution time to mine from dense dataset T30I20 with huge and long frequent patterns. Hence, the performance gain of *Max-FISM* over MFI-TransSW and Moment for dense dataset (Fig. 8(b)) was found much promising.

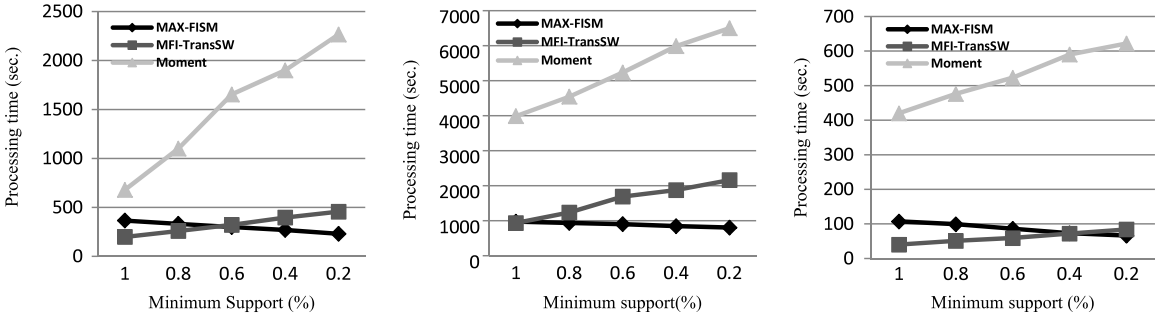
It shows that *Max-FISM* performs more efficiently than Moment and MFI-TransSW by multiple orders of magnitude.

Therefore, the results shown in Fig. 8 indicate clearly that *Max-FISM* can handle a variety of window sizes and produce the exact set of recent frequent patterns within a reasonable amount of time over datasets of various characteristics.

Experiment 3. Fig. 9 provides a runtime efficiency comparison between *Max-FISM*, Moment and MFI-TransSW using the same datasets based on changes in the min_sup value. We let the minimum support decrease from 1% to 0.2%.

As can be seen from the figure, as minimum support decreases, because the number of closed frequent itemsets increases, the running time for Moment algorithm grows. The running time also grow for Apriori-based MFI-TransSW with low minimum support thresholds. Such non-trivial cost of the approach is mainly dominated by its costly phase of handling a huge number of candidate sets [6]. As shown in Fig. 9, the running time of *Max-FISM* and MFI-TransSW are similar for higher min_sup values in the dense dataset T30I20D1000K. In the dense dataset, even at high minimum support, there are much frequent itemsets. However, the gap in running time between these algorithms increased when the min_sup value decreased. In the *Max-FISM* algorithm, when the minimum support decreases further, larger itemsets become frequent, therefore processing time of maximal frequent itemset mining phase is diminished.

Apriori-based MFI-TransSW [5] and Moment [4] required a long runtime by using low support values. However, increasing the minimum support value will negatively impact the mining results. The larger the minimum support value, the lower the number of generated patterns is, and as a result, the higher the risk of losing useful patterns. From this point of view, *Max-FISM* should be a better solution, compared to MFI-TransSW [5] and Moment in mining data streams with low support values.



(a) On dataset *T10I5D1000K*. (b) On dataset *T30I20D1000K*. (c) On dataset *BMS-POS*.

Fig. 9. Runtime comparison by varying the min_sup values.

These results show that *Max-FISM* outperforms Moment and MFI-TransSW in terms of both runtime and memory consumption in data streams of different characteristics, when used to mine an exact set of recent frequent patterns from a data stream.

6. Conclusion

In this paper, we propose an efficient single-pass algorithm, *Max-FISM*, to discover and maintain all Maximal frequent itemsets in a sliding window that contains the most recent transactions in a data stream. In the *Max-FISM* algorithm, an efficient in-memory data structure, *Max-Set*, is used to record max-itemsets of all transactions in the current sliding window. In addition, in the *Max-Set*, the leaves point to each other from right to left order. In the proposed method, in order to reduce the number of itemsets that should be maintained in the *Max-Set*, some itemsets can be pruned from the *Max-Set* periodically or when it is needed without any decrease in the results accuracy.

Experimental studies show that the running time of the *Max-FISM* algorithm is not sensitive to the sliding window size and under low minimum supports; processing time of maximal frequent itemset mining phase is diminished. In addition, when *Max-FISM* algorithm applied to dense data sets, *Max-Set*, has much less memory usage due to the high level of association among patterns in such datasets.

The Max-FISM algorithm outperforms existing algorithms, such as Moment and MFI-TransSW in terms of both runtime and memory consumption in data streams of different characteristics, when used to mine an exact set of frequent patterns from a high-speed data stream within a sliding window.

References

[1] M.N. Garofalakis, J. Gehrke, R. Rastogi, Querying and mining data streams: you only get one look a tutorial, in: Proc. 2002 ACM-SIGMOD Int. Conf. On Management-of Data, SIGMOD'02, Madison, WI, June 2002, p. 635.
[2] H.J. Woo, W.S. Lee, EstMax: tracing maximal frequent itemsets over online data streams, in: Proc. ICDM, 2007, pp. 709–714.
[3] Y. Zhu, D. Shasha, Statstream: statistical monitoring of thousands of data streams in real time, in: Proc. VLDB, 2002, pp. 358–369.
[4] Y. Chi, H. Wang, P.S. Yu, R.R. Muntz, Catch the moment: maintaining closed frequent itemsets over a data stream sliding window, in: Proceedings of Knowl. Inf. Syst. 2006, pp. 265–294.
[5] H. Li, S. Lee, Mining frequent itemsets over data streams using efficient window sliding techniques, presented at Expert Syst. Appl., 2009, pp. 1466–1477.
[6] R. Agrawal, T. Imielinski, A.N. Swami, Mining association rules between sets of items in large databases, in: Proc. SIGMOD Conference, 1993, pp. 207–216.
[7] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: Proc. ICDT, 1999, pp. 398–416.
[8] J. Pei, J. Han, R. Mao, CLOSET: an efficient algorithm for mining frequent closed itemsets, in: Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000, pp. 21–30.
[9] M.J. Zaki, C. Hsiao, CHARM: an efficient algorithm for closed itemset mining, in: Proc. SDM, 2002, pp. 457–473.
[10] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proc. SIGMOD Conference, 2000, pp. 1–12.
[11] H. Li, M. Shan, S. Lee, DSM-FI: an efficient algorithm for mining frequent itemsets in data streams, presented at Knowl. Inf. Syst., 2008, pp. 79–97.
[12] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proc. VLDB, 2002, pp. 346–357.
[13] J.X. Yu, Z. Chong, H. Lu, Z. Zhang, A. Zhou, A false negative approach to mining frequent itemsets from high speed transactional data streams, presented at Information Science, 2006, pp. 1986–2015.
[14] C. Lee, C. Lin, M. Chen, Sliding-window filtering: an efficient algorithm for incremental mining, in: Proc. CIKM, 2001, pp. 263–270.
[15] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, in: Proc. KDD, 2003, pp. 487–492.
[16] G. Mao, X. Wu, X. Zhu, G. Chen, Mining maximal frequent itemsets from data streams, J. Inform. Sci. 33 (3) (2007) 251–262.
[17] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proc. VLDB, 1994, pp. 487–499.
[18] Z. Zheng, R. Kohavi, L. Mason, Real world performance of association rule algorithms, in: Proc. KDD, 2001, pp. 401–406.