PAPER Special Section on Large Scale Algorithms for Learning and Optimization

An Efficient Algorithm for Sliding Window-Based Weighted Frequent Pattern Mining over Data Streams

Chowdhury Farhan AHMED[†a)], **Syed Khairuzzaman TANBEER**[†b)], ***Nonmembers***,
Byeong-Soo JEONG[†*c)], ***Member***, ***and*** **Young-Koo LEE**[†d)], ***Nonmember***

SUMMARY Traditional frequent pattern mining algorithms do not consider different semantic significances (weights) of the items. By considering different weights of the items, weighted frequent pattern (WFP) mining becomes an important research issue in data mining and knowledge discovery area. However, the existing state-of-the-art WFP mining algorithms consider all the data from the very beginning of a database to discover the resultant weighted frequent patterns. Therefore, their approaches may not be suitable for the large-scale data environment such as data streams where the volume of data is huge and unbounded. Moreover, they cannot extract the recent change of knowledge in a data stream adaptively by considering the old information which may not be interesting in the current time period. Another major limitation of the existing algorithms is to scan a database multiple times for finding the resultant weighted frequent patterns. In this paper, we propose a novel large-scale algorithm WFPMDS (Weighted Frequent Pattern Mining over Data Streams) for sliding window-based WFP mining over data streams. By using a single scan of data stream, the WFPMDS algorithm can discover important knowledge from the recent data elements. Extensive performance analyses show that our proposed algorithm is very efficient for sliding window-based WFP mining over data streams.

key words: *data mining, large-scale data, data streams, weighted frequent pattern mining*

1. Introduction

Data mining discovers hidden and potentially useful information from databases. Frequent pattern mining [1], [2], [9]–[12] plays an important role in data mining and knowledge discovery techniques such as association rule mining, classification, clustering, time-series mining, graph mining, web mining etc. A large number of research works have been done to find frequent patterns using Apriori-based algorithms [1], [2], FP-tree based algorithms [9], [10] and other algorithms [11], [17], [29]. However, those algorithms are working on the databases where each item has the same importance/weight and they find out only those patterns occurred frequently. Weighted frequent pattern mining [3]–[8], [28] was proposed to discover more important knowledge considering different weights of each item which plays an important role in the real world scenarios. For example, in a real world business database, frequency of gold ring is very low compared to the frequency of pen sold. As a result, knowledge about the patterns having low frequency but high weight remains hidden by finding only frequent patterns. The main contribution of the weighted frequent pattern mining is to retrieve this hidden knowledge from database.

Even though there have been a number of algorithms proposed for WFP mining, they consider all the data from the very beginning of a database. Therefore, they cannot be applied for mining the large-scale data such as data streams [13]–[16], [23]–[27], where data flows in the form of continuous stream. A data stream is a continuous, unbounded and ordered sequence of items that arrive in order of time. Due to this reason, it is impossible to maintain all the elements of a data stream. Moreover, the existing algorithms do not differentiate recently generated information from the old information which may be unimportant or obsolete in the current time period. As a result, they cannot extract the recent change of knowledge in a data stream adaptively.

Another major limitation of the existing algorithms is to scan a database multiple times for finding the resultant weighted frequent patterns. To find weighted frequent patterns from a data stream, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Therefore, single-pass and sliding window-based mechanism [16], [23]–[27] is required to find out the recent important knowledge from a data stream. In recent years, many applications generate data streams in real time, such as sensor data generated from sensor networks, transaction flows in retail chains, web click streams in web applications, performance measurement in network monitoring and traffic management, call records in telecommunications, and so on.

Motivated by these real world scenarios, in this paper, we propose a novel large-scale algorithm WFPMDS (Weighted Frequent Pattern Mining over Data Streams) for sliding window-based WFP mining over data streams. It can discover useful recent knowledge from a data stream by using a single scan. Our algorithm exploits a pattern growth mining approach to avoid the level-wise candidate generation-and-test problem. Besides retail market data, our algorithm can be well applied for the area of mining weighted web path traversal patterns. By considering different importance values for different websites, our algo-

Manuscript received December 5, 2008.
Manuscript revised March 4, 2009.

[†]The authors are with the Department of Computer Engineering, Kyung Hee University, 1 Sochun-ri, Kihung-eup, Youngin-si, Kyonggi-do, 446–701, South Korea.
[*]Corresponding author, Byeong-Soo JEONG
a) E-mail: farhan@khu.ac.kr
b) E-mail: tanbeer@khu.ac.kr
c) E-mail: jeong@khu.ac.kr
d) E-mail: yklee@khu.ac.kr

DOI: 10.1587/transinf.E92.D.1369

rithm can discover very useful knowledge about weighted frequent web path traversals in real time using only one scan of data stream. Moreover, it is also useful in the area of bio-medical and DNA data analysis as different biological gene has different importance, so, by detecting the combination of weighted gene patterns special gene patterns can be detected for a particular disease and drugs can be made based on that criterion using only one scan of data stream. Other application areas are telecommunication data, data feeds from sensor networks and stock market data analysis.

The remainder of this paper is organized as follows. In Sect. 2, we describe background. In Sect. 3, we develop our proposed WFPMDS algorithm for weighted frequent pattern mining over data streams. In Sect. 4, our experimental results are presented and analyzed. Finally, in Sect. 5, conclusions are drawn.

2. Background

2.1 Frequent Pattern Mining

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items, and let D be a transaction database, $\{T_1, T_2, \ldots, T_n\}$, where each transaction, $T_i \in D$, is a subset of I. The support/frequency of a pattern, $X\{i_p, \ldots, i_q\}$, (where $p \leq q$; $1 \leq p, q \leq m$ and $X \subseteq I$) is the number of transactions containing the pattern in the transaction database. The goal of frequent pattern mining is to find the complete set of patterns satisfying a minimum support in the transaction database. The *downward closure* property [1], [2] is used to prune the infrequent patterns. This property says that if a pattern is infrequent, then all of its super-patterns must be infrequent.

The *Apriori* [1], [2] algorithm is the initial solution of the frequent pattern mining problem, but it suffers from the level-wise candidate generation-and-test problem and requires several database scans. FP-growth [9] solves this problem by using an FP-tree-based solution without any candidate generation and using only two database scans. FP-array [10] technique was proposed to reduce the FP-tree traversals and it efficiently works especially in sparse datasets. One interesting measure h-confidence [18] was proposed to identify the strong support affinity frequent patterns. CP-tree [29] calculates all the frequent patterns using a single pass of database. There has been a significant amount of research into finding frequent patterns [11], [12], [17]. This traditional frequent pattern mining considers equal profit/weight for all items.

2.2 Weighted Frequent Pattern Mining

We have adopted definitions similar to those presented in the previous works [3]–[5].

Definition 1 (**Weight of a pattern**): The weight of an item is a non-negative real number which is assigned to reflect the importance of each item in the transaction database. For a set of items, $I = \{i_1, i_2, \ldots, i_n\}$, the weight of a pattern, $P\{x_1, x_2, \ldots, x_m\}$, is given as follows:

$$Weight(P) = \frac{\sum_{q=1}^{length(P)} Weight(x_q)}{length(P)} \quad (1)$$

Definition 2 (**Weighed support of a pattern**): A weighted support of a pattern is defined as the value that results from multiplying the pattern's support with the weight of the pattern. So the weighted support of a pattern, P, is given as follows:

$$Wsupport(P) = Weight(P) \times Support(P) \quad (2)$$

Definition 3 (**Weighed frequent pattern**): A pattern is called a weighted frequent pattern if the weighted support of the pattern is greater than or equal to the minimum threshold (δ).

Table 1 An example of retail database.

Item	Price ($)	Support (frequency)	Normalized Weight
Personal computer	800	500	0.8
Laser printer	450	320	0.45
Bubble jet printer	250	450	0.25
Digital Camera	600	700	0.6
Memory stick	200	825	0.2
Hard disk	130	350	0.13
DVD drive	100	450	0.1
CD drive	50	250	0.05

Table 1 shows an example of a retail database in which normalized weight values are assigned to items based on their prices. A normalization process is required to adjust the differences between data from various sources to create a common basis for comparison [3]–[5]. According to the normalization process, the final item weights can be determined to be within a specific weight range. For example, in Table 1 the weight values of the items are in the range from 0.05 to 0.8.

Some weighted frequent pattern mining algorithms (MINWAL [6], WARM [7], WAR [8]) have been developed based on the *Apriori* algorithm using the candidate generation-and-test paradigm. Obviously, these algorithms require multiple database scans and result in poor mining performance.

WFIM [3] is the first FP-tree-based weighted frequent pattern algorithm using two database scans over a static database. It makes use of a minimum weight and a weight range. Items are assigned fixed weights randomly from within the weight range. The FP-tree is arranged in weight ascending order and maintains the *downward closure* property. The WLPMINER [4] algorithm finds weighted frequent patterns using length decreasing support constraints. The WCloset [28] algorithm is proposed for the calculation of the closed weighted frequent patterns. The WIP [5] algorithm is proposed to discover weighted interesting patterns with a strong weight and/or support affinity.

The existing algorithms [3]–[5] show that the main challenge of weighted frequent pattern mining is that the weighted frequency of an itemset (or a pattern) does not

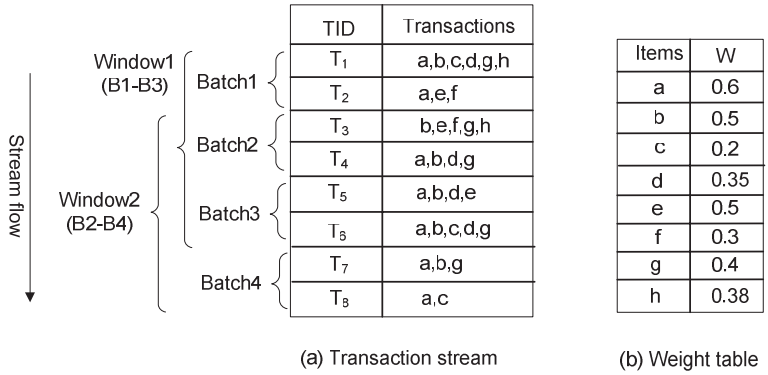

Fig. 1 Example of a transaction data stream with weight table.

have the *downward closure* property. Consider that item X has weight 0.6 and frequency 4, item Y has weight 0.2 and frequency 5, and itemset XY has frequency 3. According to Eq. (1), the weight of itemset XY will be $(0.6 + 0.2)/2 = 0.4$, and according to Eq. (2) its weighted frequency will be $0.4 \times 3 = 1.2$. The weighted frequency of X is $0.6 \times 4 = 2.4$, and of Y is $0.2 \times 5 = 1.0$. If the minimum threshold is 1.2, then pattern Y is weighted infrequent but XY is weighted frequent. As a result, the *downward closure* property is not satisfied here. WFIM and WIP maintain the *downward closure* property by multiplying each itemset's frequency by the maximum weight. In the above example, if item X has the maximum weight of 0.6, then by multiplying it with the frequency of item Y, 3.0 is obtained. So, pattern Y is not pruned at this early stage, and pattern XY will not be missed. At the final stage, this overestimated pattern Y will finally be pruned by using its actual weighted frequency.

Several single-pass mining algorithms [17], [29]–[31] have been developed in the context of traditional frequent pattern mining. Some other mining algorithms [13]–[15] have been developed to find out frequent patterns over a data stream. Sliding window-based algorithms [16], [23]–[27], [33] have also been developed for mining recent frequent patterns over a data stream. However, all these algorithms are not applicable for weighted frequent pattern mining.

The existing weighted frequent pattern mining methods consider all the transactions of a database from the very beginning and require at least two database scans. Hence, they are not suitable for stream data mining. Moreover, they cannot find important knowledge from the recent data. Therefore, we propose a sliding window-based novel algorithm for single-pass weighted frequent pattern mining in order to extract the recent change of knowledge in a data stream adaptively.

3. WFPMDS: Our Proposed Algorithm

3.1 Preliminaries

Transaction stream is a kind of data stream which occurs in retail chain or web click analysis. It may have infinite number of transactions. A batch of transactions contains a nonempty set of transactions. Figure 1 shows an example of transaction stream divided into four batches with equal length. A window consists of multiple batches. In our example, we assume that one window contains three batches of transactions. That is, *window1* contains *batch1*, *batch2* and *batch3*. Similarly *window2* contains *batch2*, *batch3* and *batch4*.

Definition 4: The support/frequency of a pattern P over a batch j is defined by the number of occurrences of that pattern in that batch and denoted as $Support_j(P)$. For example, $Support_3(ab) = 2$, i.e. the support of pattern "*ab*" in *batch3* is 2.

Definition 5: The weighted support of a pattern P over a window W, is defined by

$$Wsupport_W(P) = \left(\sum_{j=1}^{M} Support_j(P)\right) \times Weight(P) \qquad (3)$$

Here M is the number of batches in Window W. For example, the weighted support of pattern "*ab*" in *window2* is $Wsupport_2(ab) = (1+2+1) \times 0.55 = 4 \times 0.55 = 2.2$ in Fig. 1. The weight of the pattern "*ab*" $((0.6 + 0.5)/2 = 0.55)$ is calculated by using Eq. (1).

Definition 6: A pattern P is weighted frequent in window W if its weighted support in W is greater than or equal to the minimum weighted threshold (δ). For example, if the minimum weighted threshold is 2.0, pattern "*ab*" is a weighted frequent pattern in *window2* since its weighted support is 2.2 in *window2* (Fig. 1).

3.2 Tree Construction

In this section, we describe the construction process of our tree structure to capture stream data using a single pass. The header table is maintained to keep an item order in our tree structure. Each entry in a header table explicitly maintains item-id, frequency and weight information for each item. However, each node in a tree only maintains item-id and frequency information for each batch. To facilitate the tree traversals adjacent links are also maintained (not shown in

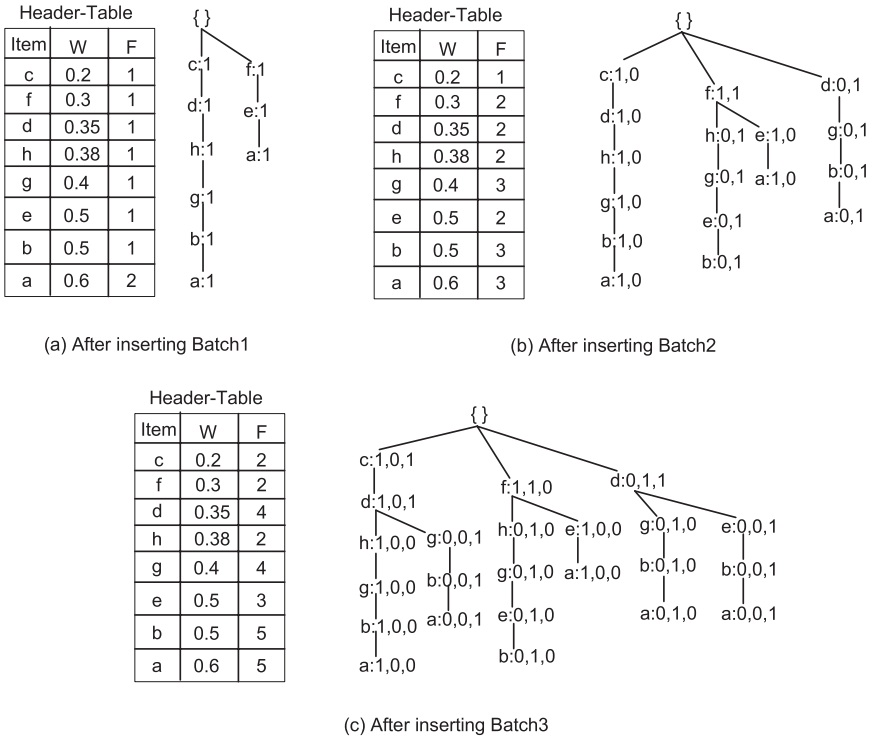

Fig. 2 Tree construction for *window1*.

the figures for simplicity) in our tree structure.

Consider the example data stream of Fig. 1 (a). At first we create the header table and keep all the items in weight ascending order. After that, we scan the transactions one by one, sort the items in a transaction according to header table item order and then insert into the tree. The first transaction T_1 has the items "*a*", "*b*", "*c*", "*d*", "*g*" and "*h*". After sorting, the new order will be "*c*", "*d*", "*h*", "*g*", "*b*" and "*a*". Figure 2 (a) shows the tree and the header table after inserting *batch1*. Figure 2 (b) shows the tree and the header table after inserting *batch2*. In the same way *batch3* is inserted into the tree. Figure 2 (c) shows the final tree for *window1*.

When the data stream moves to *batch4*, it is necessary to delete the information of *batch1* because *batch1* does not belong to *window2* and therefore the information of *batch1* becomes garbage in *window2*. We delete the information of *batch1* as shown in Fig. 3 (a). Some nodes which do not have any information for *batch2* and *batch3* can be deleted from the tree. In the case of other nodes, the frequency counters are shifted one position left to remove the frequency information of *batch1* and include the frequency information for *batch4*. As a result, now the three frequency information of each node represents *batch2*, *batch3* and *batch4*. Figure 3 (b) shows the tree after inserting *batch4*.

3.3 Mining Process

In this section, we describe the mining process of our proposed WFPMDS algorithm. As discussed in Sect. 2.2, we use the global maximum weight to utilize the *downward closure* property. The global maximum weight, denoted by *GMAXW*, is the maximum weight among all the items in the current window. For example, in Fig. 1 (b), item "*a*" has the global maximum weight of 0.6 for *window1* and *window2*.

Local maximum weight, denoted by *LMAXW*, is needed when we are doing the mining operation for a particular item. As the tree is sorted in weight ascending order, we can get the advantage of the bottom up mining operation. For example, after mining the weighted frequent patterns prefixing the item "*a*", when we go for mining operation prefixing the item "*b*", then the item "*a*" will never come in any prefix and conditional trees. As a result, now we can easily assume that the item "*b*" has the maximum weight. This type of maximum weight in mining process is known as *LMAXW*. For the mining operation prefixing item "*b*", *LMAXW* is equal to the weight of "*b*". As *LMAXW* is reducing from bottom to top, the probability of a pattern to be a candidate is also reduced. In the next section we will show the analysis regarding the measure of *LMAXW*.

Suppose we want to mine the recent weighted frequent patterns in the data stream presented at Fig. 1. It means we have to find out all the weighted frequent patterns in *window2*. Consider the minimum threshold = 1.8. Here the *GMAXW* = 0.6 and after multiplying the frequency of each item with *GMAXW*, the weighted frequency list is <*c*:1.2, *f*:0.6, *d*:1.8, *h*:0.6, *g*:2.4, *e*:1.2, *b*:3.0, *a*:3.0>. As a result, the candidate items are "*d*", "*g*", "*b*" and "*a*". Now we construct the prefix and conditional trees for these items in a bottom up fashion and mine the weighted frequent patterns in *window2*.

At first, the prefix tree of the bottom-most item "*a*" is created by taking all of the branches prefixing item "*a*"

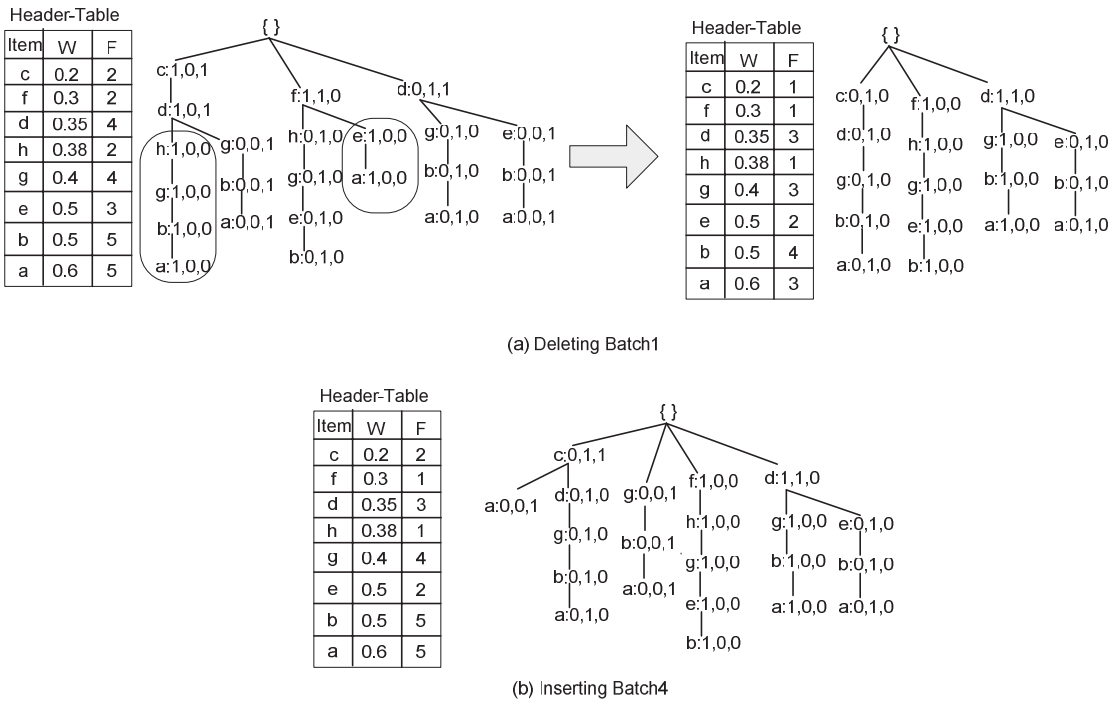

Fig. 3 Tree construction for *window2*.

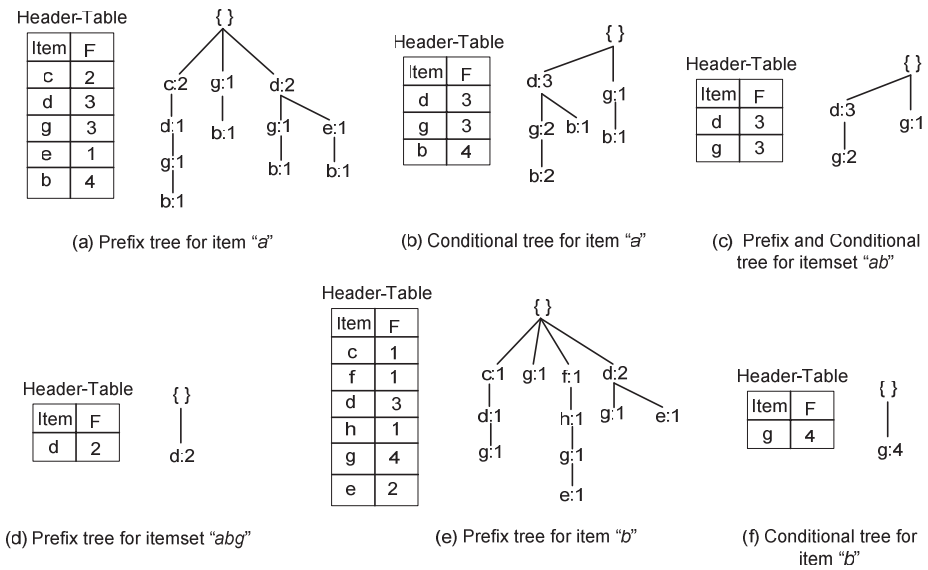

Fig. 4 Mining process.

shown in Fig. 4 (a). For the mining purpose, we add all the frequency values of a node in the prefix tree to indicate its total frequency value in this current window. For example, the first node of the leftmost branch of the prefix tree presented in Fig. 4 (a) stores "*c*:2" instead of "*c*:0, 1, 1".

To create the conditional tree for item "*a*", we have to delete the nodes from its prefix tree containing items which cannot be candidate patterns with it. For item "*a*", *LMAXW* = 0.6 and we can get its weighted frequency list by multiplying the other item's frequency with *LMAXW*. Obviously this weighted frequency is the maximum possible weighted frequency of an itemset prefixing item "*a*". So, we have to take all the patterns as a candidate having maximum weighted frequency greater than or equal to minimum threshold. As a result, the weighted frequency list for the item "*a*" is <*d*:1.8, *g*:1.8, *b*:2.4> (we should not consider global non-candidate items "*c*", "*f*", "*h*" and "*e*"). The weighted frequency list shows that items "*d*", "*g*" and "*b*" can be candidate patterns with item "*a*". Therefore, the conditional tree for item "*a*" is constructed from its prefix tree by deleting the nodes which do not contain items "*d*", "*g*" and "*b*" (shown in Fig. 4 (b)). After creating the conditional tree, candidate patterns "*ad*",

Table 2 *Wsupport* calculations of the candidate patterns for *window2*.

No.	Candidate Patterns	Wsupport (maximum)	Weight (actual)	Wsupport (actual)	Result
1	*ad*	0.6 × 3 = 1.8	((0.6+0.35)/2)= 0.475	0.475 × 3 = 1.425	Pruned
2	*ag*	0.6 × 3 = 1.8	((0.6+0.4)/2)= 0.5	0.5 × 3 = 1.5	Pruned
3	*ab*	0.6 × 4 = 2.4	((0.6+0.5)/2)= 0.55	0.55 × 4 = 2.2	Pass
4	*a*	0.6 × 5 = 3.0	0.6	0.6 × 5 = 3.0	Pass
5	*abd*	0.6 × 3 = 1.8	((0.6+0.5+0.35)/3)= 0.483	0.483 × 3 = 1.45	Pruned
6	*abg*	0.6 × 3 = 1.8	((0.6+0.5+0.4)/3)= 0.5	0.5 × 3 = 1.5	Pruned
7	*bg*	0.5 × 4 = 2.0	((0.5+0.4)/2)= 0.45	0.45 × 4 = 1.8	Pass
8	*b*	0.6 × 5 = 3.0	0.5	0.5 × 5 = 2.5	Pass
9	*g*	0.6 × 4 = 2.4	0.4	0.4 × 4 = 1.6	Pruned
10	*d*	0.6 × 3 = 1.8	0.35	0.35 × 4 = 1.4	Pruned

"*ag*", "*ab*" and "*a*" are generated.

The prefix-tree of itemset "*ab*" is created from the conditional tree of item "*a*" (shown in Fig. 4 (c)). Its weighted frequency list is <*d*:1.8, *g*:1.8>. As both the items "*d*" and "*g*" can be candidate patterns with itemset "*ab*", we cannot delete any node from this prefix tree. Therefore, the prefix tree of itemset "*ab*" is considered as its conditional tree and candidate patterns "*abd*" and "*abg*" are generated. Figure 4 (d) shows the prefix tree of itemset "*abg*" created from the conditional tree of itemset "*ab*". However, its weighted frequency list is <*d*:1.2> and not considered for further calculations.

For item "*b*" the *LMAXW* = 0.5 as the item "*a*" will not come out here. Figure 4 (e) shows the prefix tree for item "*b*" and its weighted frequency list is <*d*:1.5, *g*:2.0>. The key point is that, the maximum weighted frequency of item "*d*" with item "*b*" is 3 × 0.5 = 1.5, as *LMAXW* reduces from 0.6 to 0.5. Now without further calculation we can prune "*d*". But if *LMAXW* is 0.6 at this place, the weighted frequency of "*d*" is 3 × 0.6 = 1.8 and as a result it becomes a candidate. This is one big advantage of our algorithm.

The conditional tree of item "*b*" contains only one item "*g*" (shown in Fig. 4 (f)) and the candidate patterns "*bg*" and "*b*" are generated. For item "*g*", the *LMAXW* = 0.4 and its frequency value is 4. As a result, item "*g*" cannot form any candidate pattern with the remaining candidate item "*d*". Therefore, we do not have to create any prefix tree for the item "*g*" and we can stop the candidate generation process. We have to test all the candidate patterns with their actual weighted frequencies by using Eq. (3) and mine the actual weighted frequent patterns in *window2*. Table 2 shows that the actual weighted frequent patterns in *window2* are <*a*:3.0, *b*:2.5, *ab*:2.2, *bg*:1.8>.

3.4 Algorithm Description and Analysis

The pseudo-code of the WFPMDS algorithm is shown in Fig. 5. In this section, we describe the algorithm and analyze its complexity. In line 2, WFPMDS creates a global header table *H* to keep all the items in weight ascending order. In Sect. 3.1 and Fig. 1, it is explained that we have considered one window contains some batches. Moreover, we have used the sliding window-based approach. Therefore, after the first window, a new window is formed whenever a new batch of transactions arrives. For example, in Fig. 1 *window1* contains *batch1* to *batch3*. After that, *window2* is formed when *batch4* arrives. It consists of *batch2*, *batch3* and *batch4*. Hence, *batch1* has to be deleted from the tree before inserting *batch4*. In line 4, WFPMDS deletes this type of obsolete batches from the current window.

The "*for loop*" described in line 5 to line 9 inserts each transaction of the current batch into the tree. Our tree structure always maintains the order of its header table which is arranged in a fixed item order (weight ascending order of items). Accordingly, in line 6, WFPMDS arranges the items in a transaction in that order. Frequency values of the items are updated in line 7 and finally transaction is inserted into the tree in line 8. Our tree structure satisfies the following properties.

Property 1: The ordering of items is unaffected by the changes in frequency caused by incremental updating.

Proof: Let X and Y be two items with weight W_x and W_y respectively. Consider $W_x < W_y$ and their frequency values are F_x and F_y respectively. When a window slides from one to another, the values of F_x and F_y may vary based on the occurrences of the items in the new window. But it can not create any effect on W_x and W_y. As the items in the tree are ordered in their weight ascending order, X always comes before Y in any branch of the tree. □

Property 2: The total count of frequency values of any node in the tree is greater than or equal to the sum of total counts of frequency values of its children.

Proof: In the WFPMDS algorithm, every transaction is inserted according to the weight ascending order of items. Consider X is the parent node of Y in path P_1 of the tree and frequency value of X and Y are F_x and F_y respectively. The frequency value of Y is increased when a transaction contains items $\alpha XY\beta$. Here α and β are the set of other items before X and after Y respectively in P_1 (β can be $NULL$). Hence, F_x is incremented before F_y and F_y cannot be greater than F_x. On the other hand, if at least one transaction contains αX, then only F_x is incremented and F_x becomes larger than F_y. Therefore, $F_x \geq F_y$. □

Property 3: The tree structure can be constructed in a single scan of data stream.

```
Input: A transaction data stream, weight table, minimum threshold (δ), batch size, window size.
Output: Weighted frequent patterns for the current window.
begin
    Create the global header table H and place the items according to the weight ascending order
    foreach batch B_j do
        Delete obsolete batches for current window Win_i from the tree if required
        foreach transaction T_k in batch B_j do
            Sort the items inside T_k according to the weight ascending order
            Update frequency in the header table H
            Insert T_k in the tree
        end
        if any mining request from the user then
            Input δ from the user
            Let GMAXW be the maximum weight among all the items in the current window Win_i
            foreach each item α from the bottom of H do
                if frequency(α)× GMAXW ≥ δ then
                    Call Test_Candidate (α, frequency(α))
                    Create Prefix tree PT_α with its header table HT_α for item α
                    Call Mining ( PT_α, HT_α, α, Weight(α))
                end
            end
        end
    end
end
Procedure Mining(T, H, α, LMAXW)
begin
    foreach item β of H do
        if frequency(β)× LMAXW < δ then
            Delete β from H and T
        end
    end
    Let CT be the Conditional tree of α
    Let HC be the Header table of Conditional tree CT
    foreach item β of HC do
        Call Test_Candidate (αβ, frequency(αβ))
        Create Prefix-tree PT_αβ and Header table HT_αβ for pattern αβ
        Call Mining (PT_αβ , HT_αβ , αβ, LMAXW)
    end
end
Procedure Test_Candidate(X, F)
begin
    Let the actual weight of pattern X be W_X
    Set W_X = 0
    foreach item x_i in the pattern X do
        W_X = W_X + Weight(x_i)
    end
    W_X = W_X/Length(X)
    if F ×W_X ≥ δ then
        Add X in the weighted frequent pattern list
    end
end
```

Fig. 5 The WFPMDS algorithm.

Proof: In the WFPMDS algorithm, all items in a transaction are inserted into the tree according to weight ascending order. Moreover, frequency of every node is divided into separate counters to indicate individual batch frequency. Let the sliding window is moving from old window Win_{old} to new window Win_{new}. At this time, frequency value of every node can easily be updated for obsolete panes without rescanning the data stream and transactions of the new batches of Win_{new} are added by scanning them exactly one time. Hence, a single scan of data stream is required always. □

The above described properties are very useful in the WFPMDS algorithm. As WFPMDS is a stream data mining algorithm, Property 1 of the tree structure is very useful. Without this property we cannot guarantee that the order of item *x* and *y* will remain same in the next window and therefore tree restructuring operations may be required in each window. Property 2 is very essential for pattern growth mining operation [9]. As discussed in Sect. 1, we cannot scan stream data twice and therefore single-pass data capturing property (Property 3) is very essential for WFPMDS.

After updating the tree for the current batch, WFPMDS algorithm can mine weighted frequent patterns for the current window. It starts mining operation if there is any mining request from the user (line 10). In line 11, a minimum threshold δ is taken from the user. It calculates the global maximum weight (*GMAXW*) among all the items in line 12. The "*for loop*" described in line 13 to line 19 performs top-level of mining operation, i.e. it starts with each distinct item. This loop uses *GMAXW* to prune the non-candidate items (line 14). As discussed in Sect. 2.2, *downward closure* properly is not satisfied here. Therefore, single-element candidate patterns are tested finally by calling the Test_Candidate procedure (line 15). A prefix tree for a particular item is created in line 16 and the Mining procedure is called to generate candidate patterns prefixing that item in line 17.

The Mining procedure recursively mines the candidate patterns. It receives a pattern α, prefix tree T of α, header table H of prefix tree T and *LMAXW* of α. It creates the conditional tree CT of α and header table HC of CT by eliminating the non-candidate items from T and H respectively (line 25 to line 29). After that for each item β in HC, it creates a new candidate pattern $\alpha\beta$ by joining item β with pattern α and tests this candidate pattern by calling the Test_Candidate procedure (line 33). Finally, it creates the prefix tree $PT_{\alpha\beta}$ of pattern $\alpha\beta$, header table $HT_{\alpha\beta}$ for $PT_{\alpha\beta}$ and makes a recursive call to itself. As discussed in Sect. 3.3, for a particular distinct item x, *LMAXW* is always equal to the weight of x during the mining operation prefixing x. For example, for the mining operation prefixing item "*b*", *LMAXW*= 0.5 always. As the header table and tree is arranged in weight ascending order, no pattern prefixing item "*b*" can have larger weight than 0.5.

Lemma 1: If N_1 is the number of candidate patterns generated by using *LMAXW* and N_2 is the number of candidate patterns generated by using *GMAXW*, then $N_1 \leq N_2$.

Proof: Let a data stream has n items and their weight ascending sort-order is $i_1, i_2, \ldots, i_n$. If all these items have the same weight then always *GMAXW* = *LMAXW* during the mining operation. At this time, $N_1 = N_2$. Otherwise, *GMAXW* is equal to *LMAXW* only for the bottom-most item and in the case of other items *LMAXW* reduces from bottom to top while *GMAXW* remains fixed. Therefore, *LMAXW* can prune more patterns compared to *GMAXW*; hence, $N_1 \leq N_2$. □

Lemma 2: In the mining operation for prefixing item x, no pattern can have greater weight compared to *LMAXW*.

Proof: The WFPMDS algorithm arranges the header table and tree in weight ascending order of items. It also uses the bottom-up mining approach. Therefore, items in the upper portion of the tree cannot have greater weight compared to the items in the lower portion. Consider the weight of item x is W_x and *LMAXW* = W_x. Any other item, for example y, participating in the bottom-up mining operation for prefixing item x, can have maximum weight equal to W_x. Hence, according to Eq. (1) the maximum weight of pattern xy can be equal to *LMAXW*. □

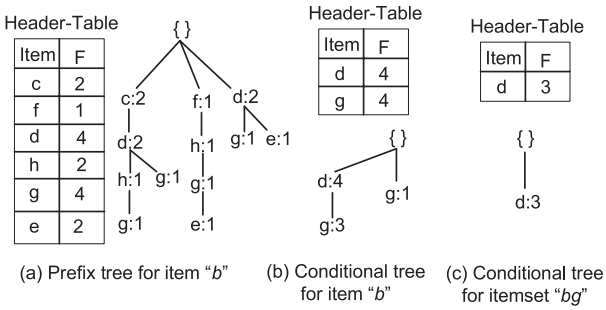

Fig. 6 Mining process for item "*b*" in *window*1.

Lemma 1 shows that *LMAXW* is more useful measure compared to *GMAXW* for pruning candidate patterns during the mining process. Moreover, Lemma 2 shows that no pattern can be erroneously pruned by using *LMAXW* as a maximum weight during mining operation prefixing a particular item. However, Observation 1 shows that actual weighted frequent patterns may be pruned erroneously by using the measures of *MinW* and *MaxW* adopted by the existing algorithms.

Observation 1: The existing algorithm WFIM uses *MinW* in each step of mining operation. It represents the minimum weight of a conditional pattern. When we perform the mining operation prefixing a pattern (for example "*ab*"), it is called a conditional pattern. Consider *window1* in the example data stream of Fig. 1 and tree constructed for this window in Fig. 2 (c). Let minimum threshold (δ) = 1.24. Now we perform the mining operation prefixing item "*b*". The prefix tree of item "*b*" is shown in Fig. 6 (a). The conditional database for prefix "*b*" (conditional pattern "*b*") contains {<*gd*:3>, <*g*:1>, <*d*:1>}. The existing WCloset algorithm uses *MaxW* (maximum weight of items within a conditional database) in each mining step. Hence, *MaxW* of the conditional database for prefix "*b*" is 0.4. However, "*bg*" and "*bd*" are actual weighted frequent patterns in *window1* with actual weight of 1.8 (0.45×4) and 1.7 (0.425×4) respectively. Even though *MaxW*(0.4) can discover these patterns for δ = 1.24, it cannot discover them or their super-patterns (if any) for δ = 1.65 (as 0.4 × 4 = 1.6). However, *MinW*(0.5) and *LMAXW*(0.5) measures can successfully mine these patterns (as 0.5 × 4 = 2.0). The conditional tree of item "*b*" is shown in Fig. 6 (b). For conditional pattern "*bg*", *MinW* = 0.4 and *LMAXW* = 0.5. The conditional database for prefix "*bg*" contains {<*d*:3>} and therefore *MaxW* = 0.38. As a result, both *MaxW* and *MinW* measures prune item "*d*" for δ = 1.24 (as 0.38 × 3 = 1.14 and 0.4 × 3 = 1.2) and miss pattern "*bgd*" and all its super-patterns (if any) as a consequence. Note that "*bgd*" is an actual weighted frequent pattern in *window1* with actual weight of 1.25 (0.41666 × 3). On the other hand, *LMAXW* can successfully discover this pattern (as 0.5 × 3 = 1.5). The conditional tree of pattern "*bg*" is shown in Fig. 6 (c). In summary, the measures of *MaxW* and *MinW* may erroneously prune some actual

weighted frequent patterns while *LMAXW* can successfully discover all of them.

The Test_Candidate procedure receives a candidate pattern X and its frequency in a particular window. In line 41 to line 45 it calculates the actual weight of X. In line 46, it calculates the actual weighted frequency of X and compares with the minimum threshold δ. If the actual weighted frequency of X is greater than or equal to the minimum threshold, it is stored in the actual weighted frequent pattern list.

4. Experimental Results and Analysis

4.1 Experimental Environment and Datasets

To evaluate the performance of our proposed algorithm, we have performed several experiments on IBM synthetic dataset (*T10I4D100K*) [19], real life datasets (*BMS-WebView-1, BMS-WebView-2, BMS-POS, kosarak*) [19], [20], [22] using synthetic weights, and a real dataset (*Chainstore*) [21] using real weight values. Our programs were written in Microsoft Visual C++ 6.0, and run with the Windows XP operating system on a Pentium dual core 2.13 GHz CPU with 1 GB main memory.

The IBM synthetic dataset *T10I4D100K* and real life datasets *BMS-WebView-1, BMS-WebView-2, BMS-POS* and *kosarak* do not provide the weight values of each item. Most of the previous weight-based frequent pattern mining research [3]–[5], [7], [8], [28] generated random numbers for the weight values of each item, but when observing real world datasets, most items are in the low weight range. Therefore, the weight value of each item was heuristically chosen to be between 0.1 and 0.9, and randomly generated using a log-normal distribution. Figure 7 shows the weight distribution of 2000 distinct items using the log-normal distribution.

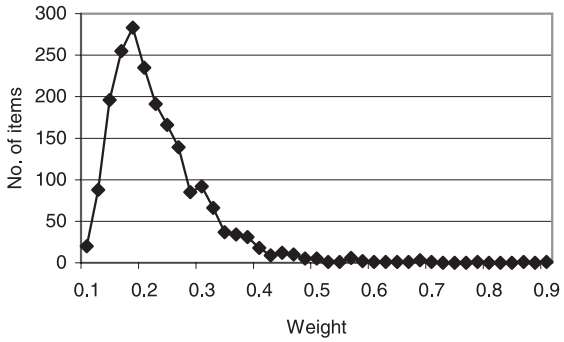

Fig. 7 Weight generation for 2000 distinct items using lognormal distribution.

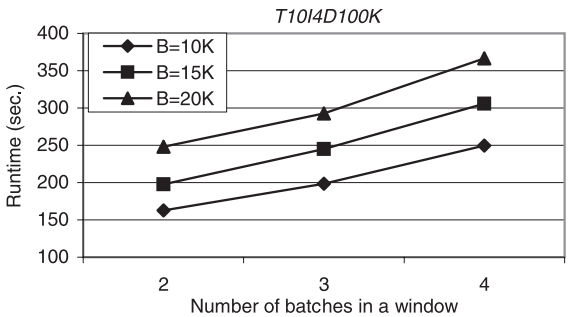

Fig. 8 Effect of window size variation on the *T10I4D100K* dataset (δ = 2%).

4.2 Performance Analysis of the WFPMDS Algorithm

4.2.1 Effect of Window Size Variation

Our proposed algorithm WFPMDS performs window-based stream data mining. It always keeps the current window information in the tree structure and mines the weighted frequent patterns in the current window by using the information kept in the tree structure. Hence, its runtime and memory requirement are dependent on the window size. A window consists of M batches and a batch consists of N transactions. Therefore, window size may vary depending on the number of batches in a window and the number of transactions in a batch. In this section, we analyze the performance of WFPMDS by varying both of these two parameters over *T10I4D100K* and *BMS-POS* datasets. We will use W and B to represent the window size and batch size respectively.

The *T10I4D100K* dataset was developed by the IBM Almaden Quest research group and obtained from the frequent itemset mining dataset repository [19]. This dataset contains 100,000 transactions and 870 distinct items. Its average transaction size is 10.1. Figure 8 shows the effect of different window sizes changing the number of batches in a window and the number of transactions in a batch. The x-axis of Fig. 8 shows different window sizes containing 2, 3 and 4 batches. Three curves show the effects of three different sizes of batches. The mining operation is performed at each window with a minimum threshold (δ) value of 2%. The y-axis of Fig. 8 shows the total runtime (tree construction, tree update and mining) of WFPMDS.

The *BMS-POS* dataset [19], [22] contains several years worth of point-of-sale data from a large electronics retailer. Since this retailer has so many different products, product categories are used as items. The transaction in this dataset is a customer's purchase transaction consisting of all the product categories purchased at one time [22]. The goal for this dataset is to find associations between product categories purchased by customers in a single visit to the retailer. It contains 515,597 transactions and 1,657 distinct items. Its average transaction size is 6.53. Our algorithm can efficiently mine this dataset by using a single-pass and sliding-window based mechanism. Figure 9 shows the effect of different window sizes changing the number of batches in a window and the number of transactions in a batch. The x-axis shows different window sizes containing 2, 3, 4 and 5 batches. Four curves show the effects of four different sizes of batches. The mining operation is performed at each window with a minimum threshold (δ) value of 3%. The y-axis shows the total runtime of WFPMDS.

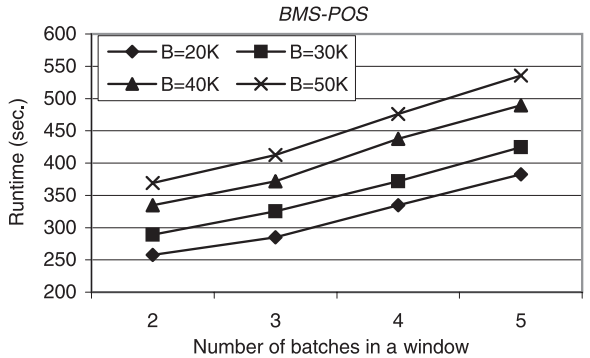

Fig. 9 Effect of window size variation on the *BMS-POS* dataset (δ = 3%).

The memory usage of WFPMDS depends on the window size. WFPMDS needs to keep more information when the number of batches in a window is increased and/or the number of transactions in a batch is increased. For the *T10I4D100K* dataset, the maximum size of the constructed tree for a window of 2 batches (B=20 K) is 4.34 MB. The tree size linearly increases when the number of batches is increased. It takes 6.482 MB and 8.63 MB memory for W=3B and W=4B respectively. The similar results are obtained from the *BMS-POS* dataset. The maximum sizes of the constructed trees for a window of 2, 3, 4 and 5 batches (B=50K) are 7.82 MB, 11.258 MB, 14.631 MB and 18.352 MB respectively.

4.2.2 Effect of Employing *LMAXW*

Our proposed algorithm WFPMDS can efficiently reduce the number of candidate patterns by using the *LMAXW* measure instead of *GMAXW* in the bottom-up mining process. Section 3.3 and Sect. 3.4 show the example and analysis to explain the effectiveness of *LMAXW*. In this section, we show the effect of employing *LMAXW* over *BMS-WebView-1* and *BMS-WebView-2* datasets.

The *BMS-WebView-1* and *BMS-WebView-2* [19], [22] datasets contain several months worth of click-stream data from two e-commerce web sites. Each transaction in these datasets is a web session consisting of all the product detail pages viewed in that session. That is, each product detail view is an item. The goal for both of these datasets is to find associations between products viewed by visitors in a single visit to the web site [22]. These two datasets contain 59,602 and 77,512 transactions respectively (with 497 and 3,340 distinct items). The average transaction sizes of them are 2.5 and 5.0 respectively. Single-pass and sliding window-based algorithms are needed for mining these types of web click-stream datasets in real time.

The effect of employing *LMAXW* depends on the distribution of weights of the items. If the variation of weights is not significant, effect of employing *LMAXW* instead of *GMAXW* is also not significant. However, as discussed in Sect. 4.1, in the real world datasets this variation is significant. We tested the WFPMDS algorithm (W=3B, B=10K) over these datasets by using both of *LMAXW* and *GMAXW* measures. Table 3 represents the comparison of number of candidates with different minimum thresholds. As shown in Table 3, the smaller the minimum threshold the larger the number of candidate patterns. Moreover, the difference between the number of candidate patterns generated by *GMAXW* and *LMAXW* also increases when the minimum threshold decreases. Due to this difference of number of candidate patterns, the runtime also varies significantly. Figure 10 and Fig. 11 show the runtime performance curves for *BMS-WebView-1* and *BMS-WebView-2* datasets respectively.

Table 3 Number of candidate comparison.

Dataset	Minimum threshold(%)	*LMAXW*	*GMAXW*
BMS-WebView-1	0.1	6124	11237
	0.3	3821	7231
	0.5	2178	4156
	0.7	874	1335
	0.9	356	479
BMS-WebView-2	0.1	28567	76784
	0.3	12891	35282
	0.5	5289	16781
	0.7	2567	7569
	0.9	862	3781

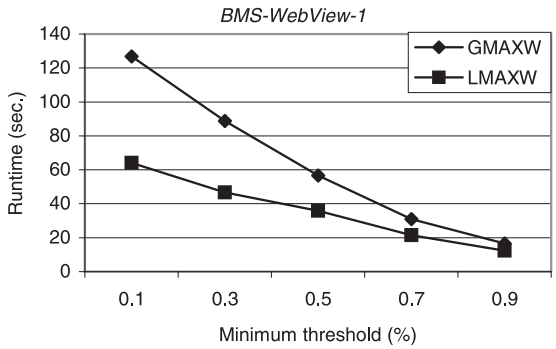

Fig. 10 Effect of employing *LMAXW* on the *BMS-WebView-1* dataset.

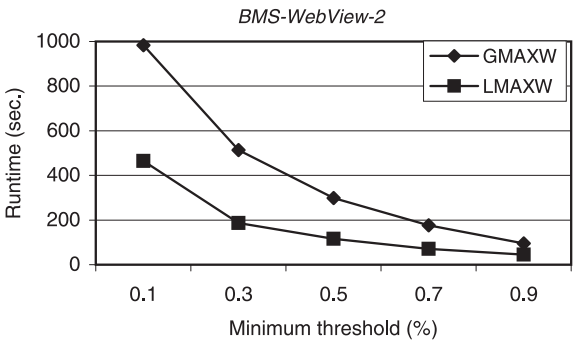

Fig. 11 Effect of employing *LMAXW* on the *BMS-WebView-2* dataset.

4.2.3 Runtime Distribution

In this section we show the runtime distribution of the WFPMDS algorithm. For the first window, WFPMDS constructs the tree by inserting the transactions for the batches of this

Table 4 Runtime distribution (sec.).

Dataset(batch and window size) (minimum threshold)	Tree construction time	Tree update time	Mining time		Total time	
			δ_1	δ_2	δ_1	δ_2
BMS-WebView-1(B=10 K,W=3 B) δ_1 = 0.9%, δ_2 = 0.3%	8.56	0.672	3.14	37.439	12.372	46.671
BMS-WebView-2(B=10 K,W=3 B) δ_1 = 0.9%, δ_2 = 0.3%	28.549	1.87	15.204	156.41	45.623	186.829
T10I4D100K(B=15 K,W=2 B) δ_1 = 5%, δ_2 = 2%	46.281	1.53	5.82	149.905	53.631	197.716
BMS-POS(B=30 K,W=4 B) δ_1 = 7%, δ_2 = 3%	173.58	3.641	12.076	198.15	189.297	375.371

window. For other windows, WFPMDS deletes the transactions of the obsolete batches and then inserts the transactions of the new batches. It also performs mining operation based on user given minimum thresholds. It is shown in Table 4 that the runtime of WFPMDS is divided into three parts. The tree construction time is the total time needed to insert all the new batches inside the tree. Similarly, the tree update time is the total time needed to delete all the obsolete batches inside the tree for each window. The mining time is the addition of all the mining time in each window. As WFPMDS captures all the transactions in the tree structure, its tree construction and tree update time do not vary for a particular window while mining operations are performed for different minimum thresholds. However, the overall runtime varies due to the variation of mining time for different minimum thresholds. Table 4 reports runtime distribution for two minimum threshold values (one high and one low) with different datasets.

4.3 Comparison with the Existing Algorithms

The existing algorithms are not suitable for stream data mining due to scanning a database at least twice. Moreover, they cannot keep batch by batch information in the tree for sliding window-based stream data mining. The existing WFIM needs two database scans to mine the weighted frequent patterns. In the first scan it finds all single-element candidate patterns and in the second scan it performs the tree creation and mining operation. The existing WCloset algorithm also needs two database scans to discover all the closed weighted frequent patterns. As it finds only closed patterns, their resultant patterns are not similar compared to that of WFPMDS. Extra computations will be required for WCloset to mine all the weighted frequent patterns from closed weighted frequent patterns. Therefore, in this section we have compared the performance of WFPMDS with WFIM.

As the WFIM algorithm is not a sliding window-based weighted frequent pattern mining algorithm over data stream, multiple executions (i.e. one execution in each window) are needed in order to compare it with WFPMDS. WFIM needs two database scans to mine the resultant patterns in the current window, for example *window1*, according to a user given threshold. Their tree structure is designed to represent information with respect to a particular user given minimum threshold. As a consequence, their tree structure cannot be used when the window slides from *window1* to *window2*. Hence, to mine the results in *window2* it has to be started from the very beginning, i.e. WFIM needs to build its tree structure by scanning *window2* twice.

On the other hand, our WFPMDS performs all the operations by using a single scan of data stream. It keeps separate information for each batch inside the tree nodes. Hence, when the current window slides to a new window it can easily discard the information of obsolete batches from the tree nodes to update the tree. After that, it inserts the information of the new batches into the tree. Therefore, it does not need to traverse any batch twice. Due to these reasons, it outperforms the existing WFIM algorithm in sliding window-based stream data mining with respect to execution time.

As discussed in Sect. 4.2.3, tree construction and tree update time of WFPMDS do not vary for a particular window while mining operations are performed for different minimum thresholds. In contrast, tree construction time of WFIM varies for different minimum thresholds as it keeps candidate items from each transaction into the tree with respect to a particular minimum threshold. Therefore, in sliding window-based stream data mining, the overall runtime difference between WFPMDS and WFIM increases when the minimum threshold decreases.

At first, we compare our algorithm with the existing WFIM algorithm by using the *kosarak* dataset. The dataset *kosarak* was provided by Ferenc Bodon and contains click-stream data of a Hungarian on-line news portal [19]. It contains 990,002 transactions and 41,270 distinct items. Its average transaction size is 8.1. Figure 12 shows the runtime comparison between WFIM and WFPMDS algorithms in this dataset. Window size W=4B, batch size B=50 K and minimum threshold range of 2% to 6% are used in Fig. 12. This figure shows that WFPMDS outperforms WFIM significantly with respect to execution time.

We have used a real-life dataset adopted from NU-MineBench 2.0, a powerful benchmark suite consisting of multiple data mining applications and databases [21]. This dataset called *Chain-store* was taken from a major chain in California and contains 1,112,949 transactions and 46,086 distinct items [21]. Its average transaction size is 7.2. We have taken real weight values for items from their profit table. Figure 13 shows the runtime comparison between the existing WFIM algorithm and our proposed WFPMDS algorithm in this dataset. Window size W=4B, batch size

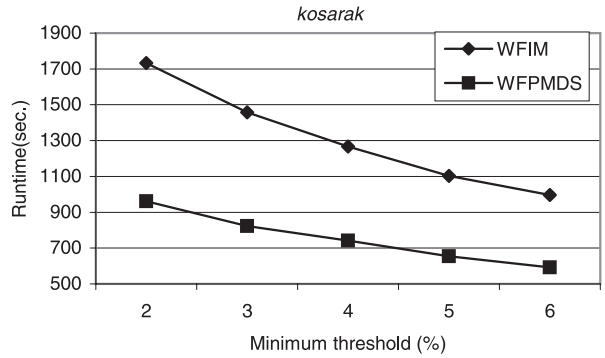

Fig. 12 Runtime comparison on the *kosarak* dataset.

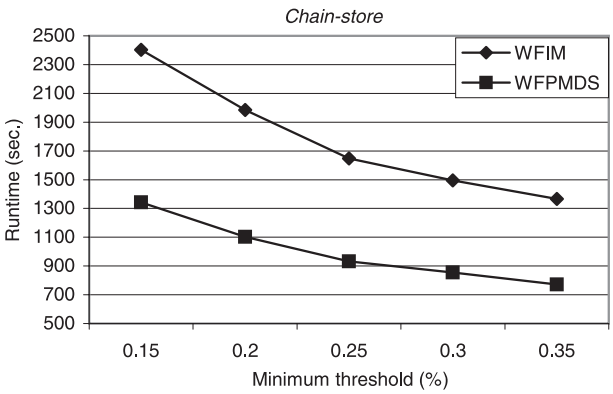

Fig. 13 Runtime comparison on the *Chain-store* dataset.

B=50 K and minimum threshold range of 0.15% to 0.35% are used in Fig. 13. This figure also shows that WFPMDS outperforms WFIM significantly with respect to execution time in stream data mining.

WFPMDS keeps all the transactions in a batch by batch fashion into the tree to achieve the single-pass sliding window-based stream data mining. In contrast, the WFIM algorithm is designed for static databases and therefore it keeps only candidate items from each transaction into the tree with respect to a particular user given minimum threshold. As a consequence, WFIM needs less memory compared to WFPMDS. However, with modern technology, main memory space is no longer a big concern [16], [17], [30], [31]. Research into prefix-tree-based frequent pattern mining [16], [17], [29]–[33] has shown that the memory requirement for the prefix trees is low enough to use the gigabyte-range memory now available. In the recent years, reducing the execution time is a major challenging issue of research. Therefore, in this paper, our main goal is to achieve a faster algorithm for stream data mining. Moreover, in Sect. 4.2.1, we have shown that our tree structure can be efficiently kept within this memory range. Hence, WFPMDS can be efficiently applied for the sliding window-based weighted frequent pattern mining over data streams using the recently available gigabyte-range memory.

5. Conclusions

The main contribution of this paper is to provide a novel large-scale algorithm for sliding window-based weighted frequent pattern mining over data streams. Our proposed algorithm WFPMDS can capture the recent change of knowledge in a data stream adaptively by using a novel tree structure. It requires only a single-pass of data stream for tree construction and mining operations. Therefore, it is quite suitable for data stream applications which need to discover valuable recent knowledge. Moreover, since our algorithm exploits a pattern growth mining approach we can easily avoid the problem of the level-wise candidate generation-and-test approach. It also saves a huge amount of memory space by keeping the recent information very efficiently in a tree structure. Extensive performance analyses show that our algorithm is very efficient for sliding window-based weighted frequent pattern mining over data streams.

Acknowledgements

We would like to express our deep gratitude to the anonymous reviewers of this paper. Their useful comments have played a significant role in improving the quality of this work.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," Proc. 12th ACM SIGMOD Int. Conf. on Management of Data, pp.207–216, May 1993.
[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," Proc. 20th Int. Conf. on Very Large Data Bases, pp.487–499, Sept. 1994.
[3] U. Yun and J.J. Leggett, "WFIM: Weighted frequent itemset mining with a weight range and a minimum weight," Proc. Fifth SIAM Int. Conf. on Data Mining, pp.636–640, USA, 2005.
[4] U. Yun and J.J. Leggett, "WLPMiner: Weighted frequent pattern mining with length decreasing support constraints," Proc. 9th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD), pp.555–567, Vietnam, 2005.
[5] U. Yun, "Efficient mining of weighted interesting patterns with a strong weight and/or support affinity," Inf. Sci., vol.177, pp.3477–3499, 2007.
[6] C.H. Cai, A.W. Fu, C.H. Cheng, and W.W. Kwong, "Mining association rules with weighted items," Proc. Int. Database Engineering and Applications Symposium, IDEAS 98, pp.68–77, Cardiff, Wales, UK, 1998.
[7] F. Tao, "Weighted association rule mining using weighted support and significant framework," Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp.661–666, USA, 2003.
[8] W. Wang, J. Yang, and P.S. Yu, "WAR: Weighted association rules for item intensities," Knowledge Information and Systems, vol.6, pp.203–229, 2004.
[9] 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, pp.53–87, 2004.
[10] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using FP-Trees," IEEE Trans. Knowl. Data Eng., vol.17, no.10, pp.1347–1362, Oct. 2005.

[11] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," Data Mining and Knowledge Discovery, vol.15, pp.55–86, 2007.
[12] J. Wang, T. Fukasawa, S. Urabe, T. Takata, and M. Miyazaki, "Mining frequent patterns securely in distributed systems," IEICE Trans. Inf. & Syst., vol.E89-D, no.11, pp.2739–2747, Nov. 2006.
[13] C. Raissi, P. Poncelet, and M. Teisseire, "Towards a new approach for mining frequent itemsets on data stream," Journal of Intelligent Information Systems, vol.28, pp.23–36, 2007.
[14] A. Metwally, D. Agrawal, and A.E. Abbadi, "An integrated efficient solution for computing frequent and top-k elements in data streams," ACM Trans. Database Syst. (TODS), vol.31, no.3, pp.1095–1133, 2006.
[15] N. Jiang and L. Gruenwald, "Research issues in data stream association rule mining," SIGMOD Record, vol.35, no.1, pp.14–19, March 2006.
[16] C.K.-S. Leung and Q.I. Khan, "DSTree: A tree structure for the mining of frequent sets from data streams," Proc. 6th IEEE Int. Conf. on Data Mining (ICDM'06), pp.928–932, 2006.
[17] C.K.-S. Leung, Q.I. Khan, Z. Li, and T. Hoque, "CanTree: A canonical-order tree for incremental frequent-pattern mining," Knowledge and Information Systems, vol.11, no.3, pp.287–311, 2007.
[18] H. Xiong, P.-N. Tan, and V. Kumar, "Hyperclique pattern discovery," Data Mining and Knowledge Discovery, vol.13, pp.219–242, 2006.
[19] Frequent itemset mining dataset repository. Available from http://fimi.cs.helsinki.fi/data/
[20] UCI machine learning repository. Available from http://kdd.ics.uci.edu/
[21] J. Pisharath, Y. Liu, J. Parhi, W.-K. Liao, A. Choudhary, and G. Memik, NU-MineBench version 2.0 source code and datasets. Available from: http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html
[22] Z. Zheng, R. Kohavi, and L. Mason, "Real world performance of association rule algorithms," Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp.401–406, 2001.
[23] J.H. Chang and W.S. Lee, "estWin: Online data stream mining of recent frequent itemsets by sliding window method," Journal of Information Sciences, vol.31, no.2, pp.76–90, 2005.
[24] J. Li, D. Maier, K. Tuftel, V. Papadimos, and P.A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," SIGMOD Record, vol.34, no.1, pp.39–44, 2005.
[25] J.H. Chang and W.S. Lee, "A sliding window method for finding recently frequent itemsets over online data streams," J. Inf. Sci. Eng., vol.20, pp.753–762, 2004.
[26] 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," Proc. Fourth SIAM International Conference on Data Mining, pp.68–79, USA, 2005.
[27] J. Chen, B. Zhou, L. Chen, X. Wang, and Y. Ding, "Finding frequent closed itemsets in sliding window in linear time," IEICE Trans. Inf. & Syst., vol.E91-D, no.10, pp.2406–2418, Oct. 2008.
[28] U. Yun, "Mining lossless closed frequent patterns with weight constraints," Knowledge-Based Systems, vol.210, pp.86–97, 2007.
[29] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, and Y.-K. Lee, "CP-tree: A tree structure for single pass frequent pattern mining," Proc. 12th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD), pp.1022–1027, 2008.
[30] H. Huang, X. Wu, and R. Relue, "Association analysis with one scan of databases," Proc. Second IEEE Int. Conf. on Data Mining (ICDM'02), pp.629–632, 2002.
[31] W. Cheung and O.R. Zaane, "Incremental mining of frequent patterns without candidate generation or support constraint," Proc. 7th International Database Engineering and Applications Symposium (IDEAS), pp.111–116, 2003.
[32] J.-L. Koh and S.-F. Shieh, "An efficient approach for maintaining association rules based on adjusting FP-tree structures," Proc. 9th International Conference on Database Systems for Advanced Applications (DASFAA), pp.417–424, 2004.
[33] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, and Y.-K. Lee, "Efficient frequent pattern mining over data streams," Proc. 17th ACM Conference on Information and Knowledge Management (CIKM), pp.1447–1448, 2008.

Chowdhury Farhan Ahmed received his B.S. and M.S. degrees in Computer Science from the University of Dhaka, Bangladesh in 2000 and 2002 respectively. From 2003–2004 he worked as a faculty member at the Institute of Information Technology, University of Dhaka, Bangladesh. In 2004, he became a faculty member in the Department of Computer Science and Engineering, University of Dhaka, Bangladesh. Currently, he is pursuing his Ph.D. degree in the Department of Computer Engineering at Kyung Hee University, South Korea. His research interests are in the areas of data mining and knowledge discovery.

Syed Khairuzzaman Tanbeer received his B.S. degree in Applied Physics and Electronics, and his M.S. degree in Computer Science from the University of Dhaka, Bangladesh in 1996 and 1998 respectively. Since 1999, he has been working as a faculty member in the Department of Computer Science and Information Technology at the Islamic University of Technology, Dhaka, Bangladesh. Currently, he is pursuing his Ph.D. in the Department of Computer Engineering at Kyung Hee University, South Korea. His research interests include data mining and knowledge engineering.

Byeong-Soo Jeong received his B.S. degree in Computer Engineering from Seoul National University, Korea in 1983, his M.S. degree in Computer Science from the Korea Advanced Institute of Science and Technology, Korea in 1985, and his Ph.D. in Computer Science from the Georgia Institute of Technology, Atlanta, USA in 1995. In 1996, he joined the faculty at Kyung Hee University, Korea where he is now an associate professor at the College of Electronics & Information. From 1985 to 1989, he was on the research staff at Data Communications Corp., Korea. From 2003 to 2004, he was a visiting scholar at the Georgia Institute of Technology, Atlanta. His research interests include database systems, data mining, and mobile computing.

Young-Koo Lee received his B.S., M.S. and Ph.D. in Computer Science from Korea Advanced Institute of Science and Technology, Korea. He is a professor in the Department of Computer Engineering at Kyung Hee University, Korea. His research interests include ubiquitous data management, data mining, and databases. He is a member of the IEEE, the IEEE Computer Society, and the ACM.