Expert Systems with Applications 41 (2014) 694–708

Contents lists available at ScienceDirect

Expert Systems with Applications

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

Sliding window based weighted maximal frequent pattern mining over data streams

Gangin Lee [a], Unil Yun [a,*], Keun Ho Ryu [b]

[a] Department of Computer Engineering, Sejong University, Seoul, South Korea
[b] Department of Computer Science, Chungbuk National University, South Korea

ARTICLE INFO

Keywords:
Data mining
Data stream
Sliding window
Weighted maximal frequent pattern mining

ABSTRACT

As data have been accumulated more quickly in recent years, corresponding databases have also become huger, and thus, general frequent pattern mining methods have been faced with limitations that do not appropriately respond to the massive data. To overcome this problem, data mining researchers have studied methods which can conduct more efficient and immediate mining tasks by scanning databases only once. Thereafter, the sliding window model, which can perform mining operations focusing on recently accumulated parts over data streams, was proposed, and a variety of mining approaches related to this have been suggested. However, it is hard to mine all of the frequent patterns in the data stream environment since generated patterns are remarkably increased as data streams are continuously extended. Thus, methods for efficiently compressing generated patterns are needed in order to solve that problem. In addition, since not only support conditions but also weight constraints expressing items' importance are one of the important factors in the pattern mining, we need to consider them in mining process. Motivated by these issues, we propose a novel algorithm, weighted maximal frequent pattern mining over data streams based on sliding window model (WMFP-SW) to obtain weighted maximal frequent patterns reflecting recent information over data streams. Performance experiments report that MWFP-SW outperforms previous algorithms in terms of runtime, memory usage, and scalability.

1. Introduction

One of the data mining areas, frequent pattern mining has been actively studied together with various approaches and widely applied in numerous fields such as industry and business as well as computer science. As well-known fundamental frequent pattern mining algorithms, there are Apriori (Agrawal & Srikant, 1994) based on Breadth First Search and FP-growth (Han, Pei, Yin, & Mao, 2004) on the basis of Depth First Search. On the basis of those basic algorithms, a variety of pattern mining algorithms have been proposed, such as frequent pattern mining without the minimum support threshold specified by users (Chuang, Huang, & Chen, 2008; Li, 2009; Zhang & Zhang, 2011), sequential frequent pattern mining (Chang, Wang, Yang, Luan, & Tang, 2009; Muzammal & Raman, 2011; Yun, Ryu, & Yoon, 2011). Furthermore, frequent pattern mining has been utilized in extensive applications such as medical and bio data analysis (Sallaberry, Pecheur, Bringay, roche, & Teisseire, 2011; Xiong, He, & Zhu, 2010), stock market and protein networks (Sim, Li, Gopalkrishnan, & Liu, 2009), network environment (Fang, Deng, & Ma, 2009; Lin, Hsieh, & Tseng, 2010), traffic data analysis (Liu, Zheng, Chawla, Yuan, & Xing, 2011), analysis of web-click streams (Li, 2008; Li, Lee, & Shan, 2006), and so on. Frequent pattern mining can be applied not only in static databases like the above methods but also in data streams. Data streams mean that transaction data are added constantly, and thus, they have continuous and unlimited features. Note that data stream mining has to satisfy the following requirements (Farzanyar, Kangavari, & Cercone, 2012). (1) Each data element needed for data stream analysis has to be examined only once. (2) Although data streams become constantly large as data elements are continuously added, memory usage for mining operations should be limited to an acceptable and constant range. (3) All of the entered data elements have to be processed as soon as possible. (4) Results of data stream analysis should be available instantly as well as their quality should also be acceptable whenever users want the results. However, the previous frequent pattern mining methods do not satisfy these requirements since they have to conduct two or more database scans to mine frequent patterns. Therefore, to overcome these problems, mining approaches with only one scan (Tanbeer, Ahmed, Jeong, & Lee, 2009a, 2009b) have been suggested. Although these data stream mining methods can extract frequent patterns over data streams effectively, there are still the following issues. In data streams, data elements are constantly added and their sizes are continuously increased according to

* Corresponding author. Tel.: +822 34082902.
E-mail addresses: ganginlee@sju.ac.kr (G. Lee), yunei@sejong.ac.kr (U. Yun), khryu@chungbuk.ac.kr (K.H. Ryu).

0957-4174/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2013.07.094

accumulation of transaction data. Therefore, frequent patterns generated over data streams also become large, which means spending a lot of time mining the patterns, and thereby it can violate one of the requirements for the data stream mining, immediate processing. In order to solve the problem, closed frequent pattern (CFP) and maximal frequent pattern (MFP) notations Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005; Chen, Bie, & Xu, 2011; Farzanyar et al., 2012; Grahne & Zhu, 2005; Gouda & Zaki, 2005; Huang, Xiong, Wu, Deng, & Zhang, 2007; Li, 2009; Luo & Chung, 2008, 2012; Priya, Vadivel, & Thakur, 2012; Selvan & Nataraj, 2010; Shiozaki, Ozaki, & Ohkawa, 2006; Thomas, Valluri, & Karlapalem, 2006; Yang, Li, Zhang, & Hu, 2007; Yun, Shin, Ryu, & Yoon, 2012; Zeng, Pei, Wang, & Li, 2009, which can represent general frequent patterns as more compact forms, can be utilized. The MFP notation guarantees more efficient pattern compressibility than that of the CFP notation although slight pattern losses can occur when MFPs are again converted into the general ones. Consequently, if the MFP method with outstanding compressibility is applied into the data stream mining, we can find valid patterns over data streams more efficiently due to its advantage. As data have been accumulated in data streams continuously, importance of certain data entered a long time ago can decline or they may be no longer needed, while that of recently added data can be relatively high. To apply these characteristics in the mining process, a variety of window model-based mining approaches (Ahmed, Tanbeer, Jeong, & Lee, 2009; Chen, Shu, Xia, & Deng, 2012; Deypir, Sadreddini, & Hashemi, 2012; Farzanyar et al., 2012; Li, 2011; Mozafari, Thakkar, & Zaniolo, 2008; Shie, Yu, & Tseng, 2012; Tanbeer et al., 2009b; Zhang & Zhang, 2011) have been proposed, and damped window, landmark window, and sliding window techniques can be selectively applied according to characteristics of data streams. Especially since the sliding window-based mining approaches perform mining operations with only the most recent data among accumulated data streams, we can obtain recent high-quality results by using them. Data streams (or stream databases) are composed of numerous items, where each item represents objects in the real world. For example, in a retail market data stream, items reflect information regarding products, and in a data stream for traffic accidents, each item becomes accident information, where importance assigned to each item is actually different. Thus, we can obtain high-quality mining results reflecting not only items' frequency (or support) but also their importance (or weight) by applying the weight factor into the data stream mining. In this paper, we propose a novel algorithm satisfying the aforementioned issues, called weighted maximal frequent pattern mining over data streams based on sliding window model (WMFP-SW). To our knowledge, it is the first approach for mining weighted maximal frequent patterns (WMFPs) over sliding window model-based data streams. Through the proposed algorithm, we can always extract mining results regarding the latest data over data streams, and can gain the resulting patterns more quickly through the MFP technique and weight conditions. The main contributions of this work are summarized as follows.

1. We introduce a novel algorithm, WMFP-SW which can efficiently mine WMFPs with only one scan over sliding window-based data stream environment and a tree structure, WMFP-SW-tree used for the WMFP mining work. We also describe another tree structure, WMFP-tree managing WMFP information and performing subset-checking tasks effectively and an array structure, WMFP-SW-array for improving efficiency of mining operations. We help understand mining processes of the proposed algorithm by providing various examples.
2. Pruning strategies for reducing needless mining operations efficiently are described. Since WMFP-SW considers not only patterns' supports but also their weights when it decides whether extracted patterns are valid or not, the corresponding pruning range becomes lager than that of general frequent pattern mining. In addition, elements except for the latest ones are excluded in the mining procedure by the sliding window model, and thereby WMFP-SW conducts mining operations with faster runtime and less memory usage. We also provide a strategy which can prune unnecessary operations causing meaningless pattern generation in single paths.
3. To evaluate performance of the proposed algorithm, we compare ours with previous state-of-the-art algorithms, and various real and synthetic datasets applying weight conditions are used in performance experiments. These experimental results show that WMFP-SW presents more outstanding performance compared to the previous ones.

The remainder of this paper is organized as follows. Related work for this paper is introduced in Section 2, and thereafter, we describe details of the proposed algorithm, data structures, and pruning techniques in Section 3. Results of performance evaluation for ours and previous algorithms are presented in Section 4, and finally we conclude this paper in Section 5.

2. Related work

As an early frequent pattern mining algorithm, Apriori (Agrawal & Srikant, 1994) finds frequent patterns over static databases. The algorithm performs mining operations in Breadth First Search (BFS) manner and has to generate numerous candidate patterns in the process of actual frequent patterns. Moreover, to obtain complete results of frequent patterns, the algorithm should scan databases repeatedly, and especially in the worst case, the scanning task has to be performed as many as the number of items of the longest transaction in a database. Thereafter, FP-Growth algorithm (Han et al., 2004) based on Depth First Search (DFS) was proposed in order to overcome that problem, and most of the numerous algorithms suggested so far are on the basis of the framework and techniques of FP-growth. The algorithm can more efficiently conduct mining work with two fixed database scans and does not generate candidate patterns in comparison to Apriori.

2.1. Sliding window-based frequent pattern mining over data streams

Although mining methods based on FP-Growth have an effect on static databases, they are not suitable for data streams accumulating data continuously. Since these methods perform more than two database scans, they do not deal with data streams instantly. Moreover, since they construct trees with items remained after infrequent items are deleted, they have to discard previously generated trees and build new trees again if new transaction data are added into data streams. In data streams, although a certain item is currently infrequent, it can become frequent one according to addition of new transaction data. However, those two scan-based methods must read databases from the first again since they already eliminated infrequent items in the previous step. To solve this, mining methods suitable for data streams (Ahmed, Tanbeer, Jeong, Lee, & Choi, 2012; Chen & Wang, 2010; Tanbeer et al., 2009a) have been proposed, and they can perform mining tasks with only one database scan, thereby responding to changes of data streams immediately. After that, sliding window-based frequent pattern mining approaches (Ahmed et al., 2009; Chen et al., 2012; Deypir et al., 2012; Farzanyar et al., 2012; Li, 2011; Mozafari et al., 2008; Shie et al., 2012; Tanbeer et al., 2009b; Zhang & Zhang, 2011) have been proposed, which can mine frequent patterns considering the latest transaction data of large data streams. Especially in those paper (Tanbeer et al., 2009a, 2009b), an efficient

tree-restructuring method, BSM was proposed. The method performs restructuring operations more effectively than previous ones such as the path adjusting method, etc. IWFP algorithm (Ahmed et al., 2012) is a weighted frequent pattern mining algorithm over data streams, applying the BSM method. Among accumulated data streams, the most important elements are recently added data in general. In other words, importance of previously added data can be lowered or meaningless, while that of lately accumulated ones can be relatively higher. Therefore, to reflect these characteristics, the sliding window model can be applied into mining process. The method divides data streams into windows composed of a set of constant-sized transactions and finds frequent patterns from recently generated windows, where the size of windows and the number of them can be assigned as various values by users. Through the sliding window-based approach, we can always obtain frequent patterns reflecting recent information. In Tanbeer et al. (2009b), Tanbeer et al. suggested a frequent pattern mining algorithm over sliding window-based data streams, applying the BSM technique to tree restructuring steps in order to raise efficiency of mining operations.

2.2. Maximal frequent pattern mining over data streams

Mining all frequent patterns over data streams as well as static databases can cause numerous computational overheads in general if data sizes are large. In sliding window-based data stream mining, since the remaining parts except for the latest windows are not considered, the overheads can be reduced, but we cannot still avoid causing them if the size of windows or the number of them becomes large. For this reason, the MFP notation, which can compress generated frequent patterns into a small number of compressed forms, can be utilized in the mining process, and a variety of MFP mining methods (Burdick et al., 2005; Chen et al., 2011; Farzanyar et al., 2012; Gouda & Zaki, 2005; Grahne & Zhu, 2005; Huang et al., 2007; Luo & Chung, 2008, 2012; Priya et al., 2012; Selvan & Nataraj, 2010; Yang et al., 2007; Yun et al., 2012; Zeng et al., 2009) have been proposed. In MAFIA algorithm (Burdick et al., 2005), vertical bitmap representation was proposed so as to help mine MFPs more efficiently. The algorithm uses an additional data structure with a bitmap form to reduce the number of tree traversals. After the bitmap is constructed, MAFIA can know pattern's frequency through AND operation of the bitmap even though it does not try to traverse trees actually. FPmax* (Grahne & Zhu, 2005) is a state-of-the-art MFP mining algorithm, where FP-array, an additional data structure for mining MFPs more quickly, was proposed, thereby decreasing tree traversal times considerably. Since FP-array has information of patterns' supports, the algorithm can calculate them in advance before trees are actually traversed when growth processes are performed. Consequently, this technique not only can reduce tree traversal operations effectively but also can enhance pruning efficiency by preventing generation of needless conditional trees. However, since the above algorithms have two scan-based processes, they are not suitable for the data stream mining.

2.3. Applying weight conditions into frequent pattern mining over data streams

Each item existing in data streams has unique importance (or weight). For instance, given items over retail data streams, support information of them mean their sales volume, and their weight information represents prices or profits for each item. Therefore, when both of those two elements are considered, we can gain mining results reflecting complex factors in the real world. Weights of items in data streams are used in the mining process after they are converted into normalized values within a certain range. The reason is that if a weight of any item is too large, it is hard to denote its weighted support as a finite number of digits. The main challenge of applying weights is to maintain the anti-monotone property. However, the application generally destroys that property since weighted infrequent patterns can become weighted frequent ones as pattern growth operations are conducted. For this reason, researchers have made efforts to maintain the anti-monotone property, and a variety of methods (Ahmed et al., 2009, 2012; Wang & Zeng, 2011; Yun & Ryu, 2011; Yun et al., 2011, 2012) have been proposed. WFPMDS (Ahmed et al., 2009) mines weighted frequent patterns over data stream environment based on the sliding window model. The algorithm conducts tree restructuring work with the BSM technique and provides the most recent mining results from the sliding window whenever users request them. In this study, the framework of the proposed algorithm, WMFP-SW is based on the state-of-the-art MFP mining algorithm, FPmax* and the outstanding tree restructuring technique, BSM.

3. Weighted maximal frequent pattern mining over data streams based on sliding window model (WMFP-SW)

In this section, we first present preliminaries, that include definitions and concepts helping understand the proposed algorithm. In addition, details of WMFP-SW are introduced, where we describe data structures for the WMFP mining over data streams based on the sliding window, techniques related to them, pruning strategies for eliminating meaningless patterns and their operations from mining process, and an overall procedure of WMFP-SW and its explanation in detail.

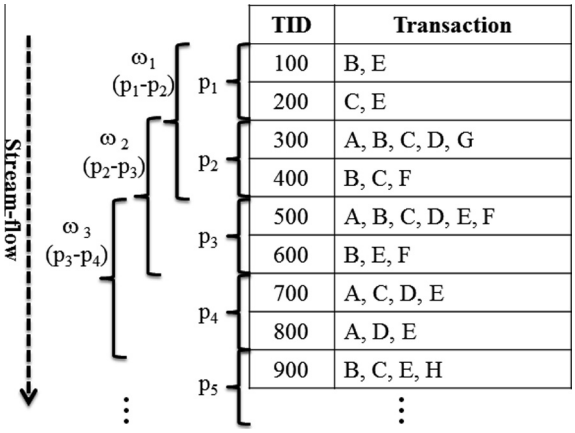
3.1. Preliminaries

Given a data stream, DS consisting of multiple transactions, Ts, DS includes Ts added so far and is denoted as $DS = [T_1, T_2, T_3, \ldots, T_n)$. Thus, new transactions such as $T_{n+1}, T_{n+2} \ldots$ can be added into DS after T_n. A set of items composing DS, I is denoted as $I = \{i_1, i_2, i_3, \ldots, i_n\}$, where each item, i has unduplicated unique values. Each T is composed of a partial or full set of I and expressed as $T = \{i_1, i_2, i_3, \ldots, i_k\}$, where every i has unique values and is not duplicated to each other. Ts are identified as unique ID, called TID. Each i in I has a weight, w, and a set of w, W is expressed as $W = \{w_1, w_2, w_3, \ldots, w_n\}$.

A pattern, P is composed of one or more items included in T. In the sliding window model, DS can be divided into multiple windows, denoted as $DS = \{\omega_1, \omega_2, \ldots, \omega_m\}$, where each window, ω consist of several panes (or batches) and represented as $\omega = \{p_1, p_2, \ldots, p_i\}$, $1 \leqslant i \leqslant n$. A pane, p is a set of transactions and denoted as $p = \{T_x, T_{x+1}, \ldots, T_y\}$, $1 \leqslant x \leqslant y \leqslant n$.

Example 1. Fig. 1 is to express data in Table 1 as a data stream, where both the window size (i.e. the number of panes) and pane size (i.e. the number of transactions) are set as 2. In the data stream, sliding window process is performed as follows. We first read p_1 and p_2 to fill ω_1. After that, when reading the next pane, p_3, we remove the old pane, p_1. Then, p_2 and p_3 belong to the current window, ω_2. In the same manner, the old pane, p_2 is deleted and the new pane p_4 is entered into the current window, ω_3 in the next step. As a result, the current window always has the latest data stream information, and thus, the sliding window-based mining methods can instantly provide users with frequent pattern results considering the most recent data whenever they request mining results.

Definition 1. Let $|\omega|$ be the number of transactions in a current window, ω, and i and j be the first and last transaction indexes among the transactions of ω respectively. Then, a support of any pattern, P, $SUP(P)$ can be calculated as the following formula.



TID	Transaction
100	B, E
200	C, E
300	A, B, C, D, G
400	B, C, F
500	A, B, C, D, E, F
600	B, E, F
700	A, C, D, E
800	A, D, E
900	B, C, E, H

Fig. 1. A sliding window-based data stream derived from Table 1.

Table 1
An example database and weight information of the database.

TID	Transaction	Item	Weight
100	B, E	A	0.5
200	C, E	B	0.7
300	A, B, C, D, G	C	0.8
400	B, C, F	D	1.0
500	A, B, C, D, E, F	E	0.4
600	B, E, F	F	0.9
700	A, C, D, E	G	0.6
800	A, D, E	H	0.3
900	B, C, E, H		

$$Check(P, TID_k) = \begin{cases} 1, if\ P\ is\ contained\ in\ T_k\ corresponding\ to\ TID_k \\ 0, otherwise \end{cases}$$

$$SUP(P) = \frac{\sum_{k=i}^{j} Check(P, TID_k)}{|\omega|} \quad (1)$$

$|\omega|$ is computed as $|\omega| = j - i + 1$. If $SUP(P)$ is not smaller than a given minimum support threshold, δ, P is a frequent pattern.

Definition 2. Given a pattern, $P = \{i_1, i_2, i_3, \ldots, i_m\}$ and a set of weights for P, $P_W = \{w_1, w_2, w_3, \ldots, w_m\}$, a weighted support of P, $WSUP(P)$ is computed as follows.

$$W_{average}(P) = \frac{\sum_{k=1}^{m} w_k}{m}$$

$$WSUP(P) = SUP(P) * W_{average}(P) \quad (2)$$

In the formula, m means the length of P, i.e. the number of items. If $WSUP(P)$ is larger than or equal to δ, P becomes a weighted frequent pattern.

Definition 3. Let P be a certain pattern and P' be a super pattern of P. Then, a set of all P's, $P_{superset}$ can be denoted as $P_{superset} = \{P'_1, P'_2, P'_3, \ldots, P'_n\}$. If P satisfies the following condition, it becomes MFP.

$$\forall P'_i \in P_{superset} \rightarrow SUP(P'_i) < \delta \quad (3)$$

Whether P is WMFP or not can be determined by the following formula changed from the equation (3).

$$\forall P'_i \in P_{superset} \rightarrow WSUP(P'_i) < \delta \quad (4)$$

3.2. Structures for WMFP-SW

Our WMFP-SW algorithm performs mining operations based on tree structures, and accordingly, we need tree structures suitable for finding WMFPs over sliding window-based data streams. For this reason, we define WMFP-SW-tree needed for pattern expansion, WMFP-tree managing extracted WMFP information and conducting subset checking operations, and WMFP-SW-array improving performance of the WMFP mining by reducing traversal time of conditional WMFP-SW-tree.

3.2.1. WMFP-SW-tree and WMFP-tree

The basic structure of WMFP-FP-tree is similar to FP-tree (Han et al., 2004), but the tree has additional weight data and is constructed with only one scan. As windows are changed, a part of the tree is removed, new data are added into the tree, and then tree restructuring operations are performed. WMFP-SW-tree is composed of a header table including items' supports (or counts), weights, and node links, and a tree actually storing data, where tail nodes with the last items for each inserted transaction have not only node counters (or supports) but also pane counters. As shown in Fig. 2(a) and (b), tail nodes have both support and pane counter information, and the pane counters are denoted as numbers in parentheses. For example, (1,0) means that a path including this tail node belongs to the first pane and its support is 1. Through this information, we can easily eliminate nodes of old panes and insert transaction data of new panes into the tree. A global WMFP-SW-tree is constructed with transaction data composing the current window while conditional WMFP-SW-trees are generated with partial data corresponding to each item in the header table. The global tree maintains support descending order by conducting tree restructuring processes continuously. Note that we explain the reason why to sort the tree in support descending order in the next section. WMFP-tree is used to store WMFPs obtained from pattern growth process. Its framework is similar to WMFP-SW-tree, but there are several different features. In its header table, support and weight information for each item is excluded since they are unnecessary when we manage WMFP information. In the WMFP-tree, level data for each node are included, instead of supports. Through them, subset checking operations for WMFPs can be performed more efficiently, and details of the subset checking procedures are available at (Grahne & Zhu, 2005). Once a global WMFP-SW-tree is constructed, it is maintained until mining process fully terminates, although its data are continuously changed due to frequent deletion and insertion tasks. In contrast, WMFP-tree is newly generated whenever mining requests occur from users since previously mined WMFP information is not available any longer. An overall procedure for mining WMFPs over data streams based on the sliding window model is as follows. (1) Transactions are read as many as the size of a current window, and they are inserted into a global WMFP-SW-tree. (2) The tree is restructured according to support descending order of the inserted items, if necessary. (3) If there occurs a mining request, WMFP mining operations are performed with the restructured global tree. (4) If new transaction data are entered to data streams, transactions included in previous old panes are deleted while new ones are inserted into the tree. (5) After returning to the step 2, we iterate the same work until the WMFP mining procedure is completely finished.

For each item contained in the header table of the tree, WMFP mining operations are conducted in the divide and conquer method, extracted WMFP information is stored in WMFP-tree, and thereafter the information is used for the WMFP subset checking process. Item selection in the header table starts from the bottom one, and we generate conditional WMFP-SW-trees regarding the selected items. After that, we find WMFPs, constructing conditional trees of the conditional ones recursively. Selected items are called prefix, and its length becomes longer as conditional trees are recursively generated. After the above operations are performed with respect to all of the items in the header table of the global tree, a complete set of WMFPs is obtained.

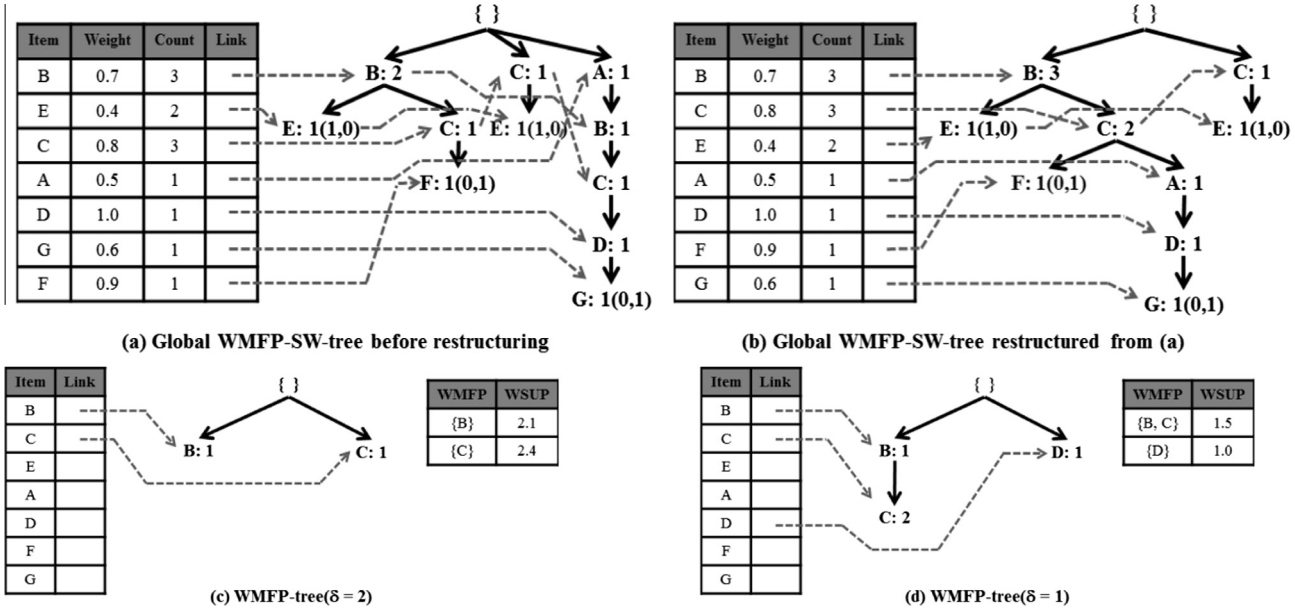


Fig. 2. WMFP-SW-tree and WMFP tree for ω_1.

Example 2. Fig. 2 represents a global WMFP-SW-tree when a current window is ω_1 in Fig. 1. ω_1 consists of *TID*:100, 200, 300, and 400, and its item order is not yet sorted since the current state is an initial step of building the tree.

Therefore, items of transactions are first stored in the tree according to their incoming sequence as shown in Fig. 2(a). Since the current WMFP-SW-tree has broken item order, restructuring process is required. Fig. 2(b) shows the tree after the restructuring work, where its item order is changed from B → E → C → A → D → G → F to B → C → E → A → D → F → G, and the number of nodes composing the tree is reduced from 11 to 9. Fig. 2(c) and (d) shows WMFP mining results when users request them. As shown in the figures, we can obtain different results according to a user-given minimum support threshold, δ without reconstructing a global tree due to the advantages of data stream mining. When δ is 2, we can gain WMFPs with 1-length, {B: 2.1} and {C: 2.4}, while when δ is 1, patterns with 2-length, {B, C: 1.5} and 1-length, {D: 1} are extracted.

3.2.2. WMFP-SW-array

The global WMFP-SW-tree can be constructed in the above manner with one scan, and all items consisting of the current window are contained in the global tree depending on the characteristics of data streams, since currently weighted infrequent items can become weighted frequent ones as the window is updated. However, conditional WMFP-SW-trees generated from the global tree do not need to have information regarding weighted infrequent items since we only require currently weighted frequent items in the process of extracting WMFPs. Thus, the invalid items should be removed when conditional trees are created. However, to conduct these operations, we have to know item information for conditional databases needed when constructing the conditional trees, which requires two scanning work. Motivated by Grahne and Zhu (2005), we therefore define an additional data structure, named WMFP-SW-array that makes conditional trees generated with only one scan, and improve mining performance by applying it into the mining process.

Definition 4. Given a conditional database for any prefix *P*, *CDB*, a set of items composing *CDB*, *ICDB* is denoted as $ICDB = \{i_1, i_2, i_3, \ldots, i_n\}$. Then, WMFP-SW-array for *CDB* is expressed as $(n-1)*(n-1)$ two-dimensional matrix, and is filled with the items in *ICDB*. Indexes for each row and column of the array are sorted in their support descending order, and each space of the array represents support information of items corresponding to lows and columns. Through it, we can learn items' support information in advance without scanning *CDB*.

However, this WMFP-SW-array technique cannot be applied to the global WMFP-SW-tree due to the following reason. In the sliding window-based data streams, all items are included in the global tree without pruning any item. As a result, the number of items has no choice but to be increased considerably although it is relatively smaller than considering entire data streams. Therefore, if we express all of the items on the window as WMFP-SW-array, we have to consume enormous runtime and memory resources to construct the corresponding array. Moreover, if the number of items in any window is enormously large, it is generally impossible to make the array in a common computing environment such as PC. However, if WMFP-SW-array is partially applied in constructing conditional trees, we can perform WMFP mining operations with no penalties in terms of runtime and memory usage. The reason is that arrays for conditional trees have relatively a small number of items by pruning invalid items as well as they can be deleted from memory when corresponding conditional trees are removed. Through the application of the array, we can obtain the following mining advantage.

Lemma 1. *Let CT be a set of initial conditional WMFP-SW-trees derived from a global tree and CT′ be a set of all possible conditional trees recursively generated from CT. Then, constructing CT′ except for CT can be performed with only one scan.*

Proof. Let *i* be an item selected in the header table of the global, i.e. prefix. Then, *i*'s conditional WMFP-SW-tree contained in *CT* is generated as the following sequence. (1) We first read all upper nodes of the node including *i* in the global tree, and construct *i*'s conditional database, *i*'s *CDB* using the read nodes. In this process, support information of *i*'s *CDB* is identified, and then *i*'s WMFP-SW-array is generated through the information. (2) Scanning *i*'s *CDB*

again, we construct *i*'s WMFP-SW-tree. Once *i*'s array is created, subsequent tasks for *CT'* require only one scan when we build conditional trees derived from *i*'s conditional tree. Assuming that any item, *j* is newly added into the prefix, *ij*'s WMFP-SW-tree included in *CT'* is constructed as follows. (1) We first find support information for the newly selected item, *j* in *i*'s WMFP-SW-array. (2) Scanning *ij*'s *CDB* and referring to the support information found from the array, we build *ij*'s WMFP-SW-tree at a time. That is, WMFP-SW-array replaces support checking operations by the first tree scan. For this reason, we can generate all conditional WMFP-SW-trees except for the initial ones in *CT* with only one scan.

Example 3. Fig. 3 is WMFP-SW-arrays obtained from Fig. 2(b), where a minimum support threshold, δ is set as 1. In this figure, numbers for each matrix mean items' supports, and real numbers next to item names represent weight information. Gray spaces of the matrixes are currently invalid items.

Note that the global WMFP-SW-array shown in Fig. 3(a) is used to help understand a concept and an example of the WMFP-SW-array technique although it is not actually made in the mining process. If G is selected as prefix in Fig. 2(b), we can know support information of valid items through Fig. 3(a). Since we already learn this information, we can construct G's conditional tree at a time traversing the tree in Fig. 2(b) and removing needless items such as {E,F}. In the process of generating the tree, G's conditional WMFP-SW-array is constructed at the same time as shown in Fig. 3(b). Since G's tree has a single path in this example, recursive operations is not performed any longer, and processes for mining WMFPs are conducted. Fig. 3(c)–(e) represent WMFP-SW-arrays regarding F, D, and A's conditional trees respectively. However, arrays for the items, E, C, and B are not generated. If the item, E is selected as prefix, the corresponding conditional database is {B}, {C}.Then, since there is no item pair in the database, its WMFP-SW-array is not required. The case of C is also equal to that of E. Since B's conditional database and tree have an empty set, its array is not constructed.

3.3. Updating WMFP-SW-tree according to the sliding window

In data streams based on the sliding window, transactions consisting of the window are continuously updated depending on stream flows. To perform this process, we need techniques for restructuring trees, deleting old panes, and inserting new panes. In this section, details of them are described and related examples are presented.

3.3.1. Restructuring WMFP-SW-tree

In the two scan-based mining methods such as FP-growth, the first scan calculates items supports, where we can know meaningless item information and their order. In the second scanning process, a complete global tree is obtained by sorting transactions in the order and inserting them into the tree. In contrast, since a global tree should be generated within only one scan in the sliding window-based data streams, another solution is needed. In order to build the global tree with one scan, transactions of the current window have to be inserted into the tree in advance according to their incoming order or other standards. In this process, we can know support information regarding items composing transactions and their order. Thereafter, we can obtain a completely sorted global tree by restructuring the previous tree referring to the found order without constructing it again. the restructuring procedure is as follows. (1) For each unsorted path in a global tree, we extract it one at a time. (2) Extracted paths are sorted in support descending order and inserted into the tree again. (3) For the nodes in the sorted and inserted paths, their node links are reconnected reflecting changed positions. (4) Until there is no longer unsorted path, we repeat the above tasks. When any path is extracted, its common support is based on that of the very end node (i.e. the leaf node) in the path. Therefore, supports of corresponding nodes in the tree are reduced as many as those of extracted paths' nodes, where if there are nodes with 0 supports, they are removed. In the process of reinserting sorted paths, if there are parts matched with the entered ones among the previous paths in the tree, these parts are not inserted in new locations but corresponding supports are just increased. If the above operations terminate, we can gain a complete global WMFP-SW-tree finishing the tree restructuring process. When WMFP-SW-tree is restructured in the sliding window-based data stream environment, support descending order is more effective than weight ascending order. In the weighted frequent pattern mining area, the most well-known sorting methods are support descending order and weight ascending order. In any tree sorted in support descending order, items with relatively high supports are located in the upper portion of the tree. As a result, sharing parts between the nodes naturally become large, which contributes to building the tree having a more compact size. However, since weights of items cannot be sorted, additional operations for computing *MaxW*, which is used to prune weighted infrequent patterns, is required. Details of it are explained in the next section again. Any tree sorted in weight ascending order has a complicated and sparse structure since it is not sorted in support descending order and thus sharing parts are relatively small. On the other hand, we can immediately learn *MaxW* without any additional task due to the effect of weight ascending order. In case mining operations are conducted with static databases and two scan-based process, support descending order can be more efficient than weight ascending order and vice versa according to characteristics of used datasets. However, when weight ascending order is applied to data streams, it causes fatal deterioration in performance despite the

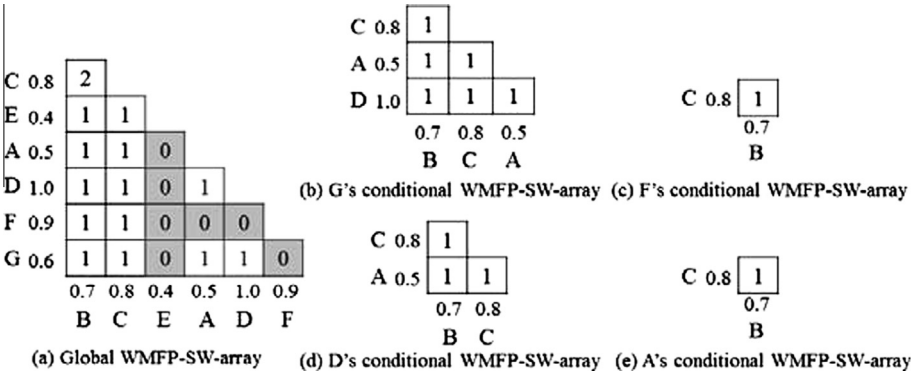


Fig. 3. WMFP-SW-arrays generated from the tree in Fig. 2(b).

advantage of *MaxW*. There are very large and various items within any window in general although they are smaller than those considering whole data streams. Furthermore, all items including weighted infrequent ones remain in the global tree according to the feature of data streams, and especially the invalid ones are likely to be located in the upper or middle portion of the tree. Therefore, the corresponding tree becomes more complex and larger in comparison to the tree construction in static databases, which means that we have to spend more time and memory traversing the tree compared to another tree having the same data but sorted in support descending order. In contrast, if we apply support descending order, invalid items hardly have an effect on constructing the compact tree since they are located in bottom portion of the tree.

Consequently, since applying support descending order over data streams based on the sliding window can reduce the number of nodes which have to be searched for mining WMFPs, it guarantees more outstanding performance than that of weight ascending order in spite of the disadvantage by *MaxW*.

Example 3. Fig. 4 shows how to restructure WMFP-SW-tree corresponding to ω_1 over the data stream in Fig. 1. Restructuring operations are performed from the first path. Lists in Fig. 4(a) represents item orders before and after the tree is sorted, where Item, Cnt, and W means item names, items' supports (or counts), and their weights respectively. Since the first three paths have an already sorted state, they do not need to be sorted again and the operations for them are omitted. However, since the last path is not sorted, its restructuring tasks are conducted. When the path is reinserted into the tree after it is sorted, a part of nodes in the path, {B, C} is shared since it is overlapped with the previous path of the tree, {B, C, F}, while the remainder of the path, {A, D, G} is inserted under the node, {C} as shown in Fig. 4(b). The red part of the tree means newly created nodes. Note that corresponding node links are reconnected with respect to the node changed in the process of the tree restructuring steps, but we omit them in the figure for a clear representation.

3.3.2. Removing old panes and inserting new panes

If users request mining results after the restructuring, WMFP mining operations are performed. After that, we have to delete old panes and add new ones to update the window and prepare the next mining request. These operations can be easily and exactly processed by pane counter information contained in tail nodes. Given a set of tail nodes included in a certain old pane of any WMFP-SW-tree, $T = \{t_1, t_2, \ldots, t_n\}$, we can obtain the tree extracting the old pane by removing from the tree the nodes as many as the pane counters corresponding to each t. Since there are sharing parts between nodes as items are inserted in the tree, a part of tree's nodes can belong to multiple transactions. However, one tail node belongs to only one transaction, and pane counters assigned to tail nodes contain support information for specific paths corresponding to the tail nodes. Therefore, we can directly eliminate old panes with no additional operations by utilizing this tail node and pane counter information. Given any old pane consisting of multiple transactions *Trs*, $P_{old} = \{Tr_1, Tr_2, \ldots, Tr_n\}$ and a set of tail nodes, *ts* for each *Tr*, $T = \{t_1, t_2, \ldots, t_n\}$, each t has support values regarding P_{old} and other normal panes. Therefore, if we decrease supports from t_1 to the root nodes in the path including t_1 according to P_{old}'s counter number, it is the same as deleting Tr_1. Iterating the same operations regarding all remaining tail nodes, all data included in P_{old} can be completely extracted, where certain nodes with 0 supports are eliminated. When nodes are deleted from a tree structure, we should consider not only the nodes but also their child nodes in general in order to prevent unintended losses of nodes. However, we can immediately remove nodes with zero supports in WMFP-SW-tree without considering their child nodes, and the following property and lemma prove that the deletion is reasonable.

Property 1. *A support of any node in WMFP-SW-tree is always greater than or equal to the total supports of all its child nodes.*

Proof. In WMFP-SW-tree, all nodes are added from its root node. Moreover, since transaction data are inserted into the tree in a line, upper nodes always have support values higher than or equal to those of lower nodes. Therefore, given any node, n and a set of child nodes of n, n', the total support of n' cannot exceed the value of n, since all of the nodes in n' share n.

Lemma 2. *When a certain node, n is removed since its support is 0, if there are child nodes of n, they also have the same support as it and thus deleted.*

Proof. If any node to be deleted, n does not have any child node, removal of it has no problems. Moreover, although n has child nodes, we can also erase it without any node losses. The reason is that any child node does not have more than parent's support depending on Property 1. That is, given a node with zero support, n and any child node of n, nc, the support of nc always becomes zero since the value is at most n's value. For this reason, we can directly eliminate nodes in WMFP-SW-tree if their supports are zero.

After old panes are completely deleted, new panes are inserted in the global tree, where added item's order follows support descending order of the items in the tree just before old panes are extracted. Inserting new panes depending on the previous order has an effect on constructing a more compact tree in comparison to inputting them according to their incoming order, which also contributes to reducing operations needed for tree restructuring processes since the number of nodes to be considered becomes smaller. After the insertion of the new panes, item order suitable for the newly updated tree is sorted again, and then, the tree is restructured

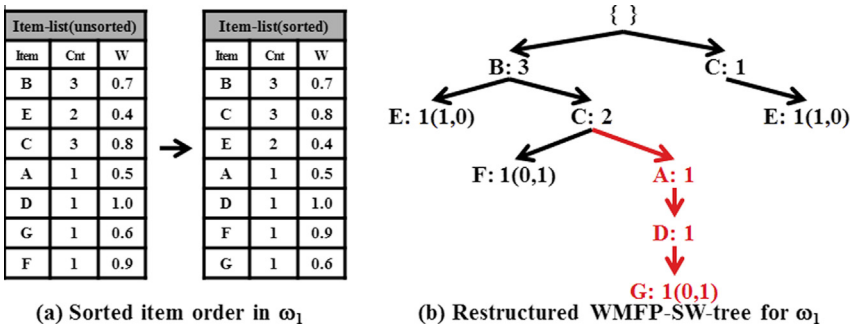


Fig. 4. Restructuring operations for WMFP-SW-tree (ω_1).

depending on the new order. Thereafter, if mining requests occur, WMFP-SW algorithm mines WMFPs and provides users with the result. Otherwise, the algorithm maintains the latest state at all times extracting and adding old and new panes again.

Example 4. Let us consider updating the tree in Fig. 4(b) as the current window is changed from ω_1 to ω_2 as shown in Fig. 1. The process of deleting the old pane, p_1 is shown in Fig. 5, and gray dotted lines and letters in Fig. 5(a) indicate transactions belonging to p_1. Fig. 5(b) represents WMFP-SW-tree after p_1 is extracted. Since the remaining pane, p_2 becomes the first pane in the current window, the corresponding pane counters shift from the second to the first. Fig. 6 shows how to insert the new pane, p_3 in the tree and restructure it. The previous item order is B → C → E → A → D → F → G. Therefore, the transactions contained in p_3 are sorted as {B, C, E, A, D, F} and {B, E, F}. As shown in Fig. 6(a), the current global tree includes all of the transactions corresponding to ω_2 according to the insertion of p_3. However, since the current order of the tree is an unsorted state, we have to restructure it. The first and second paths of the tree do not need to be sorted again since they already have correct order, while the third and fourth ones should be re-sorted since they have broken order. In Fig. 6(b), it is observed that the third path, {B, C, E, A, D, E} is sorted according to the new order, where gray and red parts mean extracting and reinserting the path respectively. After the insertion, the node, E becomes a new tail node of the path. Then, the path has two tail nodes as shown in the figure. That is, the path, {B, C, F, A, D, E} is composed of the two transactions, {B, C, F} and {B, C, F, A, D, E}, and they can be distinguished as the tail nodes. Fig. 6(c) represents processes of restructuring the last path. Since the path does not have any part sharing with the tree contrasted with the case of the previous third path, it is assigned to a new one. The tree after the restructuring tasks are finished is shown in Fig. 6(d), where we can observe that the current tree becomes more compact compared to the previous tree before the restructuring operations shown in Fig. 6(a).

3.4. Pruning strategies in WMFP-SW

In the mining process, not all mining operations generate valid patterns. They can find either meaningful patterns or useless ones as the case may be. Thus, to reduce the number of operations causing invalid pattern generation, pruning techniques can be used. In this section, we describe pruning strategies which can be applied in mining WMFPs over data streams based on the sliding window model.

3.4.1. Pruning patterns by MaxW

Weight conditions not only can be utilized to find valid patterns related to characteristics of the real world but also can be used for a strong pruning constraint. However, applying them into the mining process does not satisfy the anti-monotone property in general. That is, any weighted infrequent pattern can become weighted frequent ones as mining operations are gradually conducted on the basis of the weight conditions. As a result, incorrect pruning operations by the weights can cause pattern losses. To solve this and perform efficient pruning procedures, we define a pruning condition applied in the sliding window model, named *MaxW*.

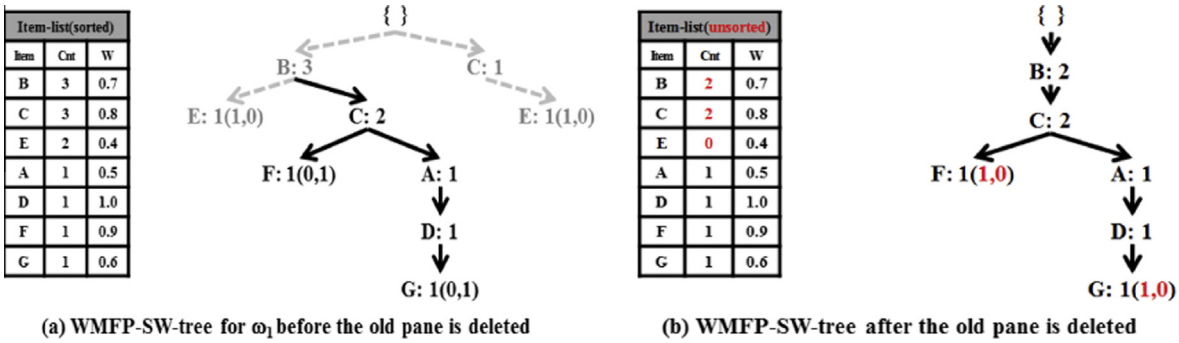
Definition 5. In a global WMFP-SW-tree, *MaxW* is set as the largest value among weights of all items included in the current window, while, in a conditional WMFP-SW-tree, it is set as the maximum weight in items of the current conditional tree.

Example 5. In Fig. 6(d), *MaxW* is assigned as follows. In the global tree, it is set as 1.0 since this value is the maximum weight in the tree's items. Therefore, assuming that a minimum support threshold, δ is 2, all of the items except for G continue to remain in the tree although the items, A and E are currently weighted infrequent. Since multiplying 1 (G's support) by 1.0 (the current *MaxW*) is lower than δ, G and all possible patterns including it do not become weighted frequent in any case. Although A and E are currently invalid, certain patterns containing them can become valid WMFPs. However, if real weights for each item are directly applied into the pruning process instead of *MaxW*, then none of patterns with the items, A and E are generated since they are pruned in the initial stages. On the other hand, if *MaxW* is used, we can prevent these pattern losses since they are preserved. In G's conditional WMFP-SW-tree, *MaxW* is still 1.0 since the item, D continues to be contained into the tree. However, in A's and F's conditional trees, *MaxW* is lowered as 0.9 since D does not participate in the mining process any longer and the new maximum weight becomes 0.9 in the conditional trees.

Even if *MaxW* for a global tree is only used in the whole pruning steps, it does not have any effect on generating correct mining results. However, *MaxW* for conditional trees can be more decreased than that for the global tree as shown in the above example, which means that the range of generated candidate WMFPs can be more reduced. Note that all of the patterns remaining after the pruning by *MaxW* do not become valid WMFPs immediately. Since they are candidates, only a portion of them become real WMFPs if their results computed by the equation (2) are not smaller than a given minimum support threshold and they satisfy the equation (4).

3.4.2. Pruning patterns in single-paths

There are additional considerations for the WMFP mining. In the general MFP mining method, if a single path occurs in the process of constructing conditional trees in a recursive manner, we can simply extract MFP from the single path by combining prefix added so far with all items of the single path. However, in the



(a) WMFP-SW-tree for ω_1 before the old pane is deleted (b) WMFP-SW-tree after the old pane is deleted

Fig. 5. Deleting the old pane from WMFP-SW-tree in the Fig. 4(b).

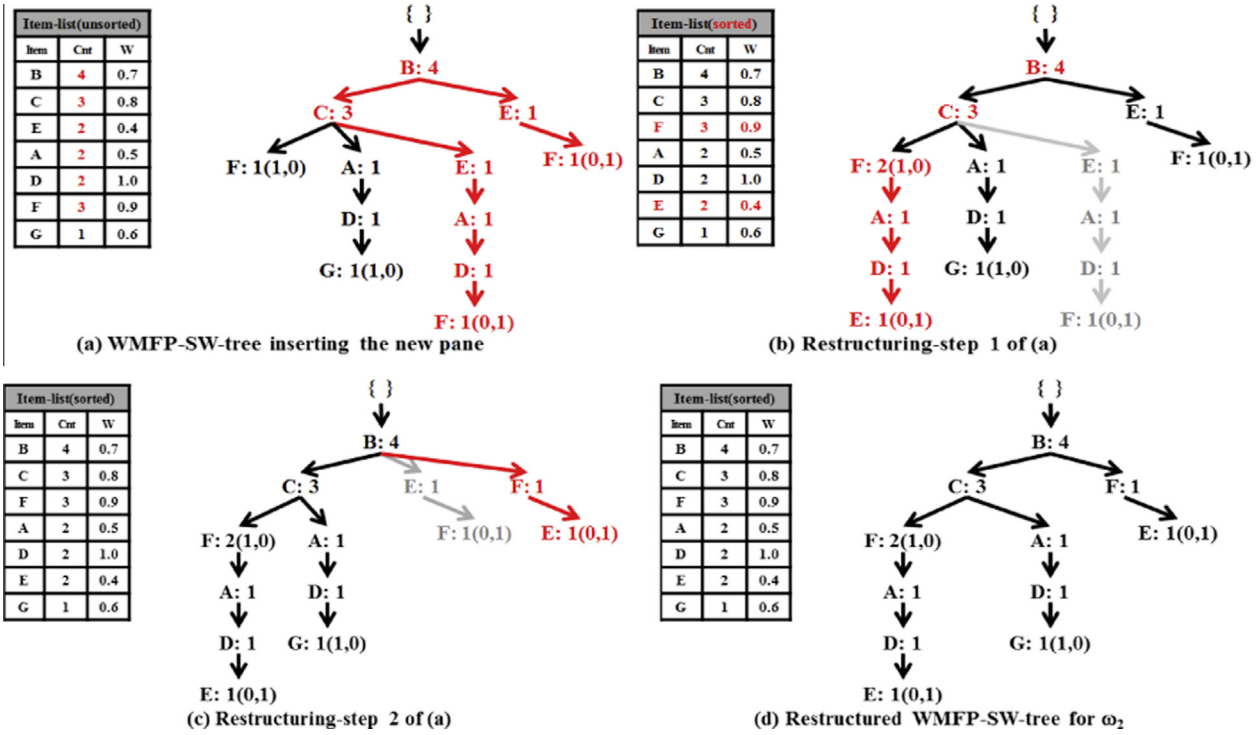


Fig. 6. Inserting the new pane into WMFP-SW-tree in the Fig. 5(b) and restructuring the tree.

WMFP mining approach, since the combination is a candidate, we have to confirm whether it is valid or not through the equation (2) and (4). If it is valid, the corresponding pattern directly becomes real WMFP. However, if it is weighted infrequent, we need additional operations for finding WMFPs from the single path, and to conduct these tasks efficiently, we use the following strategies.

Definition 6. If a candidate WMFP obtained from a certain single path, S is an invalid pattern, a set of all possible subsets from S, PS is generated and denoted as $PS = \{ps_1, ps_2, \ldots, ps_{n-1}\}$, where an empty set, ps_0 and the entire set of S, ps_n are excluded since they do not lead to generating valid WMFPs. Each ps is a set of combinations with the same length. For example, ps_1 includes all of the subsets having one length in S.

If a combination of prefix with single path's items is not WMFP, we have to find meaningful WMFPs again combining the prefix with all of the elements in PS one by one. Therefore, these operations generally require a number of time resources, but we can effectively reduce them by the following lemma.

Lemma 3. *Given a single path with n length, $S = \{i_1, i_2, \ldots, i_n\}$ and a set of all valid subsets included in S, $PS = \{ps_1, ps_2, \ldots, ps_{n-1}\}$, mining WMFPs from ps_{n-1} contributes to improving performance of the proposed algorithm by pruning needless pattern operations occurring in S.*

Proof. Considering that WMFP mining tasks are performed in regard to all of the elements in PS without any strategy, the number of the components is 2^n (it is exactly $2^n - 2$, but the constant is excluded on the basis of the Big-O notation). Assuming that k is the time spent for the one element, the total time to process all of them becomes $O(k * 2^n)$. However, if the operations start from ps_{n-1}, then the following advantage occurs according to the feature of the maximal pattern mining. In case valid WMFPs are discovered from ps_{n-1}, all subsets contained in the found patterns need not be considered again since we want only WMFPs, not all general patterns. Thus, the subsets included between ps_1 and ps_{n-2} are excluded in the next mining process, and thereby we can quicken the mining speed. Furthermore, if WMFPs found from ps_{n-1} include all items composing S, the corresponding total time becomes $O(k * n)$ (for any single path with n items, ps_{n-1} always has n elements), since we have only to calculate the operations regarding the only ps_{n-1}. As a result, given the execution time for all of the elements, T and that by the proposed technique, T', it is always true that $T \geqslant T'$

Through the above strategy, we can reduce mining times spent on single paths, and the advantage by it becomes larger as used datasets generate more single paths.

3.5. WMFP-SW algorithm

In this section, an overall procedure of WMFP-SW algorithm applying the proposed methods and techniques is described. Through the steps shown in Fig. 7, we can observe how WMFPs are mined over the sliding window-based data streams.

In the WMFP-SW procedure, the algorithm first makes preparation for the WMFP mining (line 1) and conducts its operations while there exist transactions which are not yet processed from DS (lines 2–8). Since $T = \emptyset$ means that none of data is inputted into T, if it is true, WMFP-SW fills an window from DS and constructs T (lines 3–4). Otherwise, the window is updated since $T \neq \emptyset$ indicates that previous data were filled in T (line 5). After that, T is restructured if necessary, and tasks for mining WMFPs are performed if there are requests from users (lines 6–8).

In the sub procedure, *Create_Window*, steps for inserting transactions into an empty tree are conducted (lines 1–8). The window size represents the number of panes and the pane size means the number of transactions. Transactions are inserted in T according to their incoming order (line 4), and thereafter, the algorithm computes support descending order regarding the items composing T (line 8). When the current window is updated according to the

flow of data streams, the sub procedure, *Update_Window* is performed. The old pane is deleted (lines 1–6), nodes' supports of the paths belonging to it are decreased, and specific nodes with 0 supports are directly removed in T based on Lemma 2 (lines 4–6). After the deletion, the algorithm shifts pane counters for the previous tail nodes to the left one by one (line 7). Then, the new pane is added into T, where each transaction of it is sorted in the previous order (lines 8–9). Whenever these transactions are inserted, corresponding tail node information is set (line 10). After the window is updated, its order is sorted again to prepare the next restructuring process. If the window is created or updated, T is restructured through the sub procedure, *Restructure_Tree*. For each path in T, extraction, sorting, and reinsertion tasks are conducted (lines 1–4), and nodes with 0 support are eliminated in these processes (line 5). If the algorithm performs the tasks of the WMFP-SW procedure by the line 6, it means that preparations for mining WMFPs are completely finished. If mining requests occur from users at this time, the sub procedure, *Mine_WMFP* is called. Then, for each items in the header table of T, the algorithm excludes some of the items which are permanently unnecessary in this mining process through the technique by *MaxW* (lines 2–5). If T is a single path (a global tree generally has multiple paths and thus this routine is mainly the part for conditional trees), the algorithm confirms whether it is really WMFP to combine T's all items with the prefix added so far (lines 7–15). If the condition of the line 9 is satisfied, this combination is stored into P (line 10). Otherwise, subsequent operations are performed on the basis of Definition 6 and Lemma 3, and resulting patterns are added into P (lines 12–15). If T has multiple paths (line 16), for each item of the header table in T, the algorithm generates the corresponding conditional WMFP-SW-tree and WMFP-SW-array, sets prefix, and then calls *Mine_WMFP* recursively (lines 17–20), where the created WMP-SW-array contributes to constructing subsequent conditional trees with only one scan according to Lemma 1. After the procedure is completely finished, we can gain the most recent WMFP pattern information corresponding to the current window data.

4. Performance evaluation

4.1. Experimental environment

In this section, performance evaluation for the proposed algorithm, WMFP-SW is conducted and its analysis results are provided. Target algorithms are WFPMDS (Ahmed et al., 2009) for mining weighted frequent patterns over the sliding window-based

Input : A data stream, ***DS***, A window size, $Size_w$, A pane size, $Size_p$, A minimum support threshold, δ
Output : A set of WMFPs, ***P***
Variables : ***T*** : A global WMFP-SW-tree, ***Order*** : Item order, ***prefix*** : A current prefix added so far
MaxW : A weight value used for pattern-pruning, ***PS*** : A data structure used for single paths

WMFP-SW procedure
1. ***P*** = ∅, ***T*** = ∅ , ***Order*** = ∅, ***prefix*** = ∅;
2. while there are transactions to be processed in ***DS***, do
3. if ***T*** = ∅, do
4. call ***Create_Window***(***T***, ***Order***);
5. else call ***Update_Window***(***T***, ***Order***);
6. call ***Restructure_Tree***(***T***, ***Order***);
7. if mining request occurs by users, do
8. call ***Mine_WMFP***(***T***, ***prefix***, ***P***);

Create_Window(***T***, ***Order***)
1. A current window size, ω = ∅, A current pane size, ***p*** = ∅;
2. while $\omega \neq Size_w$, do
3. while ***p*** ≠ $Size_w$, do
4. insert a transaction of ***DS*** into ***T*** according to its incoming order;
5. ***p*** ← ***p*** + 1;
6. ***p*** = ∅;
7. $\omega \leftarrow \omega + 1$;
8. ***Order*** ← support descending order for the items included in the current ***T***;

Update_Window(***T***, ***Order***)
1. for each path, p_i in the old pane, do
2. find a tail node, t_i for p_i;
3. for each node, n_k in p_i, do //bottom up manner
4. $n_k.support \leftarrow n_k.support - t_i.support.first$;
5. if $n_k.support = 0$, do
6. delete n_k;
7. shift pane counters of all remaining tail nodes in ***T*** to left by one;
8. for each transaction, tr_i of the new pane in ***DS***, do // $1 \le i \le Size_p$
9. insert tr_i into ***T*** according to ***Order***;
10. set tail node information for tr_i;
11. ***Order*** ← support descending order for the items included in the current ***T***;

Restructure_Tree(***T***, ***Order***)
1. for each path, p_i in ***T***, do
2. if p_i is not sorted, do
3. extract and sort p_i depending on ***Order***;
4. reinsert p_i into ***T*** and set p_i's tail node information again;
5. delete certain nodes with 0 support from ***T***;

Mine_WMFP(***T***, ***prefix***, ***P***)
1. for each items, i_k in ***T***'s header table, do //bottom-up manner
2. set i_k as ***prefix***;
3. ***MaxW*** ← the maximum weight among the items' weights in ***T***;
4. if ***prefix***.*support* * ***MaxW*** < δ, do
5. go to line 1;
6. else
7. if ***T*** is a single-path, do
8. ***pattern*** = ***prefix*** ∪ all items in ***T***;
9. if *WSUP*(***pattern***) ≥ δ and ***Subset_Checking***(***pattern***) is false, do
10. ***P*** = ***P*** ∪ ***pattern***;
11. else
12. ***PS*** ← all possible combinations for the items in ***T***;
13. for each combination, c_i in ***PS***, do
14. if *WSUP*(***prefix*** ∪ c_i) ≥ δ and ***Subset_Checking***(***prefix*** ∪ c_i) is false, do
15. ***P*** = ***P*** ∪ (***prefix*** ∪ c_i);
16. else //multiple path
17. generate i_k's conditional WMFP-SW-tree, ***T'*** and WMFP-SW-array;
18. for each items, i_m in the header table of ***T'***, do
19. ***prefix'*** = ***prefix*** ∪ i_m;
20. call ***Mine_WMFP***(***T'***, ***prefix'***, ***P***); //recursive call

Fig. 7. WMFP-SW algorithm.

data streams and its optimized version, WFPMDS*. As the advanced method of CPS-tree (Tanbeer et al., 2009b) which is a famous and outstanding algorithm for mining frequent patterns over the sliding window-based data streams, WFPMDS is suitable for our algorithm. Since there is no algorithm that can mine WMFPs over data streams based on the sliding window model among the previous algorithms to our knowledge, we optimized WFPMDS to allow it to extract WMFPs. We wrote all the algorithms in C++ language, and they were executed in 3.33 GHz CPU, 3 GB RAM, and WINDOWS 7 OS environment. For runtime and memory usage experiments, real datasets, which are available at http://fimi.cs.helsinki.fi/data/, Accidents, Pumsb, Retail, and Mushroom were used. Table 2 includes information regarding these datasets in detail. Furthermore, for scalability tests, synthetic datasets, TaLbNc and T10I4DxK are used and details of them are shown in Table 2, which can be obtained from the IBM dataset generator available at http://www.almaden.ibm.com/software/projects/hdb/ resources. Table 3 represents window and pane size information regarding the used datasets. As shown in the table, various values are assigned as the sizes for extensive experiments. For instance, W_1 of Accidents dataset means 50 K * 2 = 100 K. On the other hand, since both TaLbNc and T10I4DxK are the datasets for the scalability tests, their window sizes are fixed as a certain value. Performance evaluation for the algorithms is conducted as follows. First, the algorithms continue to update their own windows, iterating insertion, deletion, and restructuring steps over data streams for each dataset.

In the middle of these steps, if mining requests occur from users, they mine valid patterns with their own restructured global trees. Note that we assume that those requests occur after all transactions are read in order to perform experiments quickly and exactly. After the above processes completely terminate, we analyze their performance with figures of their total runtimes and maximum memory usages.

4.2. Runtime results

In the runtime experiments of this section, items' weights for the used real datasets are set between 0.5 and 0.8. Graphs in Figs. 8–11 represent the results of runtimes for each algorithm and dataset, where the windows have fixed sizes and these values are varied according to each dataset as shown in the figures. Fig. 8 is the results of the Accidents dataset, where WMFP-SW guarantees the most outstanding runtime performance in all cases. Moreover, the proposed algorithm shows almost constant runtime results regardless of the minimum support threshold, δ while those of the other ones become larger as δ is gradually decreased. Fig. 9 represents execution times to mine patterns with the Pumsb dataset. These results show that WMFP-SW can conduct mining operations more quickly in common with the previous cases.

In Figs. 10 and 11 for the Retail and Mushroom datasets, our algorithm also guarantees the fastest and most stable performance in every case. Due to the strategies for the WMFP-SW-array and single paths, WMFP-SW algorithm provides users with the best results regardless of datasets and δ. Moreover, degree of runtime increases due to the reduction of δ is smaller than that of the others. In contrast to our algorithm, WFPMDS shows the slowest runtimes in all of the cases. Since the algorithm extracts all weighted frequent patterns, its resulting patterns are larger than those of the other algorithms as well as needed runtimes become longer. The optimized algorithm of WFPMDS, WFPMDS* can discover the same results as those of WMFP-SW, but its performance falls behind the proposed algorithm although it outperforms the original version, WFPMDS. The performance gap between the algorithms is remarkably represented in Fig. 11. While the runtime of WMFPMDS is considerably increased as δ becomes low, the other two algorithms have slowly increasing runtime graphs due to the advantage of WMFP.

Table 2
Details of datasets used in experiments.

Dataset	# of Trans	# of Items	Avg. trans. size	Size (M)
Accidents	340,183	572	45	33.8
Pumsb	49,046	2113	74	15.9
Retail	88,162	16,470	13	3.97
Mushroom	8124	120	23	0.83
T10L1000N10000	98,043	10,000	10	4.96
T20L2000N20000	99,863	20,000	20	10.73
T30L3000N30000	99,871	30,000	30	16.43
T40L4000N40000	99,853	40,000	40	22.13
T10I4D100 K	100,000	1000	10	3.83
T10I4D200 K	200,000	1000	10	7.86
T10I4D400 K	400,000	1000	10	15.3
T10I4D600 K	600,000	1000	10	23
T10I4D800 K	800,000	1000	10	30.6
T10I4D1000 K	1,000,000	1000	10	38.3

Table 3
Window and pane sizes for the used datasets.

Dataset	Pane size (K)	Window size				
		W_1	W_2	W_3	W_4	W_5
Accidents	50	2	3	4	5	6
Pumsb	5	2	4	6	8	–
Retail	10	2	3	4	5	6
Mushroom	1	2	4	6	8	–
TaLbNc	10	4(fixed)				
T10I4DxK	10	4(fixed)				

4.3. Results of memory usage

Experiments performed in this section present memory usage results for each real datasets. Used parameters are equal to those of the runtime experiments. Fig. 12 presents memory usage results for the Accidents dataset, where we can show that all of the algorithms have stable memory consumption regardless of δ. However, our WMFP-SW requires the lowest memory usages in all cases, and this tendency is similarly represented in the other experiments. In Fig. 13 for the Pumsb dataset, although the gap between WMFP-SW and the others is smaller than that in the Accidents, WMFP-SW still outperforms the others.

In the results of the Retail dataset shown in Fig. 14, the three algorithms have stable memory usages except that WMFP-SW consumes a little more memory when δ is 0.02%. Although their absolute memory usages are different from each other, the reason why their memory consumption is almost constant is as follows. Since

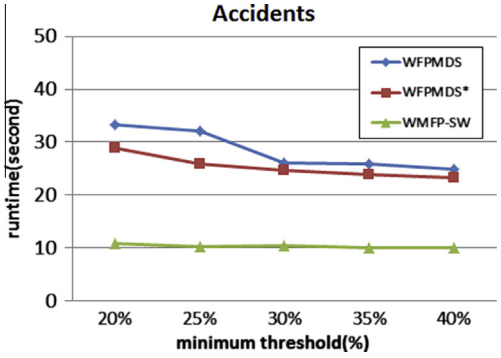


Fig. 8. Accidents dataset (W_2).

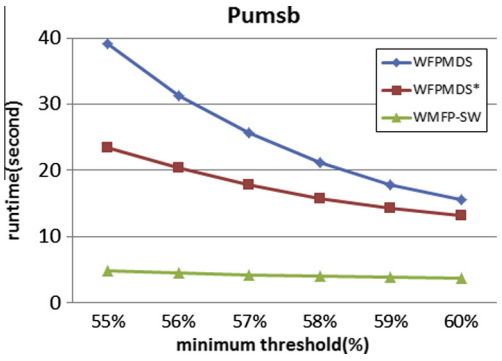


Fig. 9. Pumsb dataset (W_2).

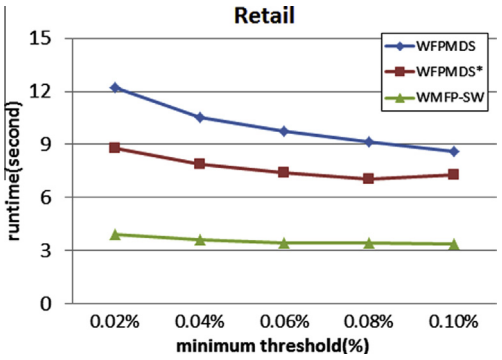


Fig. 10. Retail dataset (W_5).

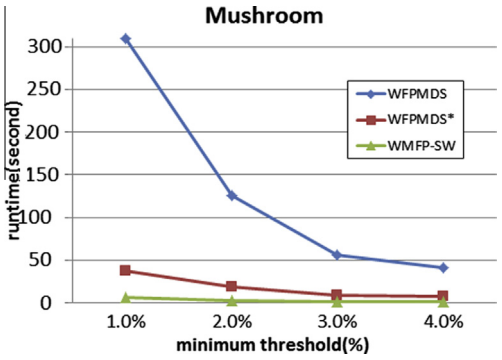


Fig. 11. Mushroom dataset (W_1).

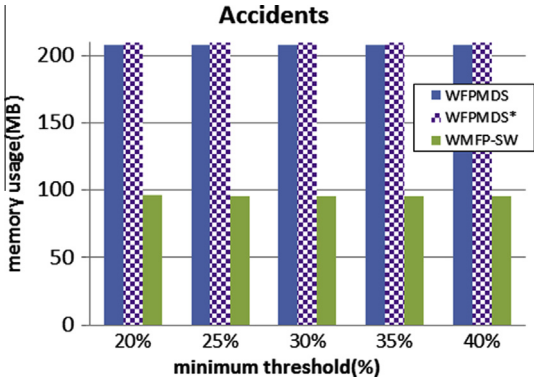


Fig. 12. Accidents dataset (W_2).

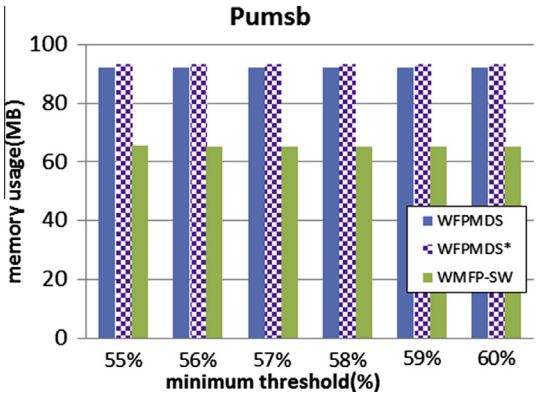


Fig. 13. Pumsb dataset (W_2).

all of the transaction data of the current window are contained into a global tree, the amount of memory used for constructing the global tree is far larger than that consumed in the mining operations in general. Furthermore, since the global trees for each algorithm maintain the same state regardless of δ, almost constant memory is required as shown in the figures, where gaps of absolute memory usages between those algorithms occur due to the proposed techniques and structural differences among them. However, in Fig. 15 for the Mushroom, different characteristics are observed. As shown in Table 2, the number of items composing the Mushroom dataset is relatively small since it has a dense feature and the smallest file size of the used datasets. As a result, its constructed global tree consumes relatively little memory, and thus, the memory usages are visibly changed according to δ as shown in the figure. Especially, WFPMDS$*$ consumes the most memory in every case compared to the other algorithms. WMFP-SW shows the most outstanding memory performance in general, although its memory usage becomes gradually similar to that of WFPMDS depending on the increase of δ.

4.4. Experimental results by changes of window sizes

In this section, we present runtime and memory usage results regarding changes of window sizes. Used parameters are based on Table 3 and assigned weights are also equal to the previous experiments.

Figs. 16 and 17 are results of the Retail dataset, and δ is fixed as 0.1%. Results of the Mushroom dataset are shown in Figs. 18 and 19, where δ is set as 5%. In Fig. 16, it is observed that all of the three algorithms have increasing runtimes as the window size becomes large. On the other hand, we can show that required runtimes are gradually decreased according to the increasing window size in Fig. 18. The reason is as follows. If there are dense and sparse datasets with the same size, restructuring trees made in the sparse dataset generally needs longer runtimes than that doing in the dense dataset since the former case creates more complex and lager trees. In the Retail, as a window size becomes larger, the number of restructured paths is also more increased since divided windows have a sparse feature. As a result, corresponding runtimes gradually increase as shown in the figure. In the Mushroom, its divided windows lose a dense characteristic as the original dataset is split into small parts. Accordingly, longer execution times are needed with respect to the small windows. However, in case the window size increases, the corresponding runtimes are reduced in contrast to the case of the Retail, since constructed trees have increasingly more dense nature as the window has more items. In the memory tests shown in Figs. 17 and 19, they have a similar tendency although their absolute memory consumption is

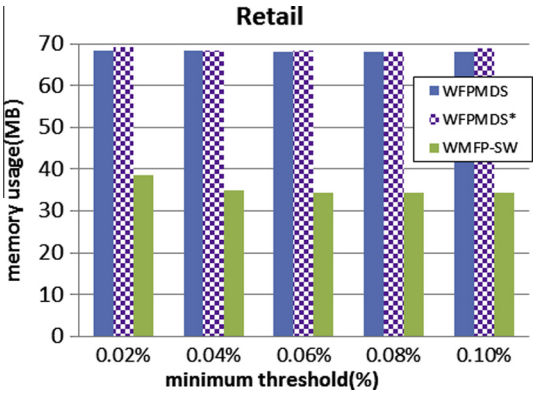


Fig. 14. Retail dataset (W_5).

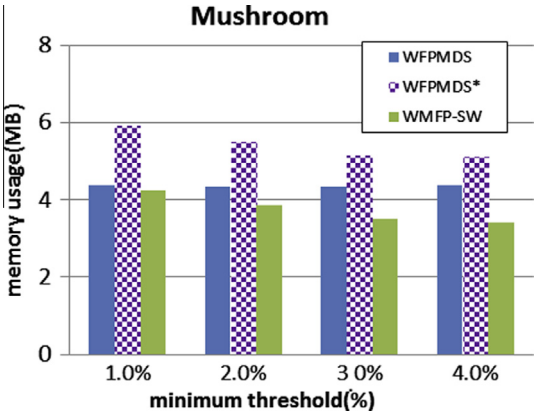


Fig. 15. Mushroom dataset (W_1).

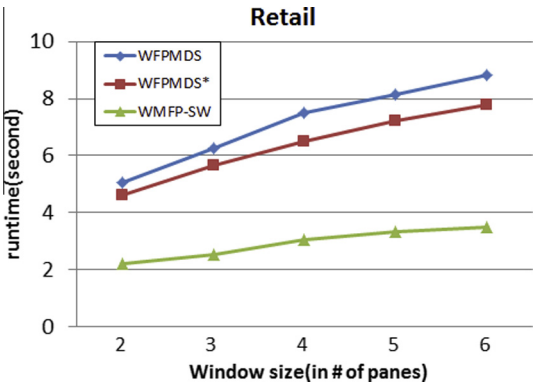


Fig. 16. Runtime of Retail dataset ($\delta = 0.1\%$).

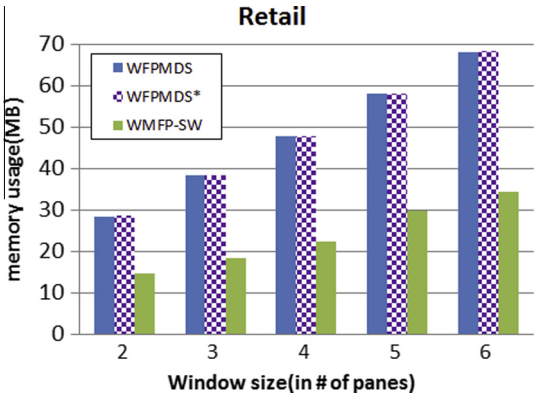


Fig. 17. Memory usage of Retail dataset ($\delta = 0.1\%$).

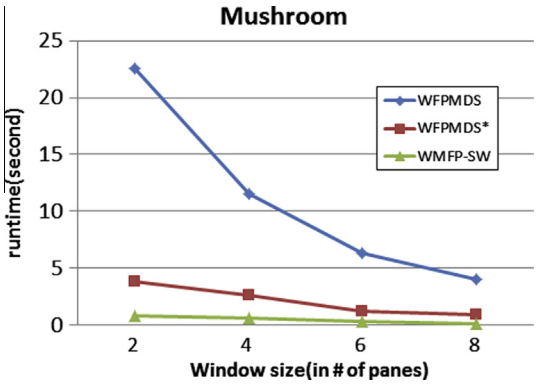


Fig. 18. Runtime of Mushroom dataset ($\delta = 5\%$).

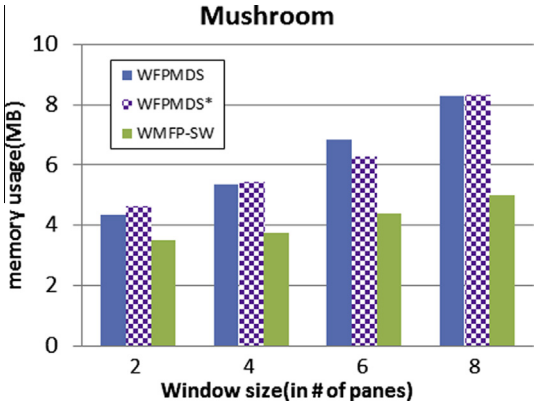


Fig. 19. Memory usage of Mushroom dataset ($\delta = 5\%$).

different from each other. In contrast to the results of the runtime experiments in Figs. 16 and 18, memory usages are gradually increased as the window sizes become larger since the sizes of the trees corresponding to the current windows increase due to the rise of the windows, as shown in Figs. 17 and 19.

4.5. Scalability results

Figs. 20 and 21 present scalability results of T10I4DxK datasets in terms of runtime and memory usage, where they have random weights between 0.5 and 0.8, their δ value is fixed as 0.1%, and x value increases from 100 to 1000 K. In WFPMDS and WFPMDS$*$ of Fig. 20, we can show that their graph slopes sharply rise after x is 200 K, which means that their scalability results by the increasing transactions are unfavorable. On the other hand, the runtime of WMFP-SW stably increases according to the growth of the transactions. In Fig. 21, all of the algorithms show steady memory consumption regardless of the number of transactions. Since they are tree-based algorithms, if the number of attributes is not increased, they have almost constant memory usages even though the number of transactions becomes larger. The next experiments, Figs. 22 and 23 represent results of the TaLbNc datasets, and their experimental environment is equal to that of T10I4DxK. In the experiments, the results of Fig. 22 have a tendency different from those of Fig. 20. WFPMDS$*$ and WMFP-SW guarantee stable runtime increases although the number of attributes becomes larger while WFPMDS has sharply increasing runtimes. Since WFPMDS

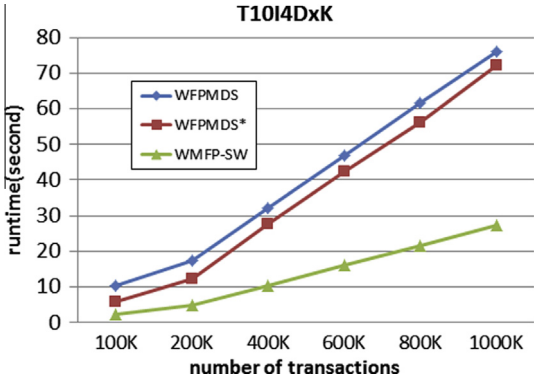


Fig. 20. Runtime scalability of T10I4DxK (δ = 0.1%).

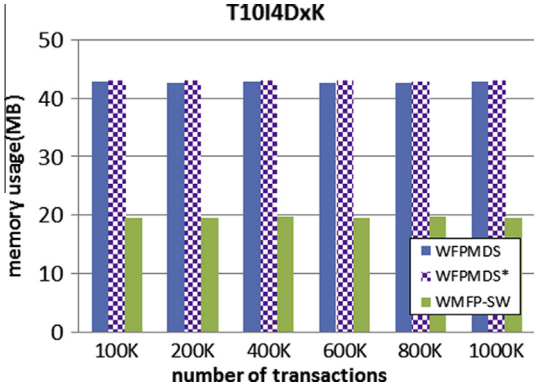


Fig. 21. Memory usage scalability of T10I4DxK (δ = 0.1%).

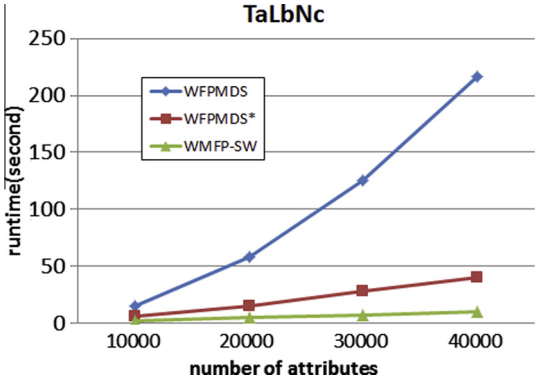


Fig. 22. Runtime scalability of TaLbNc (δ = 0.1%).

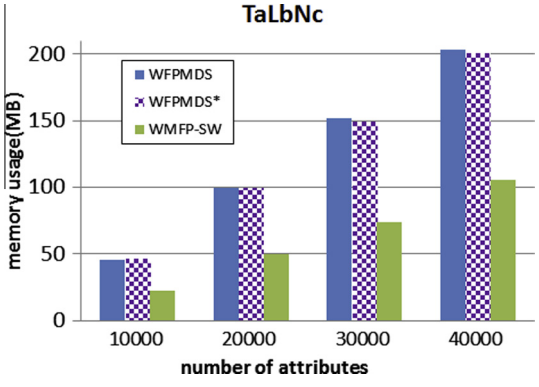


Fig. 23. Memory usage scalability of TaLbNc (δ = 0.1%).

has to extract an enormous number of WFPs according to the increase of the attributes, the corresponding graph slope rapidly rises as shown in the figure, while the others have relatively better runtime scalability due to the advantage of the maximal pattern technique. WMFP-SW especially shows the best scalability performance due to the proposed techniques. Increasing the number of attributes eventually means that more information should be contained into the tree, and thus, corresponding memory usages for the algorithms gradually grow like Fig. 23. However, we can observe that memory scalability of both WFPMDS and WFPMDS$*$ falls behind that of the proposed algorithm, WMFP-SW. Through the provided experimental results, we can know that the proposed algorithm presents outstanding performance in terms of runtime, memory usage, and scalability over the sliding window-based data streams. In addition, it is observed that our algorithm outperforms the previous algorithms in the performance experiments regarding changes of window sizes.

5. Conclusions

In this paper, we proposed an algorithm for mining weighted maximal frequent patterns (WMFPs) over data streams based on the sliding window model, called WMFP-SW, and suggested data structures needed to find WMFPs, WMFP-SW-tree, WMFP-tree, and WMFP-SW-array. WMFP-SW effectively extracted WMFPs reflecting the latest information over data streams due to the described window updating and restructuring methods, the reduction of tree scans by the WMFP-SW-array, mining strategies for single paths, and so on. The extensive experiments presented in this paper showed that the proposed algorithm could mine WMFPs more effectively than the previous algorithms by proving that WMFP-SW guaranteed more outstanding performance in terms of runtime, memory usage, and scalability. The suggested techniques and strategies can be applied in other areas such as closed frequent pattern mining, high utility pattern mining, etc. in addition to the maximal frequent pattern mining, and applying them is expected to contribute to improving performance in various fields.

Acknowledgments

This research was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF No. 2013005682 and 20080062611).

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th international conference on very large data, bases* (pp. 487–499), September 1994.

Ahmed, C. F., Tanbeer, S. K., Jeong, B. S., & Lee, Y. K. (2009). An efficient algorithm for sliding window-based weighted frequent pattern mining over data streams. *IEICE Transactions, 92-D*(7), 1369–1381.

Ahmed, C. F., Tanbeer, S. K., Jeong, B. S., Lee, Y. K., & Choi, H. J. (2012). Single-pass incremental and interactive mining for weighted frequent patterns. *Expert Systems with Applications, 39*(9), 7976–7994.

Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., & Yiu, T. (2005). MAFIA: A maximal frequent itemset algorithm. *IEEE Transactions on Knowledge and Data Engineering, 17*(11), 1490–1504.

Chang, L., Wang, T., Yang, D., Luan, H., & Tang, S. (2009). Efficient algorithms for incremental maintenance of closed sequential patterns in large databases. *Data & Knowledge Engineering, 68*, 68–106.

Chen, Y., Bie, R., & Xu, C. (2011). A new approach for maximal frequent sequential patterns mining over data streams. *International Journal of Digital Content Technology and its Applications, 5*(6), 104–112.

Chen, H., Shu, L., Xia, J., & Deng, Q. (2012). Mining frequent patterns in a varying-size sliding window of online transactional data streams. *Information Sciences, 215*, 15–36.

Chen, L., & Wang, C. (2010). Continuous subgraph pattern search over certain and uncertain graph streams. *IEEE Transactions on Knowledge and Data Engineering, 22*(8).

Chuang, K. T., Huang, J. L., & Chen, M. S. (2008). Mining Top-k frequent patterns in the presence of the memory constraint. *The International Journal on Very Large Data Bases, 17*(5), 1321–1344.

Deypir, M., Sadreddini, M. H., & Hashemi, S. (2012). Towards a variable size sliding window model for frequent itemset mining over data streams. *Computers & Industrial Engineering, 63*(1), 161–172.

Fang, G., Deng, Z., & Ma, H. (2009). Network traffic monitoring based on mining frequent patterns. *Fuzzy Systems and Knowledge Discovery, 7*, 571–575.

Farzanyar, Z., Kangavari, M. R., & Cercone, N. (2012). Max-FISM: Mining (recently) maximal frequent itemsets over data streams using the sliding window model. *Computers & Mathematics with Applications, 64*(6), 1706–1718.

Gouda, K., & Zaki, M. J. (2005). GenMax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery, 11*(3), 223–242.

Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using FP-trees. *IEEE Transactions on Knowledge and Data Engineering, 17*(10), 1347–1362.

Han, J., Pei, J., Yin, Y., & Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery, 8*(1), 53–87.

Huang, Y., Xiong, H., Wu, W., Deng, P., & Zhang, Z. (2007). Mining maximal hyperclique pattern: A hybrid search strategy. *Information Sciences, 177*(3), 703–721.

Li, H. (2008). A sliding window method for finding Top-k path traversal patterns over streaming Web click-sequences. *Expert Systems with Applications, 36*(3), 4382–4386.

Li, H. (2009). Interactive mining of Top-K frequent closed itemsets from data streams. *Expert Systems with Applications, 36*(7), 10779–10788.

Li, H. (2011). MHUI-max: An efficient algorithm for discovering high-utility itemsets from data streams. *Journal of Information Science, 37*(5), 532–545.

Li, H., Lee, S., & Shan, M. (2006). DSM-PLW: Single-pass mining of path traversal patterns over streaming Web click-sequences. *Computer Networks, 50*, 1474–1487.

Lin, K. W., Hsieh, M., & Tseng, V. S. (2010). A novel prediction-based strategy for object tracking in sensor networks by mining seamless temporal movement patterns. *Expert Systems with Applications, 37*(4), 2799–2807.

Liu, W., Zheng, Y., Chawla, S., Yuan, J., & Xing, X. (2011). Discovering spatio-temporal causal interactions in traffic data streams. In *Proceedings of the 17th international conference on knowledge discovery and data mining* (pp. 1010–1018).

Luo, C., & Chung, S. M. (2008). A scalable algorithm for mining maximal frequent sequences using a sample. *Knowledge and Information Systems, 15*(2), 149–179.

Luo, C., & Chung, S. M. (2012). Parallel mining of maximal sequential patterns using multiple samples. *The Journal of Supercomputing, 59*(2), 852–881. http://dx.doi.org/10.1007/s11227-010-0476-1.

Mozafari, B., Thakkar, H., & Zaniolo, C. (2008). Verifying and mining frequent patterns from large windows over data streams. In *Proceedings of the 24th international conference on data, engineering* (pp. 179–188).

Muzammal, M., & Raman, R. (2011). Mining sequential patterns from probabilistic databases. *Advances in Knowledge Discovery and Data Mining*, 201–221.

Priya, R. V., Vadivel, A., & Thakur, R. S. (2012). Maximal pattern mining using fast CP-tree for knowledge discovery. *International Journal of Information Systems and Social Change, 3*(1), 56–74.

Sallaberry, A., Pecheur, N., Bringay, S., roche, M., & Teisseire, M. (2011). Sequential patterns mining and gene sequence visualization to discover novelty from microarray data. *Journal of Biomedical Informatics, 44*, 760–774.

Selvan, S., & Nataraj, R. V. (2010). Efficient mining of large maximal bicliques from 3D symmetric adjacency matrix. *IEEE Transactions on Knowledge and Data Engineering, 22*(12), 1797–1802.

Shie, B., Yu, P. S., & Tseng, V. S. (2012). Efficient algorithms for mining maximal high utility itemsets from data streams with different models. *Expert Systems with Applications, 39*(17), 12947–12960.

Shiozaki, H., Ozaki, T., & Ohkawa, T. (2006). Mining closed and maximal frequent induced free subtrees. In *Proceedings of the 6th IEEE international conference on data mining* (pp. 14–18).

Sim, K., Li, J., Gopalkrishnan, V., & Liu, G. (2009). Mining maximal quasi-bicliques: Novel algorithm and applications in the stock market and protein networks. *Statistical Analysis and Data Mining, 2*(4), 255–273.

Tanbeer, S. K., Ahmed, C. F., Jeong, B. S., & Lee, Y. K. (2009a). Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences, 179*(5), 559–583.

Tanbeer, S. K., Ahmed, C. F., Jeong, B. S., & Lee, Y. K. (2009b). Sliding window-based frequent pattern mining over data streams. *Information Sciences, 179*(22), 3843–3865.

Thomas, L.T., Valluri, S.R., & Karlapalem, K. (2006). MARGIN: Maximal frequent subgraph Mining. In *Proceedings of the 6th IEEE international conference on data mining* (pp. 1097–1101).

Wang, J., & Zeng, Y. (2011). DSWFP: Efficient mining of weighted frequent pattern over data streams. *Fuzzy Systems and Knowledge Discovery, 2*, 942–946.

Xiong, Y., He, J., & Zhu, Y. (2010). TOPPER: An algorithm for mining Top k patterns in biological sequences based on regularity measurement. In *2010 IEEE international conference on bioinformatics and biomedicine workshops* (pp. 283–288).

Yang, C., Li, Y., Zhang, C., & Hu, Y. (2007). A novel algorithm of mining maximal frequent pattern based on projection sum tree. *Fuzzy Systems and Knowledge Discovery, 1*, 458–462.

Yun, U., & Ryu, K. (2011). Approximate weight frequent pattern mining with/without noisy environments. *Knowledge-Based System, 24*(1), 73–82.

Yun, U., Ryu, K., & Yoon, E. (2011). Weighted approximate sequential pattern mining within tolerance factors. *Intelligent Data Analysis, 15*(4), 551–569.

Yun, U., Shin, H., Ryu, K., & Yoon, E. (2012). An efficient mining algorithm for maximal weighted frequent patterns in transactional databases. *Knowledge-Based Systems, 33*, 53–64.

Zeng, X., Pei, J., Wang, K., & Li, J. (2009). PADS: A simple yet effective pattern-aware dynamic search method for fast maximal frequent pattern mining. *Knowledge and Information Systems, 20*(3), 375–391.

Zhang, X., & Zhang, Y. (2011). Sliding-window Top-k pattern mining on uncertain streams. *Journal of Computational Information Systems, 7*(3), 984–992.