Information Sciences 179 (2009) 3843–3865

Contents lists available at ScienceDirect

Information Sciences

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

Sliding window-based frequent pattern mining over data streams

Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong *, Young-Koo Lee

Department of Computer Engineering, Kyung Hee University, 1 Seochun-dong, Kihung-gu, Youngin-si, Gyeonggi-do 446-701, Republic of Korea

ARTICLE INFO

Article history:
Received 29 October 2008
Received in revised form 9 June 2009
Accepted 26 July 2009

Keywords:
Frequent pattern
Data stream
Sliding window
Tree restructuring

ABSTRACT

Finding frequent patterns in a continuous stream of transactions is critical for many applications such as retail market data analysis, network monitoring, web usage mining, and stock market prediction. Even though numerous frequent pattern mining algorithms have been developed over the past decade, new solutions for handling stream data are still required due to the continuous, unbounded, and ordered sequence of data elements generated at a rapid rate in a data stream. Therefore, extracting frequent patterns from more recent data can enhance the analysis of stream data. In this paper, we propose an efficient technique to discover the complete set of recent frequent patterns from a high-speed data stream over a sliding window. We develop a Compact Pattern Stream tree (CPS-tree) to capture the recent stream data content and efficiently remove the obsolete, old stream data content. We also introduce the concept of dynamic tree restructuring in our CPS-tree to produce a highly compact frequency-descending tree structure at runtime. The complete set of recent frequent patterns is obtained from the CPS-tree of the current window using an FP-growth mining technique. Extensive experimental analyses show that our CPS-tree is highly efficient in terms of memory and time complexity when finding recent frequent patterns from a high-speed data stream.

1. Introduction

Since the introduction of frequent pattern (or itemset) mining by Agrawal et al. [1], it has been actively and widely studied by the data mining and knowledge discovery research community. Mining frequent patterns from data streams has become one of the most important and challenging problems [10] for a wide range of online applications including retail chain data analysis, network traffic analysis, online analytical processing (OLAP), web-server log and click-stream mining, telecommunication data analysis, e-business and stock data analysis, and sensor network data analysis. A data stream is a continuous, unbounded, and timely ordered sequence of data elements generated at a rapid rate. Unlike traditional static databases, mining frequent patterns from stream data, in general, has additional processing requirements; i.e., each data element should be examined at most once and processed as fast as possible with the limitation of available memory. Moreover, the knowledge embedded in stream data is more likely to change as time goes by. Thus, in general, research has focused on finding and extracting useful information from the recently-arrived data elements, since processing the recent data is usually important for the applications that handle stream oriented data. Discovering recent frequent patterns from current data streams, however, has proven challenging.

To handle (or mine) continuously-generated data streams, time windows are commonly used, and are an efficient approach [3,8,22,26,27,29]. Depending on the stream data mining application, three different window models, named the

* Corresponding author. Tel.: +82 31 201 2932; fax: +82 31 202 1723.
E-mail addresses: tanbeer@khu.ac.kr (S.K. Tanbeer), farhan@khu.ac.kr (C.F. Ahmed), jeong@khu.ac.kr, jeong_khu@yahoo.com (B.-S. Jeong), yklee@khu.ac.kr (Y.-K. Lee).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.07.012

landmark window model, damped window model, or sliding window model can be used. In the landmark window model, data mining is performed utilizing all the data between a particular point of time, called the landmark, and the current time. In the damped window approach (also referred to as the time fading window approach), different weights are assigned to the data depending on the order of appearance of the data; the new data receive higher weights than the older data. In the sliding window model, only the fixed length of recently-generated data is used in mining operations. For example, given a window W on a transactional database, only the latest $|W|$ transactions or all transactions in the last $|W|$ time unit are utilized for data mining. As new transactions arrive, the oldest transactions in the sliding window expire. The sliding window model is therefore widely used to find recent frequent patterns in data streams [4,22,26,27,29].

Even though a number of algorithms have been proposed to discover frequent patterns in stream data, most of them do not differentiate recently-generated information from obsolete information that may currently be useless, insignificant, or possibly invalid. Moreover, almost all of the algorithms designed to generate recent frequent patterns from data streams produce approximate results [8,17,28,30,40]. The main problem these approaches face is to devise a data structure that is able to capture the stream content, in full, with a single-pass and in a memory-efficient manner. The inefficient data structure makes it difficult to generate a complete set of frequent patterns and results in poor mining performance. The FP-tree-based FP-growth mining technique proposed by Han et al. [11] has been found to be an efficient algorithm for mining frequent patterns from a static database using a prefix-tree data structure. The performance gain achieved by the FP-growth is due in most part to the highly compact nature of the FP-tree, which stores only the frequent items in a frequency-descending order, enabling it to maintain as much prefix sharing as possible among the patterns in the transaction database. However, the construction of such an FP-tree requires two database scans, which is the major limitation of using an FP-tree approach for handling a data stream.

Inspired by the performance of the FP-growth algorithm, several approaches for processing a data stream using an FP-tree or a similar tree-based data structure have been proposed. However, most of these approaches either produce approximate results [8,17,30] or discover a special subset (e.g., closed, maximal, and constraint-based) [6,23,42,20] of frequent patterns. Only a few of them provide exact results. DSTree [22], which is a prefix-tree that is built based on a canonical item order, has recently been proposed for mining an exact set of recent frequent patterns over a data stream using a sliding window technique. In DSTree, the sliding window consists of several batches of transactions, and the transaction information for each batch is explicitly maintained at each node in the tree structure. To discover the complete set of recent frequent patterns, the FP-growth mining technique is applied to the DSTree of the current window. Even though the construction of the DSTree requires only one scan of the data stream, it cannot guarantee that a high level of prefix sharing (like in FP-tree) will be achieved in the tree structure because the items are inserted into the tree in a frequency-independent canonical order. Moreover, upon sliding of window the tree update mechanism of DSTree may leave some 'garbage' nodes in the tree structure when the mining request is delayed. In addition, the mining phase is time-inefficient because of the structure of the DSTree.

Processing recent data to mine complete set of exact frequent patterns from data stream, therefore, requires constructing a memory-efficient summary data structure that should be built with only one scan over the stream. Using the highly compact FP-tree faces the limitation of the requirement of two database scans. Again, although the DSTree can be constructed with a single-pass over the data stream it fails to obtain the compactness which is its primary limitation to achieve both memory and time efficiency. Furthermore, efficiently deleting the old data and inserting the newly arrived data from and into the summary tree structure are other challenging issues. Till now, no specific research results an efficient approach for the above problems though it is essentially required in stream data processing. Therefore, this begs the question: *is it possible to construct a prefix-tree structure that captures the stream data with one scan over a data stream, has the compactness of an FP-tree, efficiently updates for the old and the new data, offers optimized use of memory, and yields better mining performance for finding the complete set of recent frequent patterns*? Motivated from these requirements, in this paper, we propose a novel tree structure, called the Compact Pattern Stream tree (CPS-tree) that can efficiently address all of the above. Using this tree, we can construct an FP-tree-like compact prefix-tree structure with one scan of the data stream and provide the same mining performance as the FP-growth technique through use of an efficient tree restructuring process.

The main concept behind our CPS-tree construction is that the tree structure is periodically reorganized in a frequency-descending item order after inserting part of stream data (i.e., some transactions in the stream) in the previous item order. Through repeated reorganizations, the CPS-tree can maintain as much prefix sharing as possible in a prefix-tree with a single scan over a data stream, and provides better mining performance as a result. Similar to DSTree, we use the sliding window technique to find the complete set of recent frequent patterns by decomposing each window into batches of fixed numbers of transactions. However, instead of maintaining the batch information at each node of the tree structure (as DSTree does), we introduce the novel concept of maintaining it only at the last node of each path. Moreover, CPS-tree can efficiently update the batch information in a tree when the window slides. Once the CPS-tree is constructed, we use the FP-growth mining technique to generate the complete set of exact (not approximate) frequent patterns for a data stream from the current sliding window.

Tree reorganization overhead is likely to be substantial, because our CPS-tree size has to be large enough to hold all transaction information from the current sliding window. Thus, we need to devise an efficient tree restructuring algorithm and verify that it incurs negligible overhead compared to the gain in overall runtime. Our comprehensive experimental results for both real and synthetic datasets show that frequent pattern mining in a data stream using the CPS-tree outperforms the existing state-of-the-art algorithms in terms of runtime and memory efficiency; we also show that tree restructuring can be accomplished with an acceptably small cost.

In summary, the main contributions of this work are as follows:

- To find the complete set of recent frequent patterns over a data stream in both memory and time efficient manner we propose a novel, highly compact tree structure called CPS-tree that can maintain stream transactions for the current sliding window with a single scan.
- We introduce a new technique of periodic tree restructuring mechanism that improves the degree of prefix sharing as much as possible in the tree structure dynamically, thereby reducing tree size and increasing mining efficiency.
- We introduce the novel concept of maintaining frequency count lists at the last node (i.e., the last item of a transaction), which can further reduce the size of the prefix-tree.
- We observe the applicability of our CPS-tree in landmark and damped window models.
- We observe the performance characteristics of our proposed technique through extensive experimental analyses.

The rest of the paper is organized as follows. In Section 2, we summarize the existing algorithms for frequent pattern mining from a data stream. Preliminary knowledge is presented in Section 3. Section 4 describes the structure and restructuring process of the CPS-tree in detail. We also discuss the effectiveness of different tree restructuring methods and investigate the performance characteristics of the CPS-tree in this section. Section 5 summarizes some additional functionalities of the CPS-tree. We report our experimental results in Section 6. Finally, Section 7 concludes the paper.

2. Related work

Mining frequent patterns in static [1,11,13,34,36] and incremental [33,43,21] databases has been well-addressed over the past decade. The first frequent pattern mining algorithm was *Apriori* [1] which was proposed in 1993 to find association rules [1,12,31,37,38,5,19,35] among patterns. This technique finds the frequent patterns of length k from the set of already generated candidate patterns of length $k-1$. The main performance limitations of *Apriori*-like approaches result from the requirement for multiple database scans and a large number of candidate patterns, many of which prove to be infrequent after scanning the database. To overcome these problems, Han et al. [11] proposed the frequent pattern tree (FP-tree) and the FP-growth algorithm; this algorithm reduces the number of database scans by two and eliminates the requirement for candidate generation. Introduction of this highly compact FP-tree structure led to a new avenue of research with regard to mining frequent patterns with a prefix-tree structure. However, the static nature of an FP-tree and the requirement for two database scans limit the applicability of this algorithm to frequent pattern mining over a data stream.

General issues and research issues associated with frequent pattern mining over a data stream were reviewed in [7] and [15], respectively. However, because the scope of this work includes mining a data stream using a sliding window mechanism, we provide a thorough literature review focusing primarily on studies related to window-based approaches.

Most studies about finding frequent patterns in a data stream are based on the landmark window model [25,41,44] or the sliding window model [3,6,22,27,29,24]. The first attempt to mine frequent patterns over the entire history of streaming data was proposed by Manku and Motwani [28]. They developed two single-pass algorithms, Sticky-Sampling and Lossy Counting, both of which are based on the *anti-monotone*[1] property; these algorithms provide approximate results with an error bound. Zhi-Jun et al. [44] used a lattice structure, referred to as a frequent enumerate tree, which is divided into several equivalent classes of stored patterns with the same transaction-*ids* in a single class. Frequent patterns are divided into equivalent classes, and only those frequent patterns that represent the two borders of each class are maintained; other frequent patterns are pruned. DSM-FI [25] is another algorithm that was developed to mine frequent patterns using a landmark window. Every transaction is converted into k (the total number of items in the transaction) small transactions and inserted into an extended prefix-tree-based summary data structure called the *item-suffix frequent itemset forest*. Giannella et al. [8] developed an FP-tree-based algorithm, called FP-stream, to mine frequent patterns at multiple time granularities using a novel titled-time window technique. When a new batch of transactions arrives, the algorithm processes the stream data using an FP-growth technique. The major limitation of this batch-by-batch processing approach is that when a stream flows, the FP-stream needs to build an FP-tree to capture the stream contents of each new batch.

Mining recent frequent patterns using the sliding window technique has also been studied in the literature. Lin et al. [27] presented a method to mine a data stream for frequent patterns using a time-sensitive sliding window i.e., the window size is defined by a fixed period of time. In this approach, the incoming stream within a window time period is divided into several batches, and frequent patterns are mined in each batch individually. Using a discounting mechanism, the method discards the old patterns by using an approximation table that provides the approximate counts of the expired data items. Chang and Lee [3] proposed *estWin* that finds recent frequent patterns adaptively over an online transactional data stream using the sliding window model. This algorithm requires the minimum support threshold and another parameter termed the *significant support* to adaptively maintain the approximate frequent patterns window after window.

Most of the methods discussed above find approximate frequent patterns [8,17,28,32,41,40] with an error bound or additional pruning threshold. Very few techniques [4,22,24,29] find the exact set of recent frequent patterns from a data stream. In [24], the authors proposed an *Apriori*-based algorithm, called MFI-TransSW, which finds complete set of recent frequent

[1] Any subset of a frequent pattern must be also frequent [1].

patterns by using bit-sequences to keep track of the occurrence of all items in the transactions of the current fixed-sized sliding window. To remove old data and to reflect the inclusion of new data it performs a bit-wise left-shift operation for all bit-sequences. This approach is based on transaction-sensitive sliding window where the bit-sequence update operation is performed at the arrival of every single transaction. The MFI-TransSW applies the level-wise candidate-generation-and-test methodology to find the complete set of recent frequent patterns from the current window. Therefore, it suffers from the *Apriori* [1] limitation of huge candidate pattern generation, especially when mining stream data that contain large number of and/or long frequent patterns, and/or with lower support thresholds. Furthermore, the transaction-by-transaction update mechanism may limit its performance when stream flows at high speed. Again, since the approach maintains the bit-sequence information in full for all items in the window, it fails to achieve memory efficiency when the window contains large number of transactions and distinct items, which is very common in data stream environment. Even though MFI-TransSW discovers recent frequent patterns from a data stream, it differs significantly from the proposed technique in both mining approach and data processing strategy.

The algorithm most closely related to that proposed in our study is DSTree [22], which discovers exact frequent patterns from a data stream. DSTree uses a sliding window mechanism in which the window is divided into a fixed number of equal-sized, non-overlapping batches of transactions. A canonical ordered prefix-tree structure is used to store the current window information. Each node in the tree maintains a list to explicitly store its frequency count in each batch. To avoid tree traversal during extraction of the old batch information from the tree, DSTree keeps track of the last visited batch at each node by using an additional pointer to the last updated batch number. To reflect the sliding of window it shifts the contents of frequency lists in the nodes. Upon a mining request, after capturing the information for a full window, an FP-growth-based mining technique is used to mine the complete set of frequent patterns from the tree.

The DSTree, however, has several limitations. First, because it stores items in canonical order, it does not guarantee a highly compact tree structure, which is essential when handling stream data to avoid massive storage overhead, to reduce the search space, and to accelerate the FP-growth-based mining operation. Second, DSTree maintains the batch information (i.e., frequency count) in each node with the help of a list of frequency counts; this list at each node further increases the tree size. Third, another storage overhead issue with DSTree is the need to maintain an extra batch pointer at each node to indicate the last-visited batch for this node. Fourth, as described in [22], during the tree update phase DSTree does not visit all nodes in the tree (i.e., it avoids traversing the whole tree) and does not perform shifting the frequency counts list at each node in practice. Therefore, it does not perform the frequency list update operation for the nodes which are not visited during the new incoming batch. Such tree update operation may leave some invalid nodes in the DSTree structure for the current window. For example, consider a window consists of 2 (two) batches and the current window (say, W_1) contains batches 1 and 2. Upon sliding of window, batch 1 becomes obsolete and the new batch 3 flows in (i.e., W_2 consisting of batches 2 and 3 appears). Also, consider nodes Y and Z in the DSTree constructed for W_1 appear in only batch 1, and in only batch 2, respectively. Therefore, the batch pointers of Y, and Z will point to batch 1, and batch 2, respectively. And the frequency count list (consisting of two fields) for Y contains the frequency value of Y in the first field (for batch 1) and 0 (zero) in the last field (for batch 2), and that for Z contains value 0 (zero) in its first field (for batch 1) and the frequency value of Z in the last field (for batch 2). Now, let us assume the situation (i) both Y and Z appear in batch 3, or (ii) none of Y and Z does appear in batch 3. In the case of (i) the batch pointers and the frequency count lists of Y and Z are updated accordingly in the DSTree for W_2, because both nodes are visited in batch 3. However, in the case of (ii), since it does not visit all nodes to delete nodes or shift respective frequency count lists, the DSTree in W_2 will retain Y, which is not a valid node in W_2, and will maintain incorrect information in the frequency list of Z, even though Z is a valid node in W_2. Therefore, DSTree in W_2 may carry over some expired information that produces some 'garbage' nodes. Such 'garbage' nodes in the current DSTree will obviously increase the tree size. Fifth, due to the enormous number of 'garbage' nodes, the total number of nodes in the tree may become unmanageable if the knowledge in the stream changes over time. Sixth, to ensure exact and consistent mining, before executing the mining algorithm, the tree needs to be 'cleaned' to get rid of the 'garbage' nodes (i.e., deleting all the nodes that are no longer valid for the current window and then adjusting the tree structure to update the frequency count list), which may be a costlier task than traversing the tree once to refresh its content after each batch. Therefore, the size and mining efficiency of a DSTree are highly dependent on the distribution of data in transactions, because the frequency-independent item ordering of the tree structure may not provide as much prefix sharing as possible among all the paths in the tree structure. As a result, DSTree is likely to have a much extended mining time compared to tree structures arranged according to frequency-descending item ordering. Furthermore, DSTree was constructed based on the assumption of no main memory limitation, which is unrealistic when handling very large amounts of data, such as data streams.

In contrast, our CPS-tree maintains exactly the same information about the stream data as DSTree does by storing it in an FP-tree-like highly compact tree structure, and, thereby, ensures a more storage-efficient data structure. The highly compact tree structure offers an efficient FP-growth-based mining platform. Further storage efficiency is achieved by maintaining the list of frequency counts only at the last node of each path representing a transaction, instead of keeping this information at each node. Moreover, the CPS-tree does not need the extra batch pointer at each node to keep track of the last update. The CPS-tree constantly updates itself by extracting the expired transactions after each slide of the window, guaranteeing that the tree will not contain 'garbage' nodes and constantly ensuring a ready-to-mine tree status.

3. Preliminaries

In this section, we provide definitions of key terms that explain the concepts of frequent pattern mining over a data stream more formally. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of literals, called items, that have been used as a unit of information in an application domain. A set $C = \{i_l, \ldots, i_k\} \subseteq I, l \leqslant k$ and $l, k \in [1, n]$, is called a pattern/itemset, or a q-itemset if it contains q items. A transaction $t = (tid, D)$ is a tuple where tid is a transaction-id and D is a pattern. If $C \subseteq D$, it is said that t contains C or C occurs in t. Let $size(t)$ be the size of t, i.e., the number of items in D.

A data stream DS can formally be defined as an infinite sequence of transactions, $DS = [t_1, t_2, \ldots, t_m)$, where $t_i, i \in [1, m]$ is the ith arrived transaction. A window W can be referred to as a set of all transactions between the ith and jth (where $j > i$) arrival of transactions and the size of W is $|W| = j - i$, i.e., the number of transactions between the ith and jth arrival of transactions. Let each window, W, be composed of a number of equal-sized non-overlapping transaction batches, called panes.[2] Therefore, each window is composed of a fixed number of same-sized panes. If there are m transactions and n panes (where m mod $n = 0$) in a W, then each pane consists of m/n transactions, therefore, the size of each pane is $|m/n|$. Let the window slide pane-by-pane i.e., each slide of the window introduces a new pane and removes the oldest pane from the current window.

The support of a pattern C in a W, denoted as $Sup_W(C)$, is the number of transactions in W that contain C. Therefore, a pattern is called frequent in W if its support is no less than an absolute minimal support threshold min_sup ∂, with $0 \leqslant \partial \leqslant |W|$ given by the user. Given $DS, |W|$, and a min_sup, finding the complete set of patterns in W that have support count of no less than the min_sup value is the problem of mining the exact set of recent frequent patterns, F_W, in the data stream using the sliding window mechanism.

4. CPS-tree: design, construction, and mining

In this section, we first introduce our CPS-tree and then describe efficient pane extraction mechanisms using CPS-tree. We explain how CPS-tree periodically reorganizes itself to produce a frequency-descending tree structure with the help of dynamic tree restructuring methods. We also analyze the mining performance that CPS-tree achieves due to its dynamic tree restructuring mechanism.

The FP-growth [11] technique requires two database scans to achieve a highly compact frequency-descending tree structure (i.e., an FP-tree). During the first scan, the algorithm derives a list of frequent items in a frequency-descending order. The second database scan then, according to this list, produces an FP-tree that stores transaction information involving frequent patterns in a compact fashion. Although the requirement for two database scans might not be a handicap for mining a static dataset, it can be a limitation when mining stream data. In contrast, CPS-tree builds an FP-tree-like compact frequency-descending tree structure with a single-pass of the stream data. At first, transactions in the data stream are inserted into the CPS-tree based on a predefined item order (e.g., lexicographical item order). The item order of the CPS-tree is maintained by a list, called the *I-list*, with the respective frequency count of each item. After inserting some transactions, if the item order of the *I-list* deviates significantly from the current frequency-descending item order, the CPS-tree is dynamically restructured by the current frequency-descending item order and the *I-list* updates the item order with the current one.

As mentioned earlier, the CPS-tree is designed for mining frequent patterns from stream data using a window-based mechanism, we decompose a sliding window into several equal-sized non-overlapping panes, one for each batch of transactions in the data stream. We maintain the pane-wise information separately in our CPS-tree. Therefore, when the window slides, it is easier to update the tree by removing the expired pane transactions and, at the same time, inserting transactions in the new incoming pane.

4.1. Structure of the CPS-tree

Before discussing the construction process for the CPS-tree, we provide a brief description of its structure. A CPS-tree consists of one *root* node referred to as "*null*", a set of item-prefix sub-trees (children of the *root*), and an *I-list*, consisting of each distinct item with a relative frequency count and a pointer pointing to the first node in the CPS-tree carrying the item. Like FP-tree, CPS-tree contains nodes representing an itemset and the total number of passes (i.e., support) for that itemset in the path from the *root* up to that node in the current window. We propose the novel concept of maintaining a list of pane-based support count information only at the last item-node (i.e., *tail-item*) for a transaction, instead of including it at every node in the tree (as DSTree does). We call such a list the *pane-counter*. Therefore, based on the above discussion, a *tail-item* can be defined as follows.

Definition 1 (*tail-item*). Let $\{i_1, i_2, \ldots, i_n\}$ be a transaction in a data stream with items sorted according to a predefined sort order. The item represented by i_n (i.e., the last item of a sorted transaction) is defined as the *tail-item* of that transaction. For example, if any lexicographically-sorted transaction $\{a, b, c\}$ contains items '*a*', '*b*', and '*c*', then '*c*' is the *tail-item* of the transaction. However, if this transaction is arranged in reverse lexicographic order of items, like $\{c, b, a\}$, then '*a*' will be the *tail-item* of the transaction.

[2] Pane-based handling of sliding window data was introduced in [26].

Hence, two types of nodes can be maintained in a CPS-tree: ordinary nodes and *tail-node*s. The former are the types of nodes that are used in the FP-tree, whereas the latter can be defined as follows:

Definition 2 (*tail-node*). Let $t = \{i_1, i_2, \ldots, i_n\}$ be a sorted transaction, where i_n is the *tail-item*. If t is inserted into a CPS-tree in this order, then the node of the tree that represents item i_n is defined as a *tail-node* for t. For example, if the lexicographically-sorted transaction $\{a, b, c\}$ is inserted into a CPS-tree, then the node that represents item '*c*' (i.e., the *tail-item* of the transaction) is a tail-node in the tree for that transaction.

Lemma 1. *Every leaf node in a CPS-tree must be a tail-node.*

Proof. According to Definition 1, each leaf node in a CPS-tree must represent a *tail-item*. Therefore, every leaf node must be a *tail-node*. □

Like FP-tree, each node in a CPS-tree explicitly maintains parent, children, and node traversal pointers and a support counter to record the total frequency of the node in the path. In addition, each *tail-node* maintains a *pane-counter*. The structures of an ordinary node and a *tail-node* are shown in Fig. 1.

For an ordinary node: the structure is $N(s)$, where N is the item name of the node and s is a counter to record the total support count of that item in the path in the current window.

For a *tail-node*: $N(s; [v_1, v_2, \ldots, v_n])$, where N is the item name of the node, s is a counter to record the total support count of that node in the current window, and an entry in *pane-counter*, $v_j, j \in [1,\ n]$, is the support count of the node as a *tail-node* in pane j in the current window (n is the number of panes in a window). The value v_1 refers to the support count for the oldest pane and that for the latest pane is recorded in v_n.

Whenever a new *tail-node* in pane i of the current window of size n panes is created in the tree, its *pane-counter* is initialized by keeping the support value for pane i in the ith counter and the remaining $n - 1$ counters are initialized by 0 (zero). For example, if a *tail-node* '*a*' first appears in the second pane of the current window of four panes, the structure of the node will be $a_{1;0,1,0,0}$; where the value of $s = 1$ and the contents of the *pane-counter* are 0, 1, 0, 0.

Therefore, from the above definitions and the CPS-tree's node structure, we can deduce the following lemma.

Lemma 2. *A tail-node in the CPS-tree inherits an ordinary node, but not vice versa.*

Proof. The definition of an ordinary node states that it explicitly maintains a support counter to record the total support count of the node in the path, and three types of pointers: a parent pointer, a list of child pointers, and a node traversal pointer. Like an ordinary node, a *tail-node* explicitly maintains all such information. Moreover, it maintains the *pane-counter*, which is additional information to maintain. Therefore, there is an ordinary node in every *tail-node* and, in contrast, no *tail-node* in an ordinary node, because the *pane-counter* is not maintained in an ordinary node. □

4.2. Construction of the CPS-tree

Because the CPS-tree dynamically restructures itself to maintain the frequency-descending order, its construction process consists of two phases; *insertion* and *restructuring*. The *insertion* phase captures the stream content into the tree according to the current sort order of the *I-list*. The *restructuring* phase restructures the tree in a frequency-descending order by utilizing the information already obtained. The tree building starts with an *insertion* phase and ends with a *restructuring* phase. However, these two phases are repeatedly executed several times during the process of tree construction. Due to the dynamic nature of stream data, switching from the *insertion* phase to the *restructuring* phase can be performed after capturing the pane information or dynamically analyzing the data distribution in the stream.

We use an example to illustrate the construction of a CPS-tree from stream data. Fig. 2 shows a data stream (Fig. 2a) with corresponding transaction *ID*s, and a step-by-step construction procedure for a CPS-tree (Fig. 2b–f). Let the pane size say, p (i.e., the number of transactions in a pane) and the window size say, w (i.e., the number of panes in a window) both be two (i.e., there are two transactions in each pane and one window consists of two panes). Therefore, each window will contain

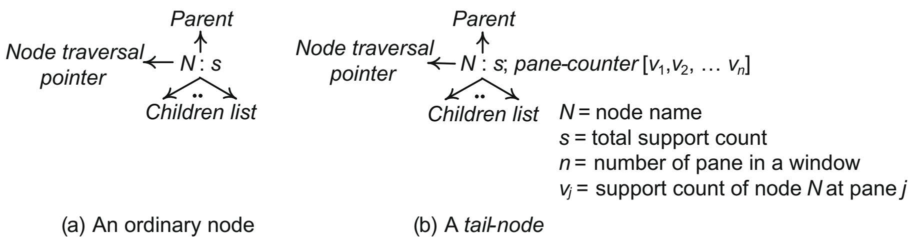


Fig. 1. Nodes of a CPS-tree: basic structure.

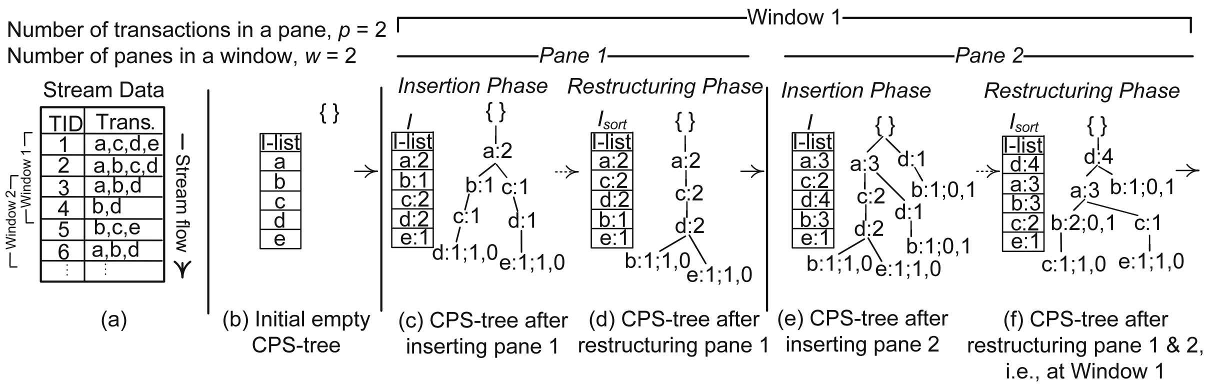

Number of transactions in a pane, p = 2
Number of panes in a window, w = 2
Window 1
Pane 1
Pane 2
Insertion Phase
Restructuring Phase
Stream Data
TID Trans.
1 a,c,d,e
2 a,b,c,d
3 a,b,d
4 b,d
5 b,c,e
6 a,b,d
Window 1
Window 2
Stream flow
I-list
I
I sort
(a)
(b) Initial empty CPS-tree
(c) CPS-tree after inserting pane 1
(d) CPS-tree after restructuring pane 1
(e) CPS-tree after inserting pane 2
(f) CPS-tree after restructuring pane 1 & 2, i.e., at Window 1

Fig. 2. Construction of the CPS-tree.

four transactions. For simplicity of description, we assume in this example that the tree is restructured after inserting each pane. However, in the following part of this section, we discuss other restructuring criteria that may be effective depending on the applications and the dynamic nature of the stream data. We also assume that the tree construction begins with inserting the transactions of the first pane in a predefined item order (e.g., lexicographical item order). Initially, the CPS-tree is empty (i.e., starts with a '*null*' *root* node), as shown in Fig. 2b. Our technique follows the FP-tree construction technique to insert a sorted transaction into the tree. To simplify figures, we do not show the node traversal pointers in trees; however, they are maintained in a similar manner to an FP-tree. We denote the frequency-independent and the frequency-descending *I-lists* by I and I_{sort}, respectively.

The construction of a CPS-tree starts with an *insertion* phase. The first *insertion* phase begins with inserting the transactions {*a, c, d, e*} and {*a, b, c, d*} of pane 1 into a tree in lexicographical item order. Fig. 2c illustrates the exact structures of the tree and *I-list* after inserting the first pane. According to the lexicographic item order, '*e*' and '*d*' are two *tail-items* of the two transactions of pane 1. Therefore, two *tail-nodes* '*e*:1;1,0' and '*d*:1;1,0' additionally maintain the *pane-counter* while the other nodes, as ordinary nodes, only keep the total support count for it in their respective paths. Since there are two panes in a window, each *tail-node*, in its *pane-counter*, registers two count values, one for each pane. The first fields of both *pane-counters* contain the support value for pane 1, and the other field of the list is initialized at zero. In addition, *tail-nodes* record their total counts separately in the respective paths.

Since the tree will be restructured after every pane, the first *insertion* phase ends here and the first *restructuring* phase starts immediately. Because the initial items are not inserted in frequency-descending order, the CPS-tree at this stage is a frequency-independent tree with a lexicographic item order. To rearrange the tree structure, the item order of I is first rearranged in a frequency-descending order to obtain I_{sort}, and then the tree is restructured according to I_{sort}. Fig. 2d shows the changes in *I-list* and the restructured CPS-tree after the first *restructuring* phase. The CPS-tree restructuring mechanisms have been discussed later in this section. Because of the restructuring operation, the status of some nodes in the tree may change from ordinary node to *tail-node* or vice versa. For instance, node '*b*:1' becomes a new *tail-node* while in contrast, *tail-node* '*d*:1;1,0' is converted to an ordinary node. However, these changes in the node status of the tree do not affect any property of the CPS-tree structure. In Section 4.4, we discuss two theorems (Theorems 1 and 2) and explain how such changes in node status can be implemented without affecting the properties of the CPS-tree. Note that after the restructuring operation, items with higher count values are arranged at the upper-most portion of the tree. Therefore, the CPS-tree at this stage is a frequency-descending tree that offers higher prefix sharing among patterns in tree nodes. The first *restructuring* phase terminates when the full processes of *I-list* rearrangement and tree restructuring have been completed. The CPS-tree construction will then enter into the next *insertion* phase.

During the next *insertion* phase, all transactions will be inserted following the new sort order of the *I-list {a, c, d, b, e}* instead of the previous item order {*a, b, c, d, e*}. The updated CPS-tree and the modified status of the *I-list* after passing the second *insertion* phase (i.e., inserting transactions in pane 2) are shown in Fig. 2e. The tree may again lose its property of compactness due to insertion of new transactions in the second *insertion* phase. Therefore, it is restructured again to maintain its frequency-descending property. The next *restructuring* phase will result in a new I_{sort} and CPS-tree structure as in Fig. 2f. Completion of this *restructuring* phase also terminates the construction of the CPS-tree for the first window. The algorithm to construct a CPS-tree is given in Fig. 3, where the *restructuring* phase is called at each pane.

Through the construction mechanism, we observe the following properties of a CPS-tree:

Property 1. *The total frequency count of any node in a CPS-tree is greater than or equal to the sum of the total frequency counts of its children.*

Property 2. *The total frequency count of any tail-node in a CPS-tree is greater than the sum of the total frequency counts of its children.*

```
Algorithm 1 (Construction of a CPS-tree for a data stream)
Input: Stream_data, Pane_size, Window_size, Initial_Sort_Order
Output: $T_{sort}$: a CPS-tree for the current window
Method:
    Begin
1:      w ← ϕ;
2:      T ← a prefix-tree with null initialization;
3:      Current_Sort_Order ← Initial_Sort_Order;
    //For the first Window
4:          While (w ≠ Window_size) do
5:              Call Insert_Pane(T);                                    // Insertion Phase
6:              Current_Sort_Order ← Frequency-descending sort order;   // Restructuring Phase
7:              Restructure T;
8:              w = w+1;
9:          End While
    //At each slide of Window
10:         Repeat
11:             Delete the oldest pane information from T;              // Extracting the old pane
12:             Call Insert_Pane(T);                                    // Insertion Phase
13:             Current_Sort_Order ← Frequency-descending sort order;   // Restructuring Phase
14:             Restructure T;
15:         End
    End

Insert_Pane(Tr)
    Begin
1:      p ← ϕ;
2:      While (p ≠ Pane_size) do
3:          Scan transaction from the current location in Stream_data;
4:          Insert the scanned transaction into Tr according to Current_Sort_Order;
5:          p = p+1;
6:      End While
    End
```

Fig. 3. The CPS-tree construction algorithm.

After sliding the window, the CPS-tree is updated by extracting the old pane information and inserting a new pane from and into the tree (lines 10 to 15 in Fig. 3). In the next subsection, we focus on the technique used to update the tree structure of the CPS-tree during stream flow.

4.3. Extracting old pane information

To provide a ready-to-mine platform and to refresh the tree structure with the exact contents of the current window, the CPS-tree is updated by traversing the whole tree every time we slide the window. To delete the expired information from the tree, it is sufficient to update only the *pane-counter* at each *tail-node* and, if necessary, reflect such changes at other nodes in the respective path, because only the *tail-node*s maintain the pane information.

The basic logic underlying the pane extraction mechanism of the CPS-tree is presented in the CPS-tree refreshing algorithm in Fig. 4. The CPS-tree refreshing operation starts by updating the *pane-counter* of each *tail-node* starting from the bottom item in the *I-list* (line 1). The first value in the *pane-counter* of each *tail-node* of the item is removed, and the remaining values in the list are shifted left by one slot to reflect the expiration of the oldest pane (line 9). Moreover, an extra entry, initialized by 0 (zero), is added at the last slot of the list to make room for storing the support of the new incoming pane (line 9). Furthermore, to keep the overall support for that node updated, the value of the total count (i.e., value of s) is also decremented at the same time by the support value (if it is not zero) in the expired pane (lines 4 and 5), and it is reflected to the support count of each node up to the *root* in the path (lines 12 and 13). During this update, any *tail-node* becomes an ordinary node if it contains zeros for all entries in its pane-counter, and the node is deleted if its total count value becomes zero (lines 6 and 7). Similarly, if support of any ordinary node in the path becomes zero during the update operation, the node is also deleted from the tree (lines 14 and 15). However, during the tree update process, no operation is performed on any ordinary node (line 3) in the tree.

We revisit the example of Fig. 2 to illustrate the expired pane handling mechanism of CPS-tree. Consider the stream data in Fig. 2a and the CPS-tree constructed for the first window as in Fig. 2f. Let the oldest pane (i.e., *tid*s 1 and 2) expire and a new pane (i.e., *tid*s 5 and 6) appear due to sliding of the window. We use the *I-list* to facilitate a fast tree traversal and then, as discussed above, we start from the bottom of the list to minimize the number of node visits. Therefore, the tree update mechanism starts by visiting and processing all the nodes of the bottom-most item '*e*' in the *I-list* one after the other. Following the node traversal pointer, we reach the first '*e*:1;1,0' node in the tree which is a *tail-node*. The value in the first slot in the

```
Algorithm 2 (Extracting old pan information)
Input: $T_{sort}$: a CPS-tree for the previous window
Output: T: a prefix-tree ready to capture new pane information
Method:
    Begin
1:      For each item i from the bottom of $I_{sort}$
2:          For each node $N_i$ for i in $T_{sort}$
3:              If ($N_i$ is a tail-node)
4:                  If ($N_i.v_1 > 0$)
5:                      $N_i.s = N_i.s - N_i.v_1$;
6:                      If ($N_i.s = 0$)
7:                          Delete $N_i$;
8:                      Else
9:                          Shift each $v_j$ of $N_i$ to left for one slot and insert a zero at the last entry;
10:                     End If
11:                     Update $I_{sort}$ for $N_i$;
12:                     For each node $N_k$ between $N_i$ and the root
13:                         $N_k.s = N_k.s - N_i.v_1$;
14:                         If ($N_k.s = 0$)
15:                             Delete $N_k$;
16:                         End If
17:                         Update $I_{sort}$ for $N_k$;
18:                     End For
19:                 End If
20:                 Shift each $v_j$ of $N_i$ to left for one slot and insert a zero at the last entry;
21:             End If
22:         End For
23:     End For
    End
```

Fig. 4. The pane extraction algorithm for the CPS-tree.

pane-counter of the node (i.e., 1) is deleted and at the same time, the same value is decremented from the total count, *s* (i.e., 1) of the node, resulting in zeros for all entries of the node. Therefore, node '*e*' is deleted from the tree. The support count of each node from this node up to the *root* is also decremented by the same amount. Furthermore, the immediate next node toward the *root* in the path, '*c*:1', is also deleted, because its total support count becomes zero and those for the other nodes (i.e., '*a*' and '*d*') are updated accordingly. To reflect these changes in the *I-list*, it is also updated following each update operation. Upon completing the *I-list* adjustment, the update process for item '*e*' is terminated, because there is no further node for '*e*' in the tree. The resultant CPS-tree is shown in Fig. 5a. The next item '*c*' has only one node, '*c*:1;1,0', in the tree (Fig. 5a), and it is also a *tail-node*. Therefore, we use a similar technique as described for '*e*' to update the tree. Fig. 5b reflects this change in the tree structure. The first node of the next item '*b*' may be '*b*:1;0,1' which is the child of node '*a*:1' is also a *tail-node*. Since the first value of its *pane-counter* is zero (Fig. 5b), transaction(s) represented by this *tail-node* did not appear in the first pane. Therefore, the total count values for this node and other nodes towards the *root* in the path do not need to be updated. However, the *pane-counter* of the node is updated by left shifting each value and inserting a zero at the end of the list to store information for the new upcoming pane. After finishing the operation, the contents of the total count and *pane-counter* of the node will be 1 and 1,0 respectively, as shown in Fig. 5c. The next node of item '*b*' is also a *tail-node* with the same information as the previous node. Therefore, it is also processed in the same fashion. The next item '*a*' has one node '*a*:1' in the tree, which is an ordinary node. So, no operation will be performed for this item. Skipping item '*a*', we get '*d*', which also has only one ordinary node ('*d*:2') in the tree. We also skip this item like the previous one and terminate the tree update operation, because there is no item left in the *I-list*. The final CPS-tree after extracting pane 1 is shown in Fig. 5c; this tree is ready in an updated form to capture the new upcoming pane information according to the CPS-tree construction mechanism.

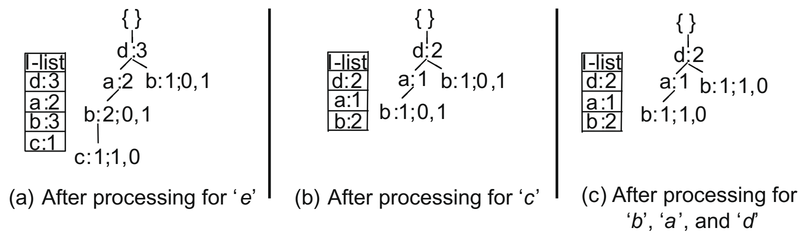


Fig. 5. Extracting old pane from CPS-tree of Fig. 2f.

At first glance, it may appear that the tree traversal cost might not be negligible in a CPS-tree. However, the tree traversal and update cost reported in the experimental result in Section 6 is found to be very similar to the cost of 'garbage' node deletion in a DSTree, and in most cases, it is smaller.

Fig. 6a shows the CPS-tree structure after inserting the new pane information (i.e., *tid*s 5 and 6) into the updated tree of Fig. 5c. After inserting the new pane, the tree is restructured in frequency-descending sort order, as shown in Fig. 6b. Therefore, based on the CPS-tree construction and pane extraction techniques discussed above, a CPS-tree holds the following lemmas:

Lemma 3. *Given a window size $|W|$ on a data stream DS, without considering the root node, the size of a CPS-tree is bounded by* $\sum_{t\in|W|}|size(t)|$.

Proof. Based on the CPS-tree construction process, each transaction t in $|W|$ contributes at best one path of $size(t)$ to a CPS-tree. Therefore, the total size contribution of all transactions in $|W|$ is at best$\sum_{t\in|W|}|size(t)|$. However, because there are usually many common prefix patterns among the transactions, the size of a CPS-tree is normally much smaller than $\sum_{t\in|W|}|size(t)|$. □

Lemma 4. *Given a stream database DS, and a current window W, the complete set of all item projections of $|W|$ transactions of W can be derived from the CPS-tree constructed on W.*

Proof. Based on the CPS-tree construction mechanism, each item projection in each transaction in W is mapped to only one path in the CPS-tree and any path from the *root* up to a *tail-node* maintains the complete projection for exactly n transactions (where n is the summation of all entries in its *pane-counter*). Moreover, the expired pane extraction mechanism of the CPS-tree ensures that the tree does not maintain any outdated information. Therefore, the CPS-tree maintains a complete set of all item projections of $|W|$ transactions of W only once. □

Lemmas 3 and 4 highlight how compactly and completely the CPS-tree captures the stream data in the current window. The compactness and completeness of the CPS-tree enables mining of F_W without any loss, because there is no requirement for any approximation or error bound. Moreover, mining the CPS-tree can be delayed until needed, because the tree constantly maintains a ready-to-mine platform at each window with exact and accurate recent information. Once the CPS-tree is constructed, we can employ the FP-growth mining technique on it to find F_W based on Properties 1, 2, and Lemma 4.

The DSTrees constructed (in lexicographic order) for the same stream data as Fig. 2a for Windows 1 and 2 are shown in Fig. 7a and b, respectively. The CPS-trees (in Fig. 2f for Window 1 and in Fig. 6b for Window 2) require significantly fewer nodes. Similar to the *tail-node*s of the CPS-tree, every node in the DSTree, as shown in the figures, records the count at each pane (shown in a similar fashion to the *pane-counter*). Upon insertion of new pane information, the contents of the list of any node are left-shifted. Each node in the DSTree also maintains the last update pointer (shown by the subscript value at each node in the figures) indicating the last visited pane number for that node. Unless a mining operation is requested, DSTree does not update the tree structure (i.e., does not visit the tree to reflect the current information), which may cause the tree to become overloaded with some 'garbage' nodes that are no longer valid for the current window. For example, there are five 'garbage' nodes in the DSTree of Window 2 in Fig. 7b that are carried over from the first window due to not removing expired pane information (i.e., pane 1) from the tree. The burden of such 'garbage' nodes may be severe when the mining request is delayed for several windows or there are concept drifts in the stream data. Consequently, during mining, the DSTree must utilize additional computation overhead either to delete the 'garbage' nodes or to pay 'attention' to ignore them. The former enables the tree structure to represent the exact content of the current stream. The latter, in contrast, may lead to inconsistent tree structures as the stream flows. Since the major goal is to mine frequent patterns, we focus on the former to ensure that the tree structure has the highest possible compactness and consistency.

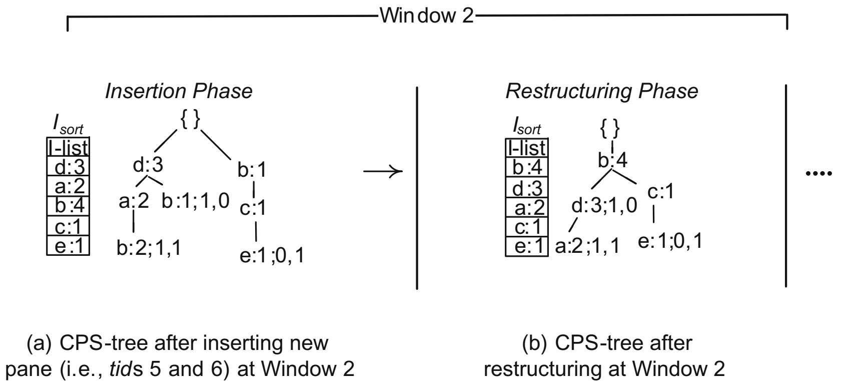


Fig. 6. The CPS-tree in Window 2 for the stream data of Fig. 2a.

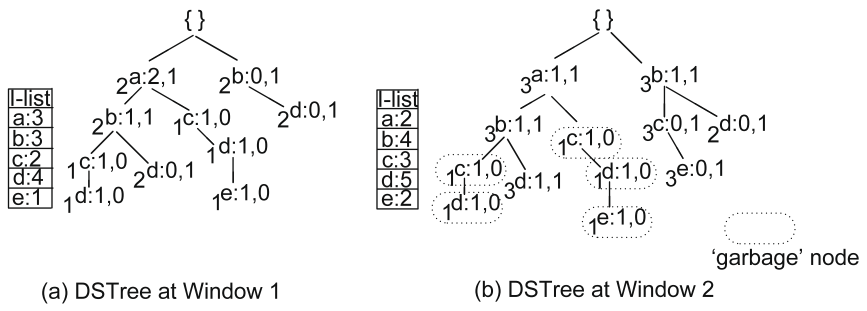


Fig. 7. DSTree for the stream data of Fig. 2a.

Based on the above CPS-tree construction process and the DSTree structure, it can be claimed that the storage efficiency that the CPS-tree achieves is due to (i) its highly compact tree structure, (ii) maintenance of a *tail-node*-based *pane-counter*, (iii) the avoidance of 'garbage' nodes, and (iv) getting rid of the last pane pointer at each node. Moreover, the frequency-descending sort order of the dynamically restructured CPS-tree enables it to achieve as much prefix sharing as possible among transactions in the current window. We discuss the tree restructuring techniques that the CPS-tree uses in the next subsection.

4.4. CPS-tree restructuring

Because the CPS-tree dynamically rearranges its structure in a frequency-descending item order, use of an efficient tree restructuring mechanism is important to reduce the overall tree restructuring overhead. The concept behind tree restructuring is to rearrange the nodes of an existing prefix-tree built based on an *I-list* order, *I*, to another order of items in *I-list*, I_{sort}. This operation involves both rearranging the items in the *I-list* and reorganizing the tree nodes (i.e., restructuring the tree structure). To restructure our CPS-tree, we use existing tree restructuring mechanisms, namely the branch sorting method (BSM) [34], and the path adjusting method [16]. Parameters such as the distribution of data in the stream and the size of the tree to be restructured are the factors that most influence the choice of an appropriate method. We refer interested readers to [34] and [16] for details of the BSM and the path adjusting method, respectively. However, we do provide a rough sketch of the methods in the following discussion. Both of these methods were proposed to handle tree nodes with only one count value (i.e., like ordinary nodes in a CPS-tree). Hence, we modify these methods for our CPS-tree to tackle the *tail-node* and *pane-counter* parameters. Let us for simplicity assume that the tree based on an old unsorted *I-list* i.e., *I*, and the tree based on a frequency-descending *I-list* i.e., I_{sort}, are denoted by *T* and T_{sort}, respectively.

The branch sorting method (BSM): During restructuring, the first operation of the branch sorting method is to rearrange the items in *I* in frequency-descending order to obtain I_{sort}. Then, based on I_{sort}, T is restructured. BSM is an array-based technique that restructures all branches, one-by-one, from the *root* of *T*. Each sub-tree under each child of the *root* is treated as a branch. Therefore, *T* contains as many branches as the number of children it has under the *root*. Each branch may consist of several paths and several *branching node*s (i.e., nodes having more than one child). While restructuring a branch, BSM sorts each path in the branch according to I_{sort} by removing it from the tree, sorting it into a temporary array, and again inserting it into the tree in sorted order. Finally, the restructuring mechanism terminates when all the branches in *T* are processed, which produces the final T_{sort}. However, during processing, it may be more likely that a path in *T* is found already arranged in I_{sort} order. Such paths are called *sorted paths*. During restructuring, if any path is found to be a *sorted path*, we not only skip the sort operation for that path, but also propagate this status information about the path to all *branching nodes* in the same path of the whole branch, indicating that the path from each *branching node* up to the *root* is sorted. Therefore, during sorting the other remaining paths in the same branch, only the sub-paths from the leaf node to the *branching node* of the *sorted path* are checked to determine whether the sub-path contains any item below (according to I_{sort}) the *branching node*. If no such item is found, we keep the sub-path from the *root* to the *branching node*, which is the common prefix path, as it is and only process the sub-path from the *branching node* to the leaf. Otherwise, the whole path from the leaf to the *root* is sorted. When all the sub-paths from the *branching node* are adjusted, we move to the next available *branching node*. To process (i.e., to sort) both *I* and any unsorted path in *T*, BSM uses a merge sort technique.

One may assumed that the BSM restructuring technique may require double computation since it removes a path from the tree structure and inserts it into the tree again in sorted order. BSM's time complexity, however, is much less than that would be expected for removing and inserting each transaction of the current window from and into the tree. Each path is processed with its total support count, that is, BSM eventually sorts all identical transactions in the current window together with a single operation. Furthermore, the *sorted path* handling mechanism enables BSM to further reduce the processing cost when the path is already sorted, because no operation is performed on any such path. Moreover, the information carried in each *branching node* in a *sorted path* may significantly cut down the time requirement by considering the processing cost only for the sub-path instead of the whole path.

Therefore, while restructuring a CPS-tree with the help of BSM, we need to reformulate BSM in such a way that the *pane-counter* information available at the *tail-node* is maintained in each path of the new restructured tree without any loss. Due to the new order of items in a path, the old *tail-item* may no longer be a *tail-item*, and another item in the path may become the new *tail-item*. We developed the following theorem to allow such an adjustment to be made consistently.

Theorem 1. *Let $P = \{a_1, a_2, \ldots, a_n\}$ be a path in a CPS-tree sorted in any canonical sort order and a_n be the current tail-node of the path. Let node_name.pcount refer to the pane-counter of a tail-node. After restructuring P, if node $a_k, k \in [1, n-1]$ represents the new tail-item of P, then instead of a_n, a_k becomes the new tail-node of the path with the adjustment a_k.pcount = a_n.pcount.*

Proof. Let $t = \{i_1, i_2, \ldots, i_n\}$ be the transaction in the stream data represented by the path $P = \{a_1, a_2, \ldots, a_n\}$ in the tree where the *tail-node* a_n represents the *tail-item* i_n. If the sort order of items in t is changed, let t' be the new transaction containing the exact set of items of t in new sort order and $i_k, k \in [1, n-1]$ be the *tail-item* of t'. Because the information carried by a transaction is independent of the order of items in it, t and t' maintain exactly the same information. Thus, the *tail-item* of t and that of t' and their corresponding *tail-nodes* (i.e., a_n, and a_k, respectively) in the tree maintain exactly the same information. Therefore, a_k is the new *tail-node* of P. □

Based on the above discussion and theorem, we provide the BSM algorithm for our CPS-tree in Fig. 8.

The path adjusting method: The other tree restructuring method, the path adjusting method, was devised to rearrange the structure of an already constructed FP-tree to another new sort order of items as a result of a database update. In this method, the paths in a prefix-tree are adjusted by recursively swapping the adjacent nodes in the path unless the path completely achieves the new sort order. Thus, it uses a bubble sort technique to process the swapping between two nodes. The basic operation of the path adjusting method is that given T and its I, the first step is to find pair of items in I that must be

```
Algorithm 3 (Branch sorting method)
Input: T and I
Output: $T_{sort}$ and $I_{sort}$
Method:
    Begin
1:      Compute $I_{sort}$ from I in frequency-descending order using merge sort technique
2:      For each branch $B_i$ in T
3:          For each unprocessed path $P_j$ in $B_i$          // from the root up to a tail-node
4:              If $P_j$ is a sorted path
5:                      Process_Branch($P_j$);
6:              Else    Sort_Path($P_j$);
7:              End If
8:          End For
9:      End For
10:     Terminate when all the branches are sorted and output $T_{sort}$ and $I_{sort}$;
    End

Process_Branch(P)
    Begin
1:      For each branching node $n_b$ in P from the $leaf_l$ node          // from the tail-node
2:          For each sub-path from $n_b$ to $leaf_k$ with $k \neq l$
3:              If items of all nodes between $n_b$ and $leaf_k$ are at below of $n_b$ in the $I_{sort}$
4:                      P = sub-path from $n_b$ to $leaf_k$;
5:                      If P is a sorted path
6:                              Process_Branch(P);
7:                      End If
8:              Else    P = path from the root to $leaf_k$;
9:              End If
10:             Sort_Path(P);
11:         End For
12:     End For
    End

Sort_Path(Q)
    Begin
1:      Reduce the count of all nodes of Q in T by the total support of $leaf_Q$;
2:      Delete all nodes having count zero from Q;
3:      Using a merge sort technique, sort items in Q in an array according to $I_{sort}$ order;
4:      Insert sorted Q into T at the location from where it was taken;
5:      Assign the pane-counter of Q to the new tail-node;
    End
```

Fig. 8. Branch sorting method algorithm for the CPS-tree.

exchanged to get the I_{sort} order. Then, all the nodes of these two items in all respective paths in T and the same items in I are exchanged. While swapping two nodes, for example between a child node 'N_c' and a parent node 'N_p' in a path, when the support count of 'N_p' is greater than that of 'N_c', this method inserts a new node in the tree (with the same name as 'N_p') as a sibling of 'N_p'. In other words, 'N_p' is divided into two nodes of the same name e.g. 'N_{p1}' and 'N_{p2}'. 'N_{p1}' is assigned as the parent of node 'N_c' with the same count as 'N_c'. The remaining children nodes of 'N_p' (if any) are assigned to 'N_{p2}' with the remaining support count value of 'N_p'. Then, the swapping between 'N_{p1}' and 'N_c' is performed by exchanging all of their parent and child pointers. However, when the support counts of both 'N_p' and 'N_c' are equal, a simple exchange operation between them is enough. Nevertheless, after swapping, this method may need to repeatedly merge the two sibling nodes (and their child nodes) of the new parent node when they are found to be the same because of the exchange operation. Finally, T_{sort} and I_{sort} are constructed when no further pair of items in I are found for exchange. Therefore, this operation is performed in three phases: insertion (splitting the parent node), exchange (swapping between two nodes), and merging (adjusting the siblings).

As mentioned previously for BSM, to effectively use this method in a CPS-tree, special measures need to be taken when one or both of the two nodes involved in swapping is/are *tail-node*(s). In this case, we use the following theorem to exchange two nodes in a path.

Theorem 2. *Let node_name.name, node_name.count, node_name.pcount, and node_name.clist denote the name, total support count of a node, the pane-counter of a tail-node, and the list of all children nodes of a parent node, respectively. Let a_p be a parent node and a_c be a child node in the CPS-tree. Swapping between them can then be performed following any one of the cases given below:*

Case I: If a_p is a *tail-node* and a_c is an ordinary node:
Step 1: Split a_p into two nodes; one *tail-node*, say a_t, with a_t.count = a_p.count – a_c.count, a_t.pcount = a_p.pcount and a_t.clist = (a_p.clist – a_c), and one ordinary node, say a_o, with a_o.count = a_c.count and a_o.clist = a_c.
Step 2: Swap between a_o and a_c.
Case II: If a_p is an ordinary node, a_c is a *tail-node* and a_p.count > a_c.count:
Step 1: Split a_p into two ordinary nodes say a_o and a_{o1}, with a_{o1}.count = a_p.count – a_c.count, a_{o1}.clist = (a_p.clist – a_c), and a_o.count = a_c.count, a_o.clist = a_c.
Step 2: Swap between a_o.name and a_c.name; and a_o.clist and a_c.clist.
Case III: If a_p is an ordinary node, a_c is a *tail-node* and a_p.count = a_c.count:
Step 1: Swap between a_p.name and a_c.name; and a_p.clist and a_c.clist.
Case IV: If a_p and a_c both are *tail-nodes*:
Step 1: Follow Step 1 in Case A.
Step 2: Follow Step 2 in Case B.

Proof. For Case I: Because $a_p.pcount$ maintains information for the transaction(s) from the *root* up to node a_p, swapping between a_p and a_c will result in an inconsistent tree structure for the original stream data. However, according to Lemma 2, the ordinary node, a_o in a_p, preserves the information for all the transaction(s) in the path where a_p is not a *tail-node* (i.e., the transactions in which both a_p and any node in $a_p.clist$ participate). Thus, any ordinary node a_o, extracted from a_p with $a_o.count = a_c.count$ maintains exactly the same information about transactions as a_c does. Based on Property 2, the *tail-node* a_t keeps all other information on the path where a_c is not a child. Hence, swapping between a_o and a_c preserves the consistency of the tree.

For Case II and Case III: Because $a_c.pcount$ maintains information for the transaction(s) from the *root* up to node a_c, swapping between a_p and a_c will result in an inconsistent tree structure for the original stream data. However, according to Theorem 1, swapping between a_o and a_c (for Case II) or between a_p and a_c (for Case III) preserves the tree structure consistency.

For Case IV: The rationale for this case is the same as that for the previous cases. □

It is clear from the above discussion that the performance of the path adjusting method depends largely on the degree of disorder among the items between two *I-lists* I and I_{sort}, and the number of nodes in the path between two swapping nodes, since it uses a bubble sort technique and the swapping between two nodes in a path considers the cost of $O(n^2)$, where n is the number of nodes involved in the path between them. Therefore, the cost of swapping two nodes in a path increases exponentially with an increase in the number of intermediate nodes between them in the path. Moreover, this computation becomes more time-intensive when there are more paths in the tree requiring such adjusting. Hence, this method may require a large amount of time to adjust a tree with larger or a higher number of branches and/or with an *I-list* characterized by a higher degree of disorder among items compared to a sorted *I-list*. In contrast, BSM uses a merge sort approach to sort the nodes of any path and therefore, the degree of disorder is immaterial to the performance during sorting, because irrespective of data distribution, the complexity of merge sort is always $O(n\log_2 n)$ where n is the total number of items in the list. Moreover, the number of intermediate nodes in a path to be sorted also has only a minor effect in BSM. The *sorted path* handling mechanism can further reduce the number of sorting operations and/or the size of data to be sorted.

Therefore, in the path adjusting method, the more the tree is previously sorted, the better the final results. BSM, in contrast, performs comparatively better when tree size and transaction length are large and the two *I-lists* differ significantly. Consequently, from the above discussion, the path adjusting method is a better choice when the degree of disorder among items in two *I-list*s and tree size are low, while BSM is a better choice when the degree of disorder and tree size are large. Due to the dynamic nature of the stream data, the appropriate restructuring method for CPS-tree can be chosen dynamically by analyzing the rank displacement of items between I and I_{sort}, and switching from the insertion phase to the restructuring phase can occur after capturing information on each pane.

One may assume that CPS-tree has tree restructuring costs as part of its overall runtime overhead. However, we argue that tree restructuring is a technique that enables the CPS-tree to share as many prefix-paths as possible among all tree nodes, which is the key to performance improvement for consecutive mining operations. In our experimental analyses, shown in Section 6, we found that a CPS-tree achieves significant reduction in runtime through the tree restructuring operation, even though there is a small overhead for the tree restructuring cost during the tree construction phase. This improvement in overall runtime is due to the rapid mining phase, executed on the compact tree structure provided by our restructured CPS-tree. In the next subsection, we discuss mining with the CPS-tree and investigate why, using FP-growth mining, the CPS-tree achieves a greater mining gain than other frequency-independent tree structures.

4.5. Mining analysis

Our primary goal in constructing a CPS-tree is to obtain a significant improvement in mining performance. So far, we have shown (i) how a CPS-tree can be constructed in frequency-descending order with a dynamic tree restructuring mechanism to facilitate FP-growth-based mining, and (ii) how such a tree maintains a ready-to-mine platform at each window, thereby ensuring the mining of exact frequent patterns in the current window without any approximations or error bounds. We revisit the FP-growth mining approach and investigate in detail why use of the CPS-tree results in a significant improvement in mining performance.

The basic operations in FP-growth-based pattern growth mining approach are counting frequent items, constructing a conditional pattern-base for each frequent item, and constructing a new conditional tree from each conditional pattern-base. Frequent item counts are facilitated by the *I-list*, which contains each distinct item along with its respective support count. Based on the following lemma, the CPS-tree reduces the search space to find a frequent item from the whole *I-list* to only the last frequent item in the list.

Lemma 5. *Given a stream database DS, a current window W, and a CPS-tree constructed on it, for any value of support threshold* ∂*, the search space during mining* F_W *can be initiated from the bottom-most item in the I-list with a support count value* $\geqslant \partial$ *to the first item above it in the I-list.*

Proof. Let X be the bottom-most item with $Sup_W(X) \geqslant \partial$ in the *I-list* of a CPS-tree on W. Since the *I-list* of a restructured CPS-tree maintains the items in frequency-descending order, the support counts of all items under X are less than $Sup_W(X)$, assuring that no item under X can be frequent. Therefore, mining F_W for W can be carried out by considering only X and its upper items in the *I-list*. □

For each frequent item X in the *I-list*, a small conditional pattern-base is generated. Each conditional pattern-base consists of the set of transformed prefix-paths of X from the original tree. To construct the conditional pattern-base, only the prefix sub-paths of nodes labeled X in the tree need to be accumulated. The frequency count of every node in the prefix-path should carry the same count as either that in the corresponding node X if X is an ordinary node, or the total count in the corresponding node X if X is a *tail-node*. Therefore, based on following corollary, during pattern growth mining, itemset-based conditional tree construction for any itemset can always be performed by avoiding the deletion of *global infrequent nodes*[3] from the conditional pattern-base if the *I-list* is arranged in frequency-descending order.

Corollary 1. *Let T be a prefix-tree constructed on the stream data of window W and let I be its corresponding I-list. If X (an item in I) is frequent, then the conditional pattern-base of X does not contain any global infrequent node for any value of min_sup, if and only if I and T are arranged in frequency-descending item order.*

Proof. Let $X_1, X_2, \ldots, X_k, \ldots X_j, \ldots, X_n$ be the contents of I. Let $X_i, \forall i \in [1, n]$ be the item of reference. The conditional pattern-base for X_i must contain item(s) from X_1 to X_{i-1}. If I is arranged in frequency-descending item order, then $Sup_W(X_i) \geqslant Sup_W(X_{i+1}), \forall i \in [1, n-1]$. Therefore, $Sup_W(X_k) \geqslant Sup_W(X_j), \forall k \in [1, j-1]$and hence if X_j is frequent, then X_k cannot be a *global* infrequent item. However, if I is not arranged in such an order, the proposition $Sup_W(X_i) \geqslant Sup_W(X_{i+1}), \forall i \in [1, n-1]$ is not satisfied and, consequently, the conditional pattern-base for X_i may contain *global infrequent node*(s). □

The above discussion reveals that the search space for finding the next frequent item in a frequency-descending *I-list* is consistently one item (with $O(1)$), starting from the bottom-most item with support $\geqslant \partial$ to the top-most item in the list. Such item ordering also ensures a *global infrequent node*-free conditional pattern-base for each frequent item; this reduces the conditional pattern-base traversal cost and supports rapid construction of the corresponding conditional tree. In contrast,

[3] Item X is a *global infrequent item* in current window W if $Sup_W(X)$ in the *I-list* is less than the user-provided *min_sup* value. Any node in the tree structure representing X is a *global infrequent node*.

if the *I-list* is arranged in a canonical order, it is quite unlikely that all the frequent items for a given *min_sup* value will appear at the top of the list. Thus, in such an *I-list*, there is likely to be a mix of frequent and infrequent items. Therefore, searching such an *I-list* for the first frequent item to begin mining requires starting from the bottom of the list. Furthermore, while traversing the *I-list* it is not guaranteed that the immediate next item will be a frequent one. That means to find the next frequent item, a linear search method needs to be applied in an upward direction until the next frequent item is found, which incurs an additional computation cost. Therefore, because conditional pattern-bases maintain the same item ordering as their own *I-list*s, which are constructed following the main *I-list* order, they can contain *global infrequent nodes*. Thus, we need to consider extra costs for deleting these nodes from the tree and adjusting the tree structure.

Moreover, the order of items in the *I-list* also has a direct influence on the size and total number of conditional databases constructed during the mining process. By using *I-lists* with a frequency-descending order, the size and number of conditional databases can be minimized. We start mining with the bottom-most, least frequent item in the *I-list*. This implies that the number of *frequent extensions*[4] of the item is much smaller, which in turn requires smaller and/or fewer conditional databases in subsequent mining. During mining in bottom-up manner, the items become increasingly frequent, but their *frequent extension* sets become smaller and the transactions in their conditional databases become shorter. This ensures that the number of conditional trees constructed in subsequent mining is not large. In contrast, if the bottom-most item is not the least frequent item, the number of *frequent extensions* of the item is usually larger. Under these circumstances, bigger and/or more conditional trees may need to be constructed in subsequent mining, which would increase the overall mining time significantly.

Therefore, for a particular window W and for any value of ∂, FP-growth mining from a frequency-descending tree would require less time than mining with other trees constructed in a frequency-independent canonical order. Motivated by this reasoning, we designed a frequency-descending tree (CPS-tree) with an efficient tree restructuring mechanism using only one scan of the stream data, in order to minimize mining time. Our experimental results, provided in Section 6, show that mining with a frequency-descending tree is multiple orders of magnitude faster than mining using a canonically-ordered tree. These results also indicate that the performance gain we achieve using our CPS-tree is predominantly due to a significant gain in the mining phase; the performance gain makes the tree restructuring overhead negligible.

5. Additional functionalities of the CPS-tree

In this section, we discuss the suitability of our CPS-tree in the context of a change in stream content and the detection of other pattern types. Furthermore, we discuss its applicability to other stream processing mechanisms.

5.1. Handling concept drifts

Most frequent pattern mining algorithms for stream data assume that any changes in the data distribution characteristics of the stream are unlikely to be dramatic. However, concept drifts are not uncommon in stream environments. Therefore, when sharp concept drifts occur in stream data, they either produce inaccurate results or incur high storage and runtime costs. Our CPS-tree handles concept drifts by dynamically capturing the data distribution in the stream and rearranging itself. The adaptive characteristics of the CPS-tree allow it to maintain compactness before and after the drifts occur. However, initial drifts may require extra computational time because of the sudden increase in tree size. In contrast, because of DSTree's canonical item-order processing, it may suffer from an unmanageably large number of nodes in progress before or after the occurrence of concept drift. Moreover, a change in data distribution will result in a large number of 'garbage' nodes if mining is requested only rarely.

5.2. Mining maximal and closed patterns

Due to the enormous number of frequent patterns that can be generated from stream data, researchers are sometimes interested in finding only the frequent maximal and closed patterns from stream data. Both these pattern types are subsets of a frequent pattern set where the frequent maximal pattern set is the set of patterns that do not have any frequent proper superset pattern. A frequent closed pattern set, in contrast, is the set of patterns where no proper superset of a pattern has the same frequency. Because our CPS-tree captures the complete information for a current window in a highly compact manner and constantly updates itself to remove expired data from the tree, the additional conditions required to mine these types of pattern sets can easily be realized. Therefore, these special pattern sets can be dynamically generated from our CPS-tree with stream data.

5.3. Compatibility with other window techniques

The dynamic nature of the construction mechanism allows other types of windows such as a tilted-time window or landmark window to be used in the CPS-tree. Tilted-time windows can be managed by weighting recent transactions more than

[4] *Frequent extensions* of X are a set of items where every member generates a frequent pattern with X.

older transactions. The transaction deletion mechanism (i.e., pane extraction mechanism) of the CPS-tree allows it to handle this type of window efficiently. Landmark windows are relatively simple to handle using a CPS-tree, as the pane expiration phase does not need to be considered. One possible way to make a tree restructuring decision, in this case, is to base it on the change of knowledge in the stream data. For both window cases, the CPS-tree provides a compact tree structure for further mining operations.

6. Experimental analyses

In this section, we present the results of our comprehensive experimental analyses of the performance of the CPS-tree over data streams for several real and synthetic datasets. Table 1 provides some statistical information about the datasets used in the experimental analyses. The first three datasets were obtained from [18]. *BMS-POS* contains several years worth of point-of-sale data from a large electronics retailer, while the other two datasets contain several months worth of click-stream data from two e-commerce web sites. *Kosarak* is also a dataset of click-stream data from a Hungarian on-line news portal. *Connect-4*, another real dataset, along with *Kosarak* was obtained from [2]. *T40I10D100K*, obtained from [14], is a synthetic dataset in which the parameters T, I, and D represent the average transaction size, average maximal potentially frequent patterns, and the number of transactions, respectively. The last column of Table 1 shows the percentage of total distinct items that appear in each transaction in each dataset. Because transactions in a database may vary in length (i.e., number of items in a transaction), we use the average transaction length in our calculations. For instance, 0.39% of the total distinct items appear in each transaction (on average) in the *BMS-POS* dataset while each transaction (on average) in the *Connect-4* dataset contains 33.33% of its total distinct items. This parameter provides a measure of whether the dataset is sparse or dense. In general, a sparse dataset contains fewer items per transaction and many distinct items in total. A dense dataset, in contrast, has many items per transaction with few distinct items. Therefore, when the value of this parameter for a dataset is relatively low (e.g., less than or equal to 10.0) we define the dataset to be sparse (as considered in the studies in [39,9]). Otherwise, it is considered a dense dataset. For example, *Connect-4* is a dense dataset while the others are sparse datasets (Table 1). The degree of sparsity or density of a dataset can also be measured using this value. The lower the value of this parameter in sparse datasets, the more sparse the dataset is, and vice versa. These statistics in Table 1 were used to describe some of the characteristics of the datasets, and we obtained consistent results for all datasets. However, due to space constraints, we report the experimental results for only some of the datasets in this section.

All programs were written in Microsoft Visual C++ 6.0 and run with Windows XP on a 2.66 GHz CPU with 1 GB memory. Runtime specifies CPU and I/Os and includes, unless otherwise specified, tree construction, tree restructuring (for the CPS-tree only), and mining time. The results shown in this section are based on the average of multiple runs for each case.

Before the comparison study, we tested the accuracy of our CPS-tree by comparing the frequent patterns discovered by the CPS-tree to those discovered by direct mining of the same set of transactions for a specific window. In all cases, the CPS-tree found exactly the same set of frequent patterns as the direct approach. This indicates that the CPS-tree can accurately mine the complete set of frequent patterns from the current window on a data stream. In the experiments, during construction of the CPS-tree, the tree restructuring operation was performed upon sliding of window and the appropriate tree restructuring method (either BSM or Path adjusting method) was chosen by dynamically monitoring the data distribution in the current stream data.

In this experiment study we compare the memory and runtime requirements of MFI-TransSW [24], DSTree [22], and our proposed CPS-tree. Unlike the pane-based window sliding approach as in DSTree and in CPS-tree, MFI-TransSW updates the window contents at the incoming of each new transaction. Therefore, it is not justified to compare MFI-TransSW's execution time with those of DSTree and CPS-tree. Again, in several studies in literature [9,11] it has been shown that for datasets where number of items is small, *Apriori* performs better, but its performance drops off rapidly in situations with a large number of frequent patterns, long patterns, and/or low minimum support thresholds. Such non-trivial cost of the approach is mainly dominated by its costly phase of handling a huge number of candidate sets [11]. This performance problem is overcome by the pattern-growth based approach when applied on a frequency-descending tree structure [11]. Therefore, even if we consider mining frequent patterns from a fixed-sized static window using both *Apriori*-based MFI-TransSW and CPS-tree, most of the cases CPS-tree will outperform MFI-TransSW. Hence, as we are concerning about pane-based sliding windows, we avoid comparing the runtime between MFI-TransSW and CPS-tree. However, we perform memory comparison between them, since memory requirement for a window is independent of the window sliding approach.

Table 1
Dataset characteristics.

Dataset	#Trans. (T)	#Items (I)	MaxTL (MTL)	AvgTL (ATL)	(ATL/I) $\times$ 100
BMS-POS	515,597	1657	164	6.53	0.39
BMS-WebView-1	59,602	497	267	2.50	0.51
BMS-WebView-2	77,512	3340	161	5.00	0.14
Connect-4	67,557	129	43	43.00	33.33
Kosarak	990,002	41,270	2498	8.10	0.02
T40I10D100K	100,000	942	77	39.61	4.20

It has been reported in [22] that DSTree, the approach mostly close to our CPS-tree, outperforms other related algorithms (e.g., FP-stream [8]) in finding the complete set of recent frequent patterns from the data stream. Therefore, in this paper, we mainly compare the performance of CPS-tree to DSTree. According to the original DSTree algorithm, the DSTree update mechanism avoids traversing the whole tree upon the sliding of window. So, for fair comparison we implemented DSTree without traversing the whole tree at each window slide, but updating only the nodes visited in the new incoming pane. However, for the sake of obtaining accurate mining result from DSTree, we refreshed it by deleting all 'garbage' nodes from the tree only before the mining operation.

We divide the experimental analysis into three parts. First, we show CPS-tree's compactness in terms of required memory and number of nodes. Second, we compare the performance of DSTree and CPS-tree from a runtime point of view. Finally, we evaluate the CPS-tree's performance according to window size.

6.1. Memory efficiency

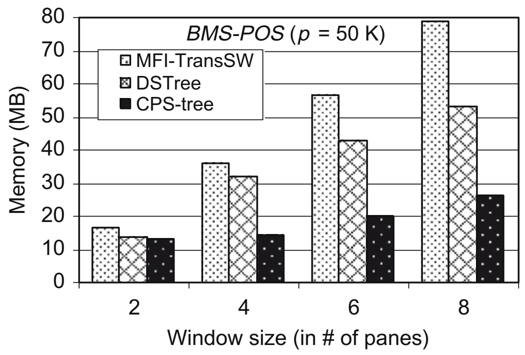
We conducted experiments to verify the memory requirements for MFI-TransSW, DSTree, and our CPS-tree on different datasets by varying the window size. Since all of these algorithms always capture the window contents in full (i.e., not depending of support threshold), the support threshold values do not influence on the required memory in each approach. Therefore, in this experiment, the reported memory requirements represent the size of the underlying data structure after capturing only the complete sliding window content when keeping the window size fixed. We also performed the test by varying the window size at each dataset. Fig. 9 shows the results of the experiment in several datasets in terms of physical memory requirement. The *x*-axes of the graphs represent the variation of window size in number of panes, while we kept the pane size (in number of transactions) fixed (as shown in each chart title) for each dataset. For example, Fig. 9a illustrates the results on *BMS-POS* when window sizes were kept fixed at 100 K $(= 50\text{ K} \times 2)$, 200 K $(= 50\text{ K} \times 4)$, 300 K $(= 50\text{ K} \times 6)$, and 400 K $(= 50\text{ K} \times 8)$ transactions, respectively, at each case. For each window size, we report the average memory required in all sliding windows.

The numbers of (distinct) items with the variation of window size for different datasets are presented in Table 2. From the second to the sixth columns of the table show the average item count in all sliding windows for each window size. In *BMS-POS*, for example, when the window size is 100 K (i.e., $p = 50$ K and $w_1 = 2$) transactions, the item count is on an average 1319 in all sliding windows. Again, in the same dataset 1573 items appear on an average in all sliding windows when the window size is fixed at 400 K (i.e., $p = 50$ K and $w_4 = 8$) transactions. For MFI-TransSW we calculate the memory for a window by only accumulating the bit-sequence sizes of all items in the window, although the algorithm requires some additional space to maintain other parameters such as item names and frequency count of each item. The bit-sequence size for an item in a window is exactly the same as the number of transactions in the window. Thus, for the window size of 100 K transactions in *BMS-POS*, MFI-TransSW requires around 16.4 MB (i.e., $100\text{ K} \times 1319$ bits) of memory, as reported in Fig. 9a. For the DSTree and our CPS-tree, Fig. 9 indicates the total size of respective tree structures on the variation of window size.

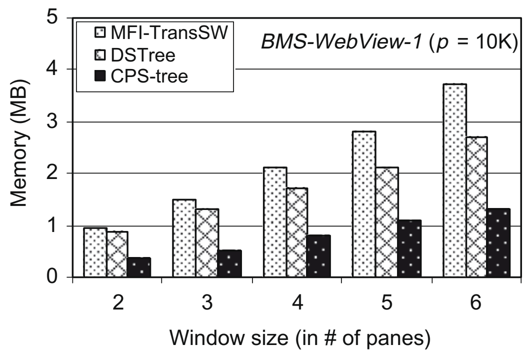
While comparing between MFI-TransSW and CPS-tree, we observe that for all window sizes in most of the datasets CPS-tree requires less memory; especially significant improvement in the sparse datasets. The more sparse the dataset is (refer to Table 1), the more gain CPS-tree achieves in required memory over MFI-TransSW. CPS-tree's memory gain over the MFI-TransSW is more prominent when the number of items and the window size are larger. For example, the size improvements of CPS-tree in *BMS-WebView-2* (Fig. 9c) and in *Kosarak* (Fig. 9e) for all window sizes are remarkable compared to other datasets. However, when the number of distinct items is small and/or the sparseness of sparse dataset is low CPS-tree may require comparatively higher memory. The results in *Connect-4* and *T40I10D100K* reflect such phenomena. But, the above tow phenomena are rare in handling stream data. Nevertheless, MFI-TransSW will require large amount of execution time to mine from dense dataset *Connect-4* with huge frequent patterns, and from dataset *T40I10D100K* with reasonably long patterns.

The improvement in memory consumption that CPS-tree achieves by using a frequency-descending item order compared with a tree constructed using a frequency-independent item order (e.g., DSTree) can be observed from Fig. 9, as well. The experimental results between these two tree structures for different datasets with fixed window size (denoted by the values of *p* and *w*) are also presented in the form of the number of nodes in Table 3. Table 3 shows the average total number of nodes in each tree structure and contains explicit counts of the average number of different types of nodes in all windows for both tree structures. *Tail-nodes* and ordinary nodes were considered for the CPS-tree, whereas regular nodes and 'garbage' nodes were considered for DSTree.

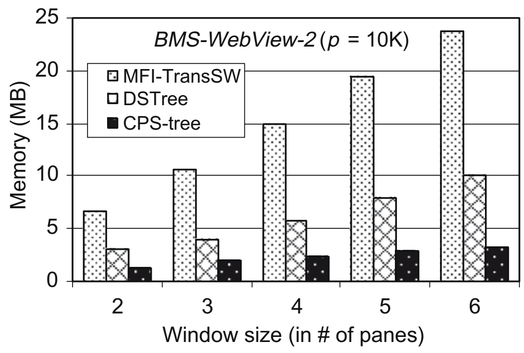
The total number of nodes required for CPS-tree was significantly lower than that for DSTree for all datasets (Table 3). The reason for this is that CPS-tree's dynamic tree restructuring phase enables it to obtain as much prefix sharing as possible, which results in a significant reduction in the number of nodes compared to frequency-independent trees. The compactness of the CPS-tree is more significant in dense datasets (e.g., *Connect-4*), due to the high level of association among patterns in such datasets, as also reflected in Fig. 9. The impact of 'garbage' nodes on the total number of nodes in DSTree is also indicated in Table 3. As shown in the last column of the table, irrespective of the type of dataset, DSTree handles an additional 16% to 25% of total nodes as 'garbage' nodes. However, initial windows are free from that impact so they are not taken into consideration in the 'garbage' node calculation. In contrast, CPS-tree is free from the 'curse' of such 'garbage' nodes. Moreover, the overall memory requirement of CPS-tree is reduced by maintaining only a few *tail-node*s (the only nodes keeping the *pane-counter* information) compared to the number of ordinary nodes. The second to the last column of Table 3 shows



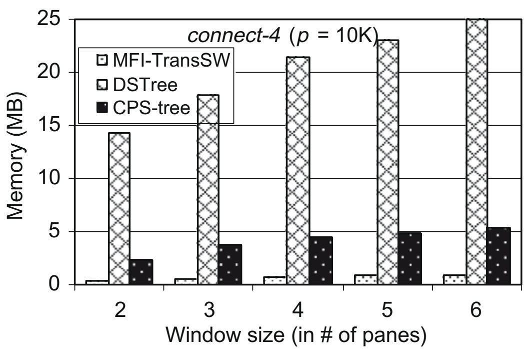
(a) On dataset *BMS-POS* with p = 50K



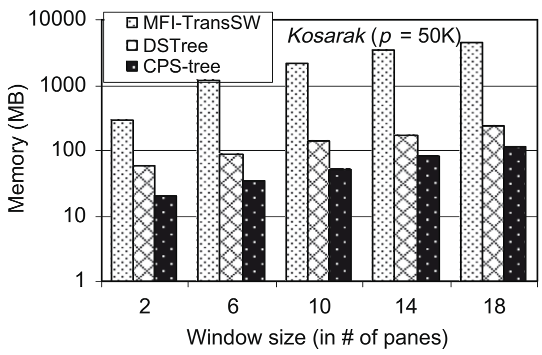
(b) On dataset *BMS-WebView-1* with p = 10K



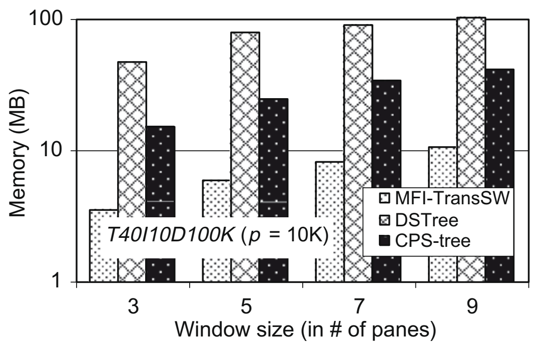
(c) On dataset *BMS-WebView-2* with p = 10K



(d) On dataset *Connect-4* with p = 10K



(e) On dataset *Kosarak* with p = 50K



(f) On dataset *T40I10D100K* with p = 10K

Fig. 9. Memory comparison.

Table 2
Number of distinct items on variation of window size.

Dataset with window parameters	Window size (in # of panes)				
	w_1	w_2	w_3	w_4	w_5
BMS-POS (p = 50 K) ($w_1 = 2, w_2 = 4, w_3 = 6, w_4 = 8$)	1319	1430	1500	1573	–
BMS-WebView-1 (p = 10 K) ($w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 5, w_5 = 6$)	380	400	425	450	497
BMS-WebView-2 (p = 10 K) ($w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 5, w_5 = 6$)	2662	2807	2964	3090	3163
Connect-4 (p = 10 K) ($w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 5, w_5 = 6$)	126	127	128	129	129
Kosarak (p = 50 K) ($w_1 = 2, w_2 = 6, w_3 = 10, w_4 = 14, w_5 = 18$)	23,375	31,349	35,318	38,138	40,298
T40I10D100K (p = 10 K) ($w_1 = 3, w_2 = 5, w_3 = 7, w_4 = 9$)	941	942	942	942	–

Table 3
Node distribution in CPS-tree and DSTree (# of nodes).

Dataset with window parameters	Tree structure	Tail-node (T)	Ordinary node (O)	'Garbage' node (G)	Regular node (R)	Total node (TN)	(T/TN) × 100	(G/TN) × 100
BMS-POS (p = 50 K, w = 4)	DSTree	–	–	153,140	781,918	935,057		16
	CPS-tree	128,147	488,307	–	–	616,454	21	
BMS-WebView-1 (p = 10 K, w = 4)	DSTree	–	–	11,266	49,521	60,787		19
	CPS-tree	12,559	24,630	–	–	37,189	34	
BMS-WebView-2 (p = 10 K, w = 4)	DSTree	–	–	27,407	150,053	177,460		15
	CPS-tree	26,287	93,961	–	–	120,248	22	
Connect-4 (p = 10 K, w = 2)	DSTree	–	–	122,914	363,965	486,879		25
	CPS-tree	19,593	95,668	–	–	115,261	17	
Kosarak (p = 50 K, w = 4)	DSTree	–	–	262,553	1,331,386	1,593,939		16
	CPS-tree	135,901	926,104	–	–	1,062,005	13	
T40I10D100K (p = 10 K, w = 4)	DSTree	–	–	361,257	1,774,634	2,135,891		17
	CPS-tree	39,987	1,400,350	–	–	1,440,337	3	

the percentage of *tail-nodes* in the CPS-tree according to the total number of nodes in each dataset. In the synthetic *T40I10D100K* dataset, only 3% of total nodes were *tail-node*s. In other datasets, we also found that *tail-node*s made a relatively small contribution, ranging from 13% to 34%.

As discussed above, the amount of information maintained in a *tail-node* in a CPS-tree is equal to or less than that maintained in a regular node in DSTree. Therefore, based on the number of nodes provided in Table 3, we can infer that CPS-tree is more memory efficient than DSTree for almost all types of datasets. Furthermore, Fig. 9 supports our claim that the physical memory requirement of DSTree for datasets with different characteristics or of MFI-TransSW for most of the datasets is much higher than that of CPS-tree. The experimental results presented in the next subsection highlight the runtime performance gain of CPS-tree due to its compact tree structure.

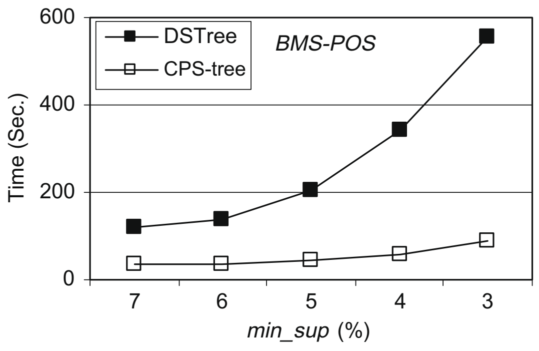
6.2. *Runtime efficiency*

We compared the overall runtime efficiency of CPS-tree and DSTree for mining the exact set of frequent patterns from the current window based on changes in the *min_sup* value. The results are shown in Fig. 10. We also investigated the reason for CPS-tree's performance improvement. Table 4 reports the runtime distribution of DSTree and CPS-tree for two *min_sup* values (one high and one low) using different datasets. Table 4 shows the runtime distribution for tree construction, tree restructuring (only for the CPS-tree), tree updating (expired pane deletion time for CPS-tree and 'garbage' node deletion time for DSTree), mining time (for the two *min_sup* values), and total time. The tree restructuring cost includes the cost of dynamically choosing the appropriate tree restructuring method. The time shown on the y-axis of each graph in Fig. 10 is the accumulated above-time distributions for both tree structures for different *min_sup* values. The figures and the columns of the table show the average time required in all active windows (i.e., windows after the initial one). The experiments were performed with a mining request in each window by varying the *min_sup* values for each dataset while the window parameters (i.e., w and p) were kept fixed at reasonably high values. We also evaluated CPS-tree's performance on the variation of window size (i.e., by varying the values of both w and p). The results are provided in the next subsection.

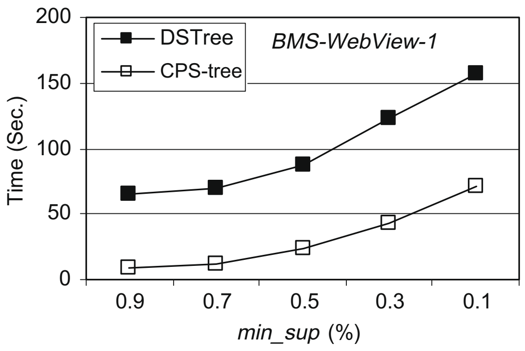
As shown in Fig. 10, the performance of CPS-tree and DSTree are similar for higher *min_sup* values in the datasets. However, the gap in performance between these algorithms increased when the *min_sup* value decreased. For most datasets, DSTree required a long runtime when either the *min_sup* value was low or the size of the frequent patterns set was large. CPS-tree, in contrast, produced results in a reasonable amount of time for both these cases. The results presented in Fig. 10 and in Table 4 clearly demonstrate that CPS-tree outperforms DSTree in terms of overall runtime by multiple orders of magnitude for both high and low *min_sup* values in all datasets analyzed. For example, for the sparse datasets *BMS-POS*, *BMS-WebView-1*, *BMS-WebView-2*, *Kosarak*, and *T40I10D100K*, DSTree required a 2- to 6-fold longer runtime than that required by CPS-tree. Again, the comparative performance gain of CPS-tree was much higher for dense datasets, as shown for *Connect-4*. In general, CPS-tree provided a higher degree of prefix sharing for dense datasets compared to sparse datasets; this facilitated CPS-tree's comparatively higher mining time gain in dense datasets.

Table 4 shows the reason for CPS-tree's runtime improvement over DSTree: a dramatic reduction in mining time due to the dynamically-obtained frequency-descending tree structure. Even though there is an additional tree restructuring cost for using CPS-tree, this cost is insignificant compared to the gain in mining obtained using CPS-tree (Table 4). Although it is possible that the tree restructuring cost for CPS-tree might become significant for high *min_sup* values, our experiments revealed that even with higher *min_sup* values and a low number of frequent patterns for datasets of different characteristics, CPS-tree still outperformed DSTree. For example, CPS-tree outperformed DSTree for both *T40I10D100K* and *Kosarak* datasets with *min_sup* values of 25% and 8%, respectively, even though both trees required a similar amount of runtime to generate a very small frequent pattern set of average size (for all windows) 1 and 14, respectively. These results show that CPS-tree outperforms DSTree when the *min_sup* values for datasets are large-enough to produce few frequent patterns.

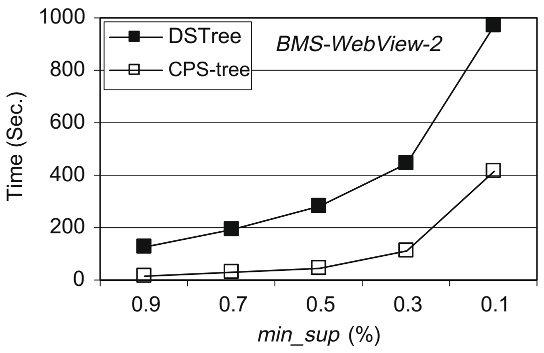
It can be noticed that, although both tree structures required similar amounts of tree construction time (Table 4), DSTree required a relatively longer time to update the tree at each window. This observation suggests that a tree traversal-based



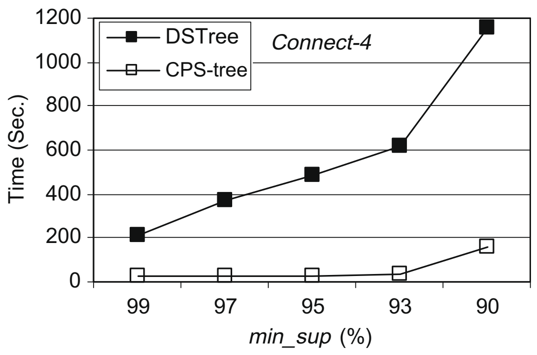
(a) On *BMS-POS* (p =50K, w = 3)



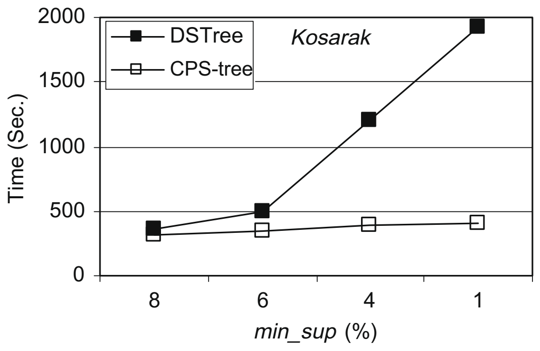
(b) On *BMS-WebView-1* (p =10K, w = 4)



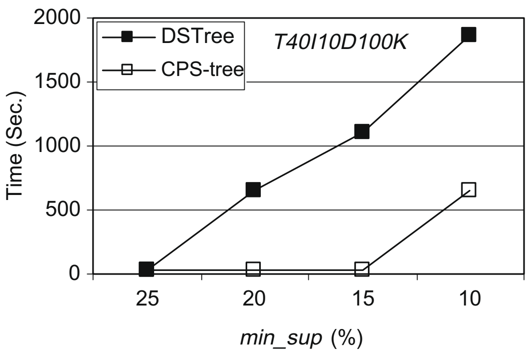
(c) On *BMS-WebView-2* (p =10K, w = 4)



(d) On *Connect-4* (p =10K, w = 2)



(e) On *Kosarak* (p =50K, w = 4)



(f) On *T40I10D100K* (p =10K, w = 4)

Fig. 10. Runtime comparison between CPS-tree and DSTree.

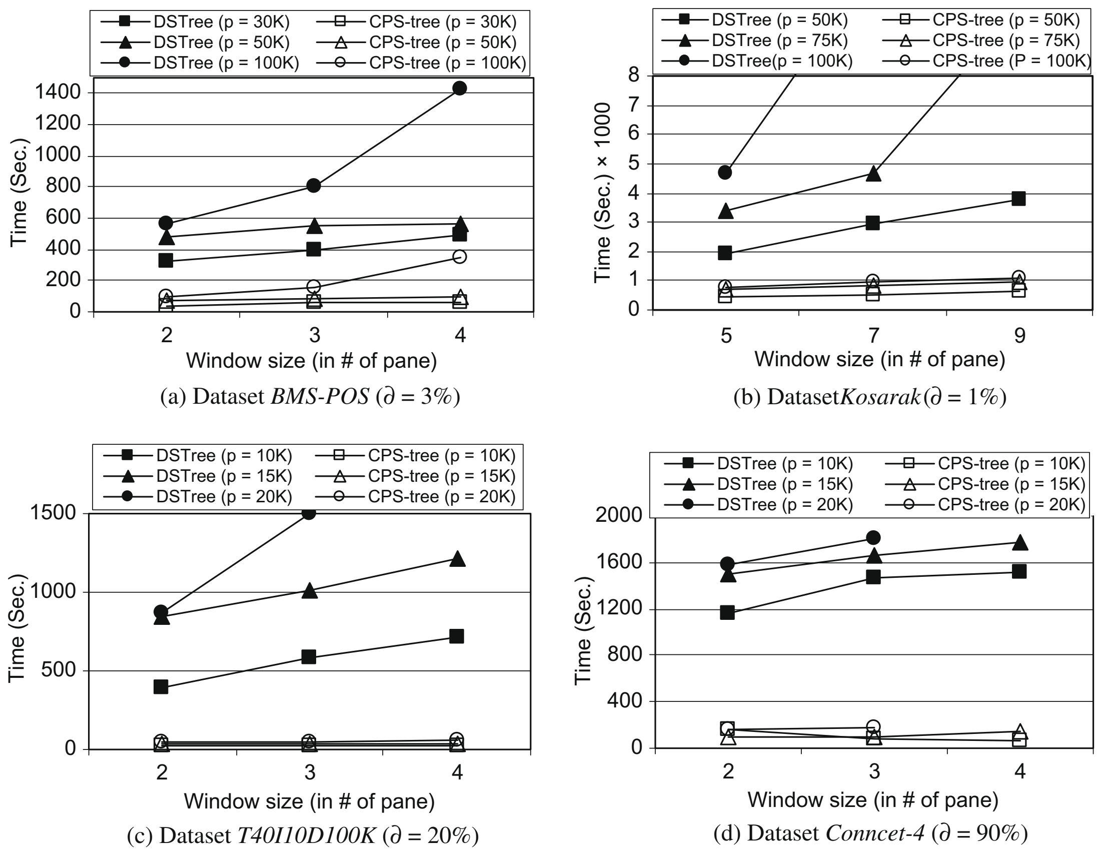
approach (CPS-tree) is better at extracting old pane information from the current window than an approach that uses pane update pointers for all tree nodes to delete all expired nodes (DSTree).

6.3. Window parameters

Because the CPS-tree can dynamically restructures itself after each window slide, its performance may vary depending on the window parameters w and p. Therefore, to determine the effect of changes in window parameters on the runtime of CPS-tree, we analyzed its performance by varying both these parameters over different datasets for different *min_sup* values. This experiment demonstrates the performance of CPS-tree when varying the window size in number of transactions, as well. In this subsection, we present the results for several sparse (*BMS-POS*, *Kosarak*, *T40I10D100K*) and dense (*Connect-4*) datasets, keeping the *min_sup* values fixed, but changing parameters w and p (i.e., window size). We do not report our results for *BMS-WebView-1* and *BMS-WebView-2*, because the results were similar to those obtained for *BMS-POS* due to similar dataset characteristics. The graphs presented in Fig. 11 show the results on *BMS-POS* for $\partial = 3\%$ (Fig. 11a), *Kosarak* for $\partial = 1\%$ (Fig. 11b),

Table 4
Runtime distribution (s).

Dataset with window parameters and *min_sup* (s)	Tree structure	Tree construction	Tree restructuring	Tree update	Mining time		Total time	
					∂_1	∂_2	∂_1	∂_2
BMS-POS	DSTree	25.27	–	3.41	108.43	528.24	137.11	556.92
($p = 50$ K, $w = 3$) $\partial_1 = 6\%, \partial_2 = 3\%$	CPS-tree	26.29	4.46	1.80	5.09	56.32	37.64	88.87
BMS-WebView-1	DSTree	2.54	–	0.21	70.01	154.03	72.75	156.77
($p = 10$ K, $w = 4$) $\partial_1 = 0.7\%, \partial_2 = 0.1\%$	CPS-tree	2.60	0.46	0.10	12.58	67.63	15.73	70.78
BMS-WebView-2	DSTree	11.20	–	0.76	180.31	956.03	192.27	967.99
($p = 10$ K, $w = 4$) $\partial_1 = 0.7\%, \partial_2 = 0.1\%$	CPS-tree	11.57	2.42	0.33	14.57	404.16	28.90	418.48
Connect-4	DSTree	16.96	–	1.82	195.93	1141.90	214.71	1160.68
($p = 10$ K, $w = 2$) $\partial_1 = 99\%, \partial_2 = 90\%$	CPS-tree	12.50	10.06	1.46	0.03	132.22	24.04	156.24
Kosarak	DSTree	301.19	–	9.03	52.00	1619.98	362.22	1930.20
($p = 50$ K, $w = 4$) $\partial_1 = 8\%, \partial_2 = 1\%$	CPS-tree	300.46	16.63	4.69	0.06	88.56	321.84	410.34
T40I10D100K	DSTree	20.72	–	9.32	0.01	1834.55	30.05	1864.59
($p = 10$ K, $w = 4$) $\partial_1 = 25\%, \partial_2 = 10\%$	CPS-tree	16.85	6.90	3.34	0.01	630.44	27.10	657.53



(a) Dataset *BMS-POS* ($\partial = 3\%$)
(b) Dataset*Kosarak*($\partial = 1\%$)
(c) Dataset *T40I10D100K* ($\partial = 20\%$)
(d) Dataset *Conncet-4* ($\partial = 90\%$)

Fig. 11. The performance of CPS-tree on variation of window parameters.

T40I10D100K for $\partial = 20\%$ (Fig. 11c), and *Connect-4* for $\partial = 90\%$ (Fig. 11d). The y-axes in the graphs represent the average total time (including construction time, tree restructuring time only for the CPS-tree, pane extraction time, and mining time) required in all active windows. While the x-axes show the variation in number of panes. Therefore, each graph illustrates the trends in execution time with the variation of window size. For example, in *BMS-POS* results are presented when window size varied from 60 K ($p = 30$ K, $w = 2$) to 400 K ($p = 100$K, $w = 4$) transactions. Again, for *Kosarak* window sizes were considered from 250 K ($p = 50$ K, $w = 5$) to 900 K ($p = 100$ K, $w = 9$) transactions.

Larger pane and window sizes (i.e., the more the transactions in a pane and the greater the number of panes in a window) resulted in a longer total tree construction time for both DSTree and CPS-tree. Furthermore, the larger the size and number of panes, the longer the time it took to restructure the CPS-tree. However, changes in window size in number of panes had a negligible effect on CPS-tree's overall runtime for the above datasets, as shown in the graphs in Fig. 11. This is because mining time has a far greater impact on the total runtime than tree construction or tree restructuring time. As observed in the previous experiment, CPS-tree outperformed DSTree by significantly reducing the mining time; the overall runtime required by CPS-tree is therefore small enough to handle larger windows in different datasets. For DSTree, in contrast, a sharp increase in runtime according to an increase in window size was observed. Therefore, in most cases, DSTree was not able to produce the resultant pattern set within a reasonable amount of time when the values of w and p were large (i.e., when window size was large). Thus, it was not possible to plot the runtime required by DSTree for all the data points in x-axes in (i) Fig. 11b for $p = 75$ K, and $p = 100$ K, and (ii) Fig. 11c for $p = 20$ K.

In sparse datasets, variation in window parameters does not have much impact on mining time because there is a relatively small change in the total number and average size of frequent patterns from window to window. In contrast, in dense datasets, the number and average size of frequent patterns may vary substantially for different window parameters. Hence, the performance gain of CPS-tree over DSTree for dense dataset (Fig. 11d) was found much promising than that for sparse datasets. Therefore, the results shown in Fig. 11 indicate clearly that CPS-tree can handle a variety of window parameters and produce the exact set of recent frequent patterns within a reasonable amount of time over datasets of various characteristics.

The above experiments demonstrate that despite the addition of a tree restructuring cost for CPS-tree, it outperforms other related algorithms in terms of both runtime and memory consumption in data streams of different characteristics, when used to mine an exact set of recent frequent patterns from a data stream. CPS-tree is broadly applicable, as shown by its performance gain compared to DSTree for various datasets (refer to Table 1). The key to CPS-tree's performance gain is the construction of a tree with a highly compact structure, achieved by periodically reorganizing the prefix-tree structure in frequency-descending order with the help of efficient and effective tree-restructuring mechanisms.

7. Conclusions

In this paper, we introduced a novel concept of dynamic tree restructuring to handle stream data. We proposed CPS-tree, an algorithm that uses tree restructuring to achieve a frequency-descending highly compact prefix-tree structure with a single-pass. CPS-tree remarkably reduces memory consumption and the mining time required to discover the exact set of recent frequent patterns from a high-speed data stream. To completely and efficiently perform the tree restructuring operation, we also introduced an efficient mechanism to restructure the tree constructed from stream data. We showed that despite its additional tree restructuring cost, CPS-tree's overall runtime was orders of magnitude greater than that of DSTree, a frequency-independent tree structure. We investigated the reason behind CPS-tree's performance gain as well. Furthermore, CPS-tree's easy-to-maintain features and adaptive characteristics should enable it to handle concept drifts in the incoming data stream efficiently.

Acknowledgements

We would like to express our deep gratitude to the anonymous reviewers of the paper. Their useful and constructive comments and suggestions have played a significant role in improving the quality of this work. This work was supported by a grant from Kyung Hee University in 2009 (KHU-20090759).

References

[1] R. Agrawal, T. Imielinski, A.N. Swami, Mining association rules between sets of items in large databases, in: Proc. the ACM SIGMOD Conference on Management of Data, 1993, pp. 207–216.
[2] C.L. Blake, C.J. Merz, UCI Repository of Machine Learning Databases, University of California-Irvine, Irvine, CA, 1998.
[3] J.H. Chang, W.S. Lee, estWin: Online data stream mining of recent frequent itemsets by sliding window method, Journal of Information Science 31 (2) (2005) 76–90.
[4] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, in: Proc. ACM SIGKDD Conference, 2003, pp. 487–492.
[5] G. Chen, Q. Wei, Fuzzy association rules and the extended mining algorithms, Information Sciences 147 (1–4) (2002) 201–228.
[6] Y. Chi, H. Wang, P.S. Yu, R.R. Muntz, Catch the moment: maintaining closed frequent itemsets over a data stream sliding window, Knowledge and Information Systems 10 (3) (2006) 265–294.
[7] M.M. Gaber, A. Zaslavsky, S. Krishnaswamy, Mining data streams: a review, ACM SIGMOD Record 34 (2) (2005) 18–26.
[8] C. Giannella, J. Han, J. Pei, X. Yan, P.S. Yu, Mining frequent patterns in data streams at multiple time granularities, in: Data Mining: Next Generation Challenges and Future Directions, AAAI/MIT Press, 2004 (Chapter 6).
[9] R.P. Gopalan, Y.G. Sucahyo, High performance frequent patterns extraction using compressed FP-tree, in: Proc. The SIAM International Workshop on High Performance and Distributed Mining, April 2004.
[10] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, in: Data Mining and Knowledge Discovery, 10th Anniversary Issue, 2007, pp. 55–86.
[11] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proc. the 2000 ACM SIGMOD International Conference on Management of Data, 2000, pp. 1–12.
[12] P.-Y. Hsu, Y.-L. Chen, C.-C. Ling, Algorithms for mining association rules in bag databases, Information Sciences 166 (2004) 31–47.
[13] T. Hu, S.Y. Sung, H. Xiong, Q. Fu, Discovery of maximum length frequent itemsets, Information Sciences 178 (2008) 69–87.
[14] IBM, QUEST Data Mining Project. <http://www.almaden.ibm.com/cs/quest>.
[15] N. Jiang, L. Gruenwald, Research issues in data stream association rule mining, ACM SIGMOD Record 35 (1) (2006) 14–19.
[16] J.-L. Koh, S.-F. Shieh, An efficient approach for maintaining association rules based on adjusting FP-tree structures, in: Y.-J. Lee, J. Li, K.-Y. Whang, D. Lee (Eds.), Proc. the DASFAA, 2004, pp. 417–424.
[17] J.-L. Koh, S.-N. Shin, An approximate approach for mining recently frequent itemsets from data streams, in: A.M. Tjoa, J. Trujillo (Eds.), Proc. DaWaK, 2006, pp. 352–362.

[18] R. Kohavi, C. Brodley, B. Frasca, L. Mason, Z. Zheng, KDD-Cup 2000 organizers' report: peeling the onion, SIGKDD Explorations 2(2) (2000) 86–98. <http://www.ecn.purdue.edu/KDDCUP>.
[19] A.J.T. Lee, R.W. Hong, W.M. Ko, W.K. Tsao, H.H. Lin, Mining spatial association rules in image databases, Information Sciences 177 (7) (2007) 1593–1608.
[20] A.J.T. Lee, C.-S. Wang, An efficient algorithm for mining frequent inter-transaction patterns, Information Sciences 177 (2007) 3453–3476.
[21] Y.-S. Lee, S.-J. Yen, Incremental and interactive mining of web traversal patterns, Information Sciences 178 (2) (2008) 287–306.
[22] C.K.-S. Leung, Q.I. Khan, DSTree: a tree structure for the mining of frequent sets from data streams, in: Proc. ICDM, 2006, pp. 928–932.
[23] C.K.-S. Leung, Q.I. Khan, Efficient mining of constrained frequent patterns from streams, in: Proc. 10th International Database Engineering and Applications Symposium, 2006.
[24] H.-F. Li, S.-Y. Lee, Mining frequent itemsets over data streams using efficient window sliding techniques, Expert Systems with Applications 36 (2009) 1466–1477.
[25] H.-F. Li, S.-Y. Lee, M.-K. Shan, An efficient algorithm for mining frequent itemsets over the entire history of data streams, in: Proc. International Workshop on Knowledge Discovery in Data Streams, 2004.
[26] J. Li, D. Maier, K. Tuftel, V. Papadimos, P.A. Tucker, No pane, no gain: efficient evaluation of sliding-window aggregates over data streams, SIGMOD Record 34 (1) (2005) 39–44.
[27] C.-H. Lin, D.-Y. Chiu, Y.-H. Wu, A.L.P. Chen, Mining frequent itemsets from data streams with a time-sensitive sliding window, in: Proc. SIAM International Conference on Data Mining, 2005.
[28] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proc. VLDB, 2002, pp. 346–357.
[29] B. Mozafari, H. Thakkar, C. Zaniolo, Verifying and mining frequent patterns from large windows over data streams, in: Proc. ICDE, 2008 pp. 179–188.
[30] M. Seno, G. Karypis, Finding frequent patterns using length-decreasing support constraints, Data Mining and Knowledge Discovery 10 (2005) 197–228.
[31] L. Shen, H. Shen, L. Cheng, New algorithms for efficient mining of association rules, Information Sciences 118 (1999) 251–268.
[32] C. Silvestri, S. Orlando, Approximate mining of frequent patterns on streams, Intelligent Data Analysis 11 (2007) 49–73.
[33] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, Y.-K. Lee, Efficient single-pass frequent pattern mining using a prefix-tree, Information Sciences 179 (2009) 559–583.
[34] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, Y.-K. Lee, CP-tree: a tree structure for single-pass frequent pattern mining, in: T. Washio et al. (Eds.), Proc. PAKDD, 2008, pp. 1022–1027.
[35] Y.J. Tsay, J.Y. Chiang, An efficient cluster and decomposition algorithm for mining association rules, Information Sciences 160 (1–4) (2004) 161–171.
[36] Y.-J. Tsay, T.-J. Hsu, J.-R. Yu, FIUT: a new method for mining frequent itemsets, Information Sciences 179 (2009) 1724–1737.
[37] F.-H. Wang, On discovery of soft associations with "most" fuzzy quantifier for item promotion applications, Information Sciences 178 (2008) 1848–1876.
[38] C.-Y. Wang, S.-S. Tseng, T.-P. Hong, Flexible online association rule mining based on multidimensional pattern relations, Information Sciences 176 (2006) 1752–1780.
[39] F.-Y. Ye, J.-D. Wang, B.-L. Shao, New algorithm for mining frequent itemsets in sparse database, in: Proc. the Fourth International Conference on Machine Learning and Cybernetics, 2005, pp. 1554–1558.
[40] J.X. Yu, Z. Chong, H. Lu, Z. Zhang, A. Zhou, A false negative approach to mining frequent itemsets from high speed transactional data streams, Information Sciences 176 (14) (2006) 1986–2015.
[41] J.X. Yu, Z. Chong, H. Lu, A. Zhou, False positive or false negative: mining frequent itemsets from high speed transactional data streams, in: Proc. VLDB, 2004, pp. 204–215.
[42] M.J. Zaki, C.-J. Hsiao, Efficient algorithms for mining closed itemsets and their lattice structure, IEEE Transactions on Knowledge and Data Engineering 17 (4) (2005) 462–478.
[43] S. Zhang, J. Zhang, C. Zhang, EDUA: an efficient algorithm for dynamic database mining, Information Sciences 177 (2007) 2756–2767.
[44] X. Zhi-Jun, C. Hong, C. Li, An efficient algorithm for frequent itemset mining on data streams, in: Proc. ICDM, 2006, pp. 474–491.