Mining Regular Patterns in Data Streams

Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, and Byeong-Soo Jeong

Department of Computer Engineering, Kyung Hee University
1 Sochun-dong, Kihung-eup, Youngin-si, Kyonggi-do, Republic of Korea, 446-701
{tanbeer,farhan,jeong}@khu.ac.kr

Abstract. Discovering interesting patterns from high-speed data streams is a challenging problem in data mining. Recently, the support metric-based frequent pattern mining from data stream has achieved a great attention. However, the occurrence frequency of a pattern may not be an appropriate criterion for discovering meaningful patterns. Temporal regularity in occurrence behavior can be a key criterion for assessing the importance of patterns in several online applications such as market basket analysis, gene data analysis, network monitoring, and stock market. A pattern can be said *regular* if its occurrence behavior satisfies a user-given interval in the data steam. Mining *regular* patterns from static databases has recently been addressed. However, even though mining *regular* patterns from stream data is extremely required in online applications, no such algorithm has been proposed yet. Therefore, in this paper we develop a novel tree structure called Regular Pattern Stream tree (RPS-tree), and an efficient mining technique for discovering *regular* patterns over data stream. Using a sliding window method the RPS-tree captures the stream content, and with an efficient tree updating mechanism it constantly processes exact stream data when the stream flows. Extensive experimental analyses show that our RPS-tree is highly efficient in discovering *regular* patterns from a high-speed data stream.

Keywords: Data mining, data stream, pattern mining, regular pattern, sliding window.

1 Introduction

A data stream is a continuous, unbounded, and timely ordered sequence of data elements generated at a rapid rate. Unlike traditional static databases, 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. Even though mining user-interest based patterns from data stream has become a challenging issue, interests in online stream mining for discovering such patterns dramatically increased [1], [2], [10], [11], [12].

Mining frequent patterns [3], [6], from transactional databases has been actively and widely studied in stream data mining [2], [10], [11], [12] for over a decade. The rationale behind mining frequent patterns is that only patterns occurring at a high frequency in a database are of interest to users. Therefore, a pattern is called frequent if its occurrence frequency (i.e., support) in the database exceeds the user-given

H. Kitagawa et al. (Eds.): DASFAA 2010, Part I, LNCS 5981, pp. 399–413, 2010.
© Springer-Verlag Berlin Heidelberg 2010

support threshold. However, the occurrence frequency may not always represent the significance of a pattern. The other important criterion for identifying the interestingness of a pattern might be the shape of occurrence i.e., whether the pattern occurs periodically, irregularly, or mostly in a specific time interval.

The significance of patterns with temporal regularity can be revealed in a wide range of applications where users might be interested on the occurrence behavior (*regularity*) of patterns rather than just the occurring frequency. For example, in a retail chain data, some products may be sold more regularly than other products. Thus, even though both of the products are sold frequently over the entire selling history or for a specific time period (e.g., for a year), the products still need to be managed independently. That is, it is necessary to identify a set of items that are sold together at a regular interval for a specified time period. Also, to improve web site design, a site administrator may be interested in regularly visited web page sequences rather than web pages that are heavily hit only for a specific period. As for genetic data analysis, the set of all genes that co-occur at a fixed interval in DNA sequence may carry more significant information to scientists. Again, in stock market the set of stocks indices that rise at a regular interval might be of special interest to stock brokers and traders. The pattern regularity can also be a useful metric among other applications such as network monitoring, telecommunications or the sensor network.

Traditional frequent pattern mining techniques fail to uncover such *regular* patterns because they focus only on the high frequency patterns. Recently, Tanbeer et al. [4] studied the pattern appearance behaviour in static transactional databases. With the help of a *regularity* measure determined by the maximum interval at which a pattern occurs in a database, the study introduced a tree structure called RP-tree to discover *regular* patterns satisfying a user-given *regularity* threshold. The RP-tree requires two database scans and contains the information for only *regular* items in the database. However, with the recent development of technology several online applications require to handle a bulk amount of data in the form of data stream. For example, retail chains record millions of transactions, telecommunications companies connect thousands of calls, and popular web sites log millions of hits at a regular basis. It is, therefore, obvious that, because of the two database scans and the prior knowledge about the *regularity* threshold requirements, the RP-tree is inefficient in discovering *regular* patterns in the above data stream scenarios. Hence, to find *regular* patterns efficiently from data streams we require efficient algorithm that can capture the stream content with one scan and can competently mine the resultant patterns.

Motivated from the above demand, we address a new problem of mining *regular* patterns in data streams. We propose a novel single-pass tree structure, called the RPS-tree (Regular Pattern Stream tree), to capture the stream contents in a compact manner. Using an efficient pattern growth-based mining technique the RPS-tree can mine set of the *regular* patterns in stream data for a user-given *regularity* threshold.

To efficiently handle (or mine) continuously-generated data streams, sliding windows [10], [11], [12] are commonly used because of its flexibility to monitor the stream data at runtime. As new transactions arrive, the oldest transactions in the sliding window expire. Because of the efficient stream handling mechanism, we will exploit the sliding window in our approach.

Main idea of our RPS-tree is to develop a simple, but yet powerful, tree structure that captures the stream content for the current window in full with a single scan in a

canonical item order. Such construction feature enables its easy maintenance without any information loss during the slide of window. To the best of our knowledge, RPS-tree is the first effort to mine *regular* patterns from data streams. The experimental analyses on both real and synthetic data show that mining *regular* patterns from data streams with our RPS-tree is more efficient than that with the RP-tree.

The rest of the paper is organized as follows. Section 2 summarizes the existing algorithms related to our work. The detail discussion on RP-tree is also presented here. Section 3 introduces the problem of *regular* pattern mining in data stream. The structure and mining of our proposed RPS-tree are given in Section 4. We report our experimental results in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Many algorithms have been proposed for mining frequent patterns [3], [8] from static database, since its introduction by Agrawal et al. [6]. Han et al. [3] proposed the frequent pattern tree (FP-tree) and the FP-growth algorithm to mine frequent patterns with a pattern growth approach using only two database scans. Even though FP-growth algorithm has been highly efficient, it is not suitable for mining stream data because of its two database scans requirement.

A large number of techniques have been developed recently to mine frequent patterns from data stream [2], [10], [11], [12]. Algorithms in [10] and [11] use the sliding window concept to capture stream content with the help of a tree-based data structure. To facilitate the efficient mining and tree updating, the DSTree in [11] and the CPS-tree in [10] are constructed for the full window content. Using the FP-growth [3] algorithm both approaches discover the exact set of recent frequent patterns from the data stream with single scan. However, none of the support metric-based frequent pattern mining models is appropriate for discovering the special occurrence (i.e., *periodic* or *cyclic* or *regular*) characteristics of patterns from data stream.

Mining *periodic* patterns [1], [7], *cyclic* patterns [7], [9] and *regular* patterns [4] in static databases have been well-addressed over the last decade. *Periodic* pattern mining problem in time-series data focuses on the cyclic behavior of patterns either in the whole [7] (*full periodic patterns mining*) or at some point [1] (*partial periodic patterns mining*) of time-series. Such pattern mining has also been studied as a wing of sequential pattern mining [5], [9] in recent years. In [9], the authors extended the basic form of sequential patterns to cyclically repeated patterns. A progressive time list-based verification method to mine *periodic* patterns from a sequence of event sets was proposed in [5]. Ozden et al. [7] proposed a method to discover the association rules [6] occurring cyclically in a transactional database. Although mining *periodic* and *cyclic* patterns are closely related to our work, these algorithms cannot be directly applied for finding *regular* patterns from a data stream because they consider time-series or sequential data where the database is static.

Recently, Tanbeer et al. [4] proposed the Regular Pattern tree (RP-tree in short) to exactly mine the *regular* patterns from static transactional databases. The study defines a new *regularity* measure for a pattern determined by the maximum interval at which the same pattern occurs in a database.

Table 1. A transactional data stream (*DS*)

Id	Transaction	Id	Transaction	Id	Transaction
1	*a, c, e, f*	4	*c, d, e*	7	*a, c, d, e*
2	*b, c, f*	5	*a, b, c, e*	8	*c, d, e, f*
3	*b, c, f*	6	*c, d, e*	9	*a, c*

Construction of an RP-tree requires two database scans: one is for collecting the *regularity* of all distinct items and the other is for building the tree only for the *regular* items in each transaction. To keep track of the occurrence information, RP-tree explicitly maintains the transaction-*id*s (*tid*) of all transactions in the tree structure. It stores the *tid* of a transaction only at the last node of the transaction. The other nodes do not need to carry any occurrence information or support count (as does in FP-tree).

By applying an FP-growth-based [3] efficient pattern growth mining technique and exploiting the *tid*-information kept in the tree structure, RP-tree generates the complete set of *regular* patterns for the user-given *regularity* threshold. While mining *regular* patterns from an RP-tree, the transaction occurrence information maintained in it is used to calculate the *regularity* of each generated pattern.

However, as mentioned before, even though RP-tree efficiently finds *regular* patterns from static transactional databases, it is not suitable for mining *regular* patterns from data streams because of its *regularity* threshold-based tree structure, and two database scans requirement.

3 Problem Definition

Let $L = \{i_1, i_2, \ldots, i_n\}$ be a set of literals, called items that have been used as a unit information of an application domain. A set $X = \{i_j, \ldots, i_k\} \subseteq L$, where $j \leq k$ and $j, k \in [1, n]$, is called a *pattern* (or an *itemset*). A transaction $t = (tid, Y)$ is a tuple where *tid* represents a transaction-*id* (or time of transaction occurrence) and Y is a pattern. If $X \subseteq Y$, it is said that t contains X or X occurs in t. Let $size(t)$ be the size of t, i.e., the number of items in Y.

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 i-th arrived transaction. A window W can be referred to as a set of all transactions between the i-th and j-th (where $j > i$) arrival of transactions and the size of W is $|W| = j - i$, i.e., the number of transactions between the i-th and j-th arrival of transactions. Let each slide of window introduce and expire $slide\_size$, $1 \leq slide\_size \geq |W|$, transactions into and from the current window.

If X occurs in t_j $j \in [1, |W|]$, such transaction-*id* is denoted as t_j^X, $j \in [1, |W|]$. Therefore, $T_W^X = \{t_j^X, ..., t_k^X\}$, $j, k \in [1, |W|]$ and $j \leq k$ is the set of all transaction-*id*s where X occurs in the current window W.

Definition 1 (a period of *X* in *W*). Let t_{j+1}^X and t_j^X $j \in [1, (|W| - 1)]$, be two consecutive transaction-*id*s in T_W^X. The number of transactions (or the time difference) between t_{j+1}^X and t_j^X is defined as a period of X, say p^X (i.e., $p^X = t_{j+1}^X - t_j^X$, $j \in [1, (|W| - 1)]$). For

the simplicity of period computation, a '*null*' transaction with no item is considered at the beginning of W, i.e., $t_f = 0$ (*null*), where t_f represents the *tid* of the first transaction to be considered. Similarly, t_l, the *tid* of the last transaction to be considered, is the *tid* of the $|W|$-th transaction in the window, i.e., $t_l = t_{|W|}$. For instance, in the stream data in Table 1, consider the window is composed of eight transactions (i.e., *tid* = 1 to *tid* = 8 make the first window, say W_1). Then the set of transactions in W_1 where pattern $\{b,c\}$ appears is $T_{W_1}^{\{b,c\}} = \{2, 3, 5\}$. Therefore, the periods for $\{b,c\}$ are 2 (= 2 - t_f), 1 (= 3 - 2), 2 (= 5 - 3), and 3 (= t_l - 5), where $t_f = 0$ and $t_l = 8$.

The occurrence periods, defined as above, present the exact information about the appearance behavior of a pattern. A pattern will not be *regular* if, at any stage in W, it appears after sufficiently large period. The largest occurrence period of a pattern, therefore, can provide the upper limit of its periodic occurrence characteristic. Hence, the measure of the characteristic of a pattern of being *regular* in a W (i.e., the *regularity* of that pattern in W) can be defined as follows.

Definition 2 (*regularity* of pattern X in W). Let for a T_W^X, P_W^X be the set of all periods of X i.e., $P^X = \{p_1^X, ..., p_s^X\}$, where s is the total number of periods of X in W. Then, the *regularity* of X in W can be denoted as $reg_W(X) = Max(p_1^X, ..., p_s^X)$. For example, in the *DS* of Table 1 $reg_{W_1}(b,c) = 3$, since $P_{W_1}^{\{b,c\}} = Max(2, 1, 2, 3) = 3$.

Therefore, a pattern is called a *regular* pattern in W if its *regularity* in W is no more than a user-given maximum *regularity* threshold called *max_reg* λ, with $1 \leq \lambda \leq |W|$. The *regularity* threshold is given as the percentage of window size.

The *regular* patterns in W, therefore, satisfy the downward closure property [6], i.e., if a pattern is found to be *regular*, then all of its non-empty subsets will be *regular*. Thus, if a pattern is not *regular*, then none of its supersets can be *regular*. Given *DS*, $|W|$, and a *max_reg*, finding the complete set of *regular* patterns in W, R_w that have *regularity* of no greater than the *max_reg* value is the problem of mining *regular* patterns in data stream.

4 RPS-Tree: Design, Construction, and Mining

In this section, we first introduce our RPS-tree for data stream and describe efficient tree update mechanism for RPS-tree. We also discuss the mining of an RPS-tree here.

4.1 Design of an RPS-Tree

The structure of an RPS-tree consists of one *root* node referred to as the "*null*", a set of item-prefix sub-trees (children of the *root*), and an item header table called *regular pattern stream* table (RPS-table in short). Similar to an FP-tree [3] and an RP-tree [4], each node in an RPS-tree represents an itemset in the path from the *root* up to that node.

The RPS-tree maintains the occurrence information of all transactions (in the current window) in the tree structure. To explicitly track such information, it keeps a list

of transaction-*id* information only at the last item-node (say, *tail-item*) for a transaction. Such list is called a *tid-list*. Hence, an RPS-tree maintains two types of nodes; say *ordinary node*s and *tail-node*s. The former are types of nodes that do not maintain the *tid-list*, whereas the following definition describes the latter type.

Definition 3 (*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 an RPS-tree in this order, then the node of the tree that represents item i_n is defined as the *tail-node* for t. For example, if the first transaction (i.e., *tid* = 1) in the *DS* of Table 1 is inserted into an RPS-tree in lexicographical order, then the node that represents item '*f*' (i.e., the *tail-item* of the transaction) is the *tail-node* in the tree for that transaction.

Nodes of both types explicitly maintain parent, children, and node traversal pointers. In addition, each *tail-node* maintains a *tid-list* and a *tail-node* pointer. The *tail-node* pointer points to either the next *tail-node* in the tree if any, or '*null*'. Irrespective of the node type, no node in the RPS-tree maintains a support count value as does in an FP-tree [3].

The RPS-table consists of each distinct item in the current window with relative *regularity* and a pointer pointing to the first node in the RPS-tree that carries the item. Specifically, the RPS-table of an RPS-tree consists of three fields in sequence (i, r, p); item name (i), *regularity* of i (r), and a pointer to the RPS-tree for i (p). The item name is just a symbol to identify each item. The *regularity* is calculated by traversing the RPS-tree after the construction, which is explained in the next subsection. The item pointer facilitates the fast traversal to the whole tree in the mining phase. In addition, an RPS-tree maintains a *tail-node* pointer (say, tn_p) to point to the first *tail-node* in the tree. These pointers will facilitate fast tree traversal during the *regularity* calculation and tree update operation.

4.2 Construction of an RPS-Tree

The construction of the RPS-tree is featured in such a way that it takes only one scan over the high-speed data stream to capture the full content of the current window. In the RPS-tree, items are arranged according to any canonical order, which can be determined by the user prior to the tree construction. Once the item order is determined (say, for the initial window), items will follow this order in our RPS-tree for subsequent windows.

We use an example to illustrate the step-by-step construction process of an RPS-tree for the *DS* in Table 1. Let us assume that the RPS-tree is constructed in lexicographic order and each window is composed of eight transactions (i.e., the initial window, W_1 contains *tid*s from 1 to 8) as shown in Fig. 1(a).

The construction of an RPS-tree is similar to that of an FP-tree [3]. Initially, the RPS-tree is empty (i.e., starts with a '*null*' *root* node). To simplify the figures, we do not show the node traversal pointers in the trees, although they are maintained as in an FP-tree.

The first transaction to be inserted is {*a, c, e, f*} (i.e., *tid* = 1). As shown in Fig. 1(b), the transaction is inserted in the lexicographic order. Notice that '*f*' is the *tail-item* of the transaction and the *tail-node* "*f*:1" explicitly maintains the *tid* information in its *tid-list*. Also, tn_p points to the first *tail-node* "*f*:1" in the tree (as shown by dotted arrow in the figure). Fig. 1(c) shows the status of the RPS-tree after inserting

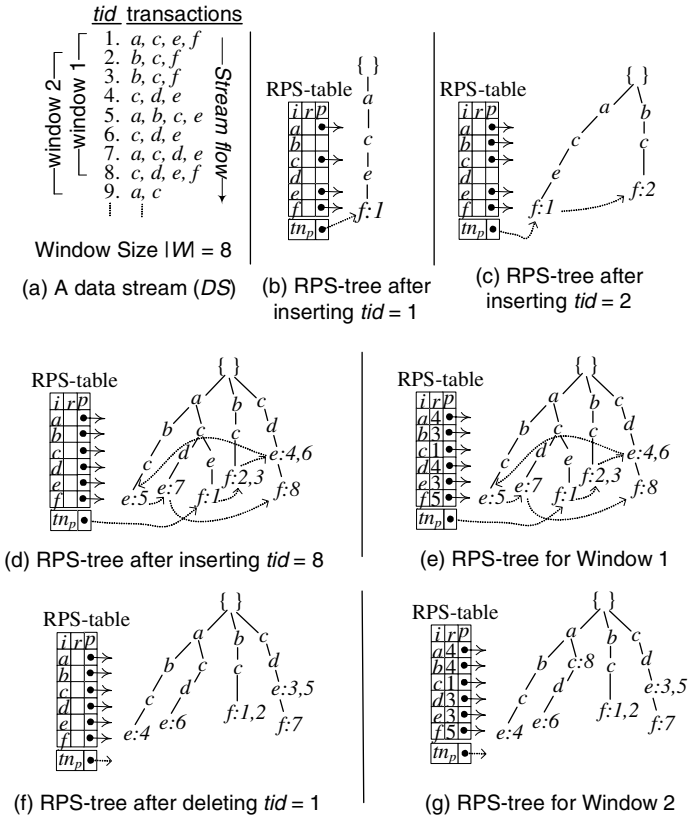


Fig. 1. Construction and update of an RPS-tree for the *DS* in Table 1

the second transaction (i.e., *tid* = 2) in similar fashion. The *tail-node* pointer is updated to point to the next *tail-node* "*f*:2" as shown in the figure. The RPS-tree after capturing all transactions of W_1 is presented in Fig. 1(d). Notice that the RPS-tree in Fig. 1(d) captures the complete information of W_1 in a compact fashion. However, the *regularity* of items in the RPS-table has not been computed yet.

To assist the *regularity* calculation, each item in the RPS-table is assigned a temporary array. Then, starting from the tn_p, and following *tail-node* pointers we visit each *tail-node* and accumulate the *tid*(s) available in its *tid-list* in the respective temporary arrays for every item from that *tail-node* up to the *root*. For example, after visiting the first two *tail-node*s of "*f*:1" and "*f*:2,3" in the RPS-tree of Fig. 1(d), the contents of the temporary arrays for items '*a*', '*b*', '*c*', '*e*', and '*f*' (i.e., items from *tail-node*s up to the *root*) are $T^a = \{1\}$, $T^b = \{2, 3\}$, $T^c = \{1, 2, 3\}$, $T^e = \{1\}$, and $T^f = \{1, 2, 3\}$.

Therefore, after finishing the traversal for all *tail-node*s, we obtain the complete list of *tid*s for each item in its respective temporary array. Thus, for instance, the set of transactions for item '*a*' we obtain, $T^a = \{1, 5, 7\}$. Then, it is rather simple calculation to find the P^a from T^a, which gives $reg_{W_1}(a) = 4$. The process of accumulating the

*tid*s and calculating the *regularity* of items in the RPS-table is termed as refreshing the RPS-table. Finally, Fig. 1(e) shows the final status of the RPS-tree and the RPS-table with the *regularity* of each item after the RPS-table refreshing operation. The RPS-tree in Fig. 1(e) is ready for mining the set of *regular* patterns from it upon request.

4.3 Updating the RPS-Tree

The simple construction feature of the RPS-tree enables it to delete the oldest and insert new transactions in an efficient manner. Because our RPS-tree keeps the *tail-node* pointers, one can easily locate the transaction(s) to be removed. To illustrate the RPS-tree updating mechanism, we use our running example of RPS-tree construction.

Suppose the window slides transaction-by-transaction (*slide_size* = 1) i.e., each slide of window expires the oldest and inserts one new transactions. Therefore, in this example, *tid* = 1 expires and a new transaction *tid* = 9 appears with the sliding of window.

To reflect the deletion of the oldest transaction we avoid the costly tree traversal operation. Rather following the *tail-node* pointers we visit only the *tail-node*s in the RPS-tree and adjust only the *tid-list*s of each *tail-node* in the tree for deleted transaction(s). We delete the *tid*s in the *tid-list* of each *tail-node* if their values are less than or equal to the *slide_size*; otherwise, we decrement them by *slide_size*. In process, we delete a *tail-node* and its path towards the *root* if its *tid-list* becomes empty. For example, we delete the *tail-node* "*f*:1" and its parent node "*e*", since after adjusting the *tid*s, the *tid-list* of "*f*:1" becomes empty. However, we avoid deleting nodes (toward the *root*) at the parent of "*e*", since it (the parent) has a child other than "*e*". Such operation ensures the deletion of only the expired transactions from the tree. The RPS-tree after deleting the oldest transaction (i.e., *tid* = 1) from the RPS-tree of W_1 and adjusting the *tid-list*s in all *tail-node*s is shown in Fig. 1(f). For the simplicity of figures we avoid showing the *tail-node* pointers in the figures. However, they are maintained as explained above.

Notice that the RPS-tree in Fig. 1(f) is ready to capture the new incoming transaction(s) in the sliding window. New transactions can be easily added to the RPS-tree by using the same technique as illustrated in Figs. 1(b) – (d). Usually, the *regularity* of patterns may change with the sliding of window (i.e., with the deletion and insertion of old and new transactions). For example, with $\lambda = 3$, and $|W| = 8$ for the *DS* in Fig. 1(a) the *regular* patterns $\{b\}$, and $\{b,c\}$ in W_1 become *irregular* (i.e., a pattern whose *regularity* is greater than *max_reg*) in W_2. Again, the *irregular* patterns $\{d\}$ and $\{c,d,e\}$ in W_1 become *regular* in W_2. Therefore, to reflect the correct *regularity* of each item in the current window, we perform the RPS-table refreshing operation at each window. Fig. 1(g) shows the status of the RPS-tree in W_2 after inserting new transaction and refreshing the RPS-table. Similar to the RPS-tree in Fig. 1(e), the complete set of *regular* patterns for the current window then can be mined from the RPS-tree in Fig. 1(g).

Based on the RPS-tree construction technique discussed above, we have the following property and lemma on the completeness of an RPS-tree. Let for each transaction t in a window W, *item*(t) be the set of all items in t and is called the full item projection of t.

Property 1: An RPS-tree contains *item*(*t*) for each transaction in a window only once.

Lemma 1: Given a stream database *DS* and a sliding window *W*, *item*(*t*) of all transactions in *W* can be derived from the RPS-tree for the *W*.

Proof: Based on the RPS-tree construction and updating mechanism and Property 1, *item*(*t*) of each transaction *t* is mapped to only one path in the RPS-tree and any path from the *root* up to a *tail-node* maintains the complete projection for exactly *n* transactions (where *n* is the total number of entries in the *tid-list* of the *tail-node*). ■

One may assume that the structure of an RPS-tree may not be memory efficient, since it explicitly maintains *tid*s of each transaction in the tree structure. But we argue that the RPS-tree achieves the memory efficiency by keeping such transaction information only at the *tail-node*s and avoiding the support count field at each node in the tree. Moreover, keeping the *tid* information in tree structure has also been found in literature for efficiently mining frequent patterns [2], [8]. To a certain extent, some of those studies additionally maintain support count and/or the *tid* information [2], [8] in each tree node. Furthermore, with modern technology, main memory space is no longer a big concern. Hence, we made the same realistic assumption as in many studies [11] that we have enough main memory space (in the sense that the trees can fit into the memory).

Since each transaction *t* in *W* contributes at best one path of *size*(*t*) to an RPS-tree, the total size contribution of all transactions in *W* can be at best $\sum_{t \in |W|} | size(t) |$. However, because there are usually many common prefix patterns among the transactions, the size of an RPS-tree is normally much smaller than $\sum_{t \in |W|} | size(t) |$.

It may be assumed that RPS-table refreshing mechanism of RPS-tree may require higher computation cost compared to scanning the stream data twice as in RP-tree. But, we argue that the cost of refreshing the RPS-table by traversing the paths from the *tail-node*s up to the *root* of the RPS-tree is much less than that by scanning the database a second time, since reading transactions from the memory-resident tree is much faster than scanning them from the database. Also note that, while accumulating the *tid*s from a *tail-node* during refreshing the RPS-table, we process as many transactions at a time as the size of its *tid-list*. This multiple transactions processing technique further reduces the RPS-table refreshing cost compared to obtaining the *regularity* of items through a second scan of the stream data. In the next subsection, we discuss the *regular* pattern mining process from the RPS-tree constructed for the current window.

4.4 Mining the RPS-Tree

Similar to the FP-growth [3] mining approach, we recursively mine the RPS-tree of decreasing size to generate *regular* patterns by creating conditional pattern-bases (*PB*) and corresponding conditional trees (*CT*) without additional database scan. Before discussing the mining process we explore the following important property and lemma of an RPS-tree.

Property 2: Each *tail-node* in an RPS-tree maintains the occurrence information of all nodes in the path (from that *tail-node* up to the *root*) in the transactions of its *tid-list*.

Lemma 2: Let $Z = \{a_1, a_2, ..., a_n\}$ be a path in an RPS-tree where node a_n, being the *tail-node*, carries the *tid-list* of the path. If the *tid-list* is carried to node a_{n-1}, then node a_{n-1} maintains the occurrence information of path $Z' = \{a_1, a_2, ..., a_{n-1}\}$ for the same set of transactions in the *tid-list* without any loss.

Proof: Based on Property 2, the *tid-list* in node a_n explicitly maintains the occurrence information of Z' for the same set of transactions. Therefore, the same *tid-list* at node a_{n-1} exactly maintains the same information for Z' without any loss. ■

Using the features revealed by the above property and lemma and based on the downward closure property [6], we proceed to mining the RPS-tree for only *regular* items starting from the bottom up to the top in the RPS-table. If an item *i* in the RPS-table is an *irregular* item, we ignore mining for it. However, following the node traversal pointers we only visit each node N_i for *i* in the RPS-tree and carry (i.e, copy) N_i's *tid-list* to its parent N^p. Therefore, the parent node N^p is temporarily converted to a *tail-node* if it was an *ordinary node*; otherwise (i.e., if N^p is a *tail*-node), the *tid-list* is added with its previous *tid-list*. At the same time, from N_i we delete the *tid-list* it borrowed as a parent node from its children (if any). This process of carrying the *tid-list* of a (temporary) *tail-node* to its parent node is termed as *carry-tid* and the set of *tid*(s) carried to the parent is called as *carried-tid*.

We use our running example to illustrate the mining on an RPS-tree. Consider mining the RPS-tree of Fig. 1(e) for $\lambda = 3$. Since '*f*', the bottommost item in the RPS-table, is not *regular* (i.e., $reg_{W_1}(f) > 3$), we only perform the *carry-tid* operation for each of its nodes in the RPS-tree. Fig. 2(a) shows the status of the RPS-tree after the *carry-tid* operation for '*f*'. The *tid*s shown in dark box in the figure are *carried-tid*s.

Mining for each *regular* item *i* in the RPS-table, on the other hand, is performed by constructing the conditional pattern-base PB_i for *i* by projecting only the prefix subpaths of N_i in the RPS-tree with the *tid-list* of N_i. During this projection, we only include *regular* items. Determination of whether an item is *regular* can be easily done by a simple look-up (an $O(1)$ operation) at the RPS-table. There is no worry about possible omission or doubly counting of items. While visiting each N_i, we perform the *carry-tid* operation for the node as well.

To store the *regularity* of items with *i*, a small RPS-table, say RPS-table$_i$, is maintained for PB_i. While constructing PB_i, to compute the *regularity* of each item *j* in the RPS-table$_i$, based on Property 2 we map all N_i's *tid-list*s to all items in the respective path explicitly in temporary arrays (one for each item). Once the PB_i is constructed, the contents of the temporary array for *j* in the RPS-table$_i$ represent the T^{ij} (i.e., set of all *tid*s where items *i* and *j* occur together) in PB_i. Therefore, it is a rather simple calculation to compute $reg_W(j)$ from T^{ij} by generating P^{ij}. The conditional tree for *i* CT_i is, then, constructed from its PB_i by removing all *irregular* items and their respective nodes from the RPS-table$_i$ and PB_i, respectively. If the deleted node is a *tail-node*, based on Lemma 2 its *tid-list* is pushed-up to its parent node.

Let *j* be the bottommost item in RPS-table$_i$ of CT_i. Then the pattern $\{i,j\}$ is generated as a *regular* pattern with the *regularity* of *j* in the RPS-table$_i$. The same process of creating a conditional pattern-base and its corresponding conditional tree is repeated for further extensions of pattern $\{i,j\}$.

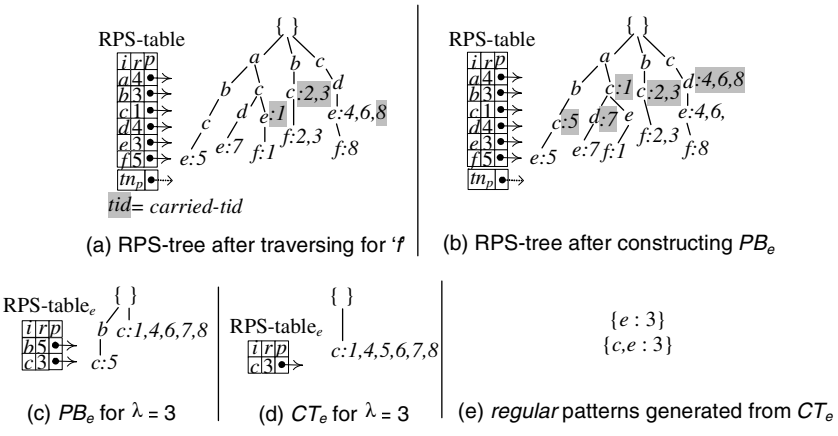


Fig. 2. Mining the RPS-tree of Fig. 1(e) for λ = 3

The next item in the RPS-table in Fig. 2(a) (i.e., '*e*') is a *regular* item (i.e., $reg_{W_1}(e) \leq 3$). Therefore, we construct the PB_e, and then CT_e. We also perform the *carry-tid* operation while constructing the PB_e. The structure of the RPS-tree after the *carry-tid* operation for '*e*' is illustrated in Fig. 2(b). Fig. 2(c) shows the structure of the PB_e. The CT_e is constructed by removing all *irregular* items and their respective nodes from the RPS-table$_e$ and PB_e. The CT_e in Fig. 2(d) is, therefore, constructed by deleting all entries for *irregular* item '*b*'. The set of all *regular* patterns mined from the CT_e is given in Fig. 2(e). The value after ':' indicates the *regularity* of individual pattern. The whole process is repeated until the top of the RPS-table (i.e., '*a*').

Notice that after each successful *carry-tid* operation any node in the RPS-tree retains its original status of either as an *ordinary node* or a *tail-node* (e.g., nodes "*e*" and "*e*:4,6" from Fig. 2(a) to Fig. 2(b)). Also, since we start mining from the bottommost item in the RPS-tree, there is no scope of missing any *tid-list* in the whole tree from carrying upward. It can be noticed that, when mining for all items in the RPS-table is completed, the *carry-tid* operations will accumulate a copy of all *tid*s at the *root* node. It is then rather a trivial task to remove them from the *root* to make the tree consistent to be updated for the next window content.

Therefore, from the above mining process we can say that for a given *max_reg* and *W* the R_W can be generated from an RPS-tree constructed on the window contents. In the next section, we evaluate the performance of our RPS-tree.

5 Experimental Analyses

In this section, we present the experimental results and related analysis on the comparison of proposed RPS-tree with its state-of-the-art counterparts. To the best of our knowledge, the RPS-tree is the first effort to address the problems of *regular* pattern mining in data stream. Therefore, we compare its performance with that of the RP-tree [4], the existing algorithm available for *regular* pattern mining. All programs are

Table 2. Dataset characteristics

Dataset	#Trans.(T)	#Items(I)	MaxTL(MTL)	AvgTL(ATL)
BMS-POS	515,597	1,657	164	6.53
Kosarak	990,002	41,270	2,498	8.10
T10I4D100K	100,000	870	29	10.10

written in Microsoft Visual C++ 6.0 and run with Windows XP on a 2.66 GHz machine with 1GB of main memory. The runtime specifies the total execution time, i.e., CPU and I/Os.

We use several real and synthetic datasets (as in Table 2) which are frequently used in frequent pattern mining experiments, since they maintain the characteristics of transactional data. The first two datasets were obtained from [14]. *BMS-POS* contains several years worth of point-of-sale data from a large electronics retailer. *Kosarak* is a dataset of click-stream data from a Hungarian on-line news portal. *T10I4D100K*, developed by [13], is a synthetic dataset. In all experiments, we consider *slide_size* = 1. In the first experiment, we study the compactness of our RPS-tree in stream data.

5.1 Memory Efficiency

We conducted experiments to verify the memory requirements for our RPS-tree on different datasets by varying the window size. Since RPS-tree is a *regularity* threshold independent tree structure, the *regularity* threshold values do not influence on its memory requirements. Therefore, in this experiment, the reported required memory represents the size of the underlying tree structure after capturing only the complete sliding window content. Because RP-tree is a *regularity* threshold-based tree structure, we do not compare its memory requirement with RPS-tree.

Table 3 reports RPS-tree's memory requirement (on average for all window for a fixed window size) in several datasets with the variation of window size at each case. In *BMS-POS*, for example, when the window size is 100K (i.e., $|W|^1$ = 100K), the required memory is on an average 13.81 MB in each window. Again, in the same dataset RPS-tree consumes on an average 33.51 MB memory when $|W|^4$ = 400K.

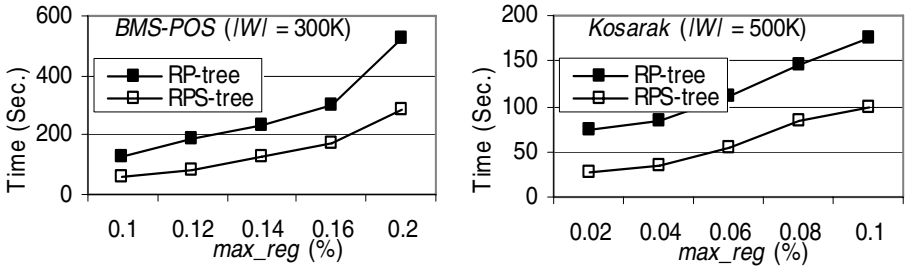
Hence, from the data in Table 3 it can be observed that when capturing the stream data of different characteristics, an RPS-tree is memory efficient for the available memory now-a-days. In the next experiment, we compare execution time between our RPS-tree and existing RP-tree.

Table 3. Memory requirement (MB) with window size variation in RPS-tree

Dataset with different window sizes	For window size				
	$\|W\|^1$	$\|W\|^2$	$\|W\|^3$	$\|W\|^4$	$\|W\|^5$
BMS-POS ($\|W\|^1$ = 100K, $\|W\|^2$ = 200K, $\|W\|^3$ = 300K, $\|W\|^4$ = 400K)	13.81	22.26	29.97	33.51	-
Kosarak ($\|W\|^1$ = 100K, $\|W\|^2$ = 300K, $\|W\|^3$ = 500K, $\|W\|^4$ = 700K, $\|W\|^5$ = 900K)	55.67	84.92	130.41	159.24	228.97
T10I4D100K ($\|W\|^1$ = 30K, $\|W\|^2$ = 50K, $\|W\|^3$ = 70K, $\|W\|^4$ = 90K)	3.51	5.09	6.96	8.93	-

5.2 Runtime Efficiency

To study the runtime performance experiments were conducted with a mining request at each window by varying the *max_reg* values for each dataset while the window size |*W*| was kept fixed at reasonably high values. The results of the experiment are shown in Fig. 3. The time shown on the *y*-axes are the total time for scanning the window content, tree construction, tree update and RPS-table refreshing time (only for RPS-tree), and mining. Notice that mining data stream with RP-tree requires scanning each window content twice, since it was originally proposed for static databases.



(a) On *BMS-POS* (|*W*| = 300K) (b) On *Kosarak* (|*W*| = 500K)

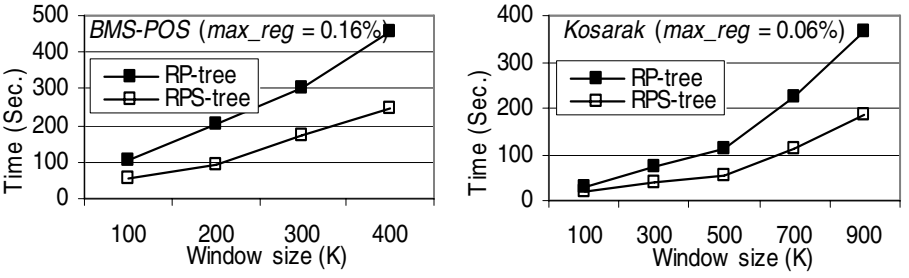
Fig. 3. Runtime comparison

As shown in Fig. 3, the higher the *max_reg* values, the longer the overall time required by both trees. The reason is that, the higher the *max_reg* value, the greater the number of *regular* patterns can be generated from the current window. However, the results clearly demonstrate that RPS-tree outperforms RP-tree in terms of overall runtime by multiple orders of magnitude for both high and low *max_reg* values. The key to this performance gain of RPS-tree is its efficient tree updating mechanism that only scans the new incoming transaction(s) once, while RP-tree requires scanning the whole window content twice. The gain of RPS-tree over the RP-tree becomes more prominent when the window size is larger. We also evaluated RPS-tree's performance on the variation of window size, as shown in the next experiment.

5.3 Window Size

Because RPS-tree captures the full window content, its performance may vary depending on the window size i.e., |*W*|. Hence, to determine the effect of changes in window size on the runtime of RPS-tree, we analyzed its performance by varying |*W*| over different datasets while keeping the *max_reg* value fixed. The graphs presented in Fig. 4 show the results on *BMS-POS* for *max_reg* = 0.16%, and *Kosarak* for *max_reg* = 0.06%. The *y*-axes in the graphs represent the average total time (including construction time, tree update time for the RPS-tree only, and mining time) required in all active windows.

Larger window sizes resulted in a longer total tree construction time for both trees. However; the overall runtime required by RPS-tree is small enough to handle larger



(a) On *BMS-POS* (*max_reg* = 0.16%) (b) On *Kosarak* (*max_reg* = 0.06%)

Fig. 4. RPS-tree's performance on size of *W*

windows in different datasets. For RP-tree, in contrast, a sharp increase in runtime according to an increase in window size was observed. As a result, the performance gaps between the two tree structures widen for larger windows. For example, in *Kosarak* for *max_reg* = 0.06% when |*W*| = 100K, RPS-tree's gain is not much prominent (Fig. 4(b)). However, for |*W*| = 900K RPS-tree achieves a significant improvement in overall runtime. Similar results we obtained in *BMS-POS* as well. Therefore, these results show that RPS-tree is better than RP-tree in handling larger windows and producing the exact set of *regular* patterns within a reasonable amount of time over data streams.

The above experiments demonstrate that RPS-tree outperforms the state-of-the-art algorithms in mining *regular* patterns from data streams of various characteristics (refer to Table 2). The easy and simple maintenance phase of the RPS-tree has been the key to its significant performance gain.

6 Discussions and Conclusions

In this paper, we define the *regularity* of a pattern by its maximum occurrence interval (in a window) calculated from its *tids* (Definition 2) obtained during mining. However, other parameters such as the arithmetic mean or variance of occurrence intervals can also be considered as *regularity* measures for finding interesting patterns from data streams. Since RPS-tree maintains the exact occurrence information for all transactions in the current window, and the mining phase provides the complete *tids* for each pattern, computing such parameters can also be simple similar to computing the maximum occurrence interval for a pattern.

In conclusions, we introduced a new concept of mining interesting patterns (called *regular* patterns) that occur with a temporal *regularity* in high-speed data streams. We proposed a novel tree structure, RPS-tree, to capture the stream content in memory-efficient manner and to enable *regular* pattern mining from it. To obtain the fast and interesting results RPS-tree can be updated efficiently for the current content of the stream. The experimental analysis reveals that RPS-tree is significantly faster than other algorithm that can be used in mining *regular* patterns from a data stream.

References

1. Han, J., Dong, G., Yin, Y.: Efficient Mining of Partial Periodic Patterns in Time Series Database. In: 15th ICDE, pp. 106–115 (1999)
2. Zhi-Jun, X., Hong, C., Li, C.: An Efficient Algorithm for Frequent Itemset Mining on Data Streams. In: ICDM, pp. 474–491 (2006)
3. Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns without Candidate Generation. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 1–12 (2000)
4. Tanbeer, S.K., Ahmed, C.F., Jeong, B.-S., Lee, Y.-K.: Mining Regular Patterns in Transactional Databases. IEICE Trans. on Inf. & Sys. E91-D(11), 2568–2577 (2008)
5. Huang, K.-Y., Chang, C.-H.: Mining Periodic Patterns in Sequence Data. In: Kambayashi, Y., Mohania, M., Wöß, W. (eds.) DaWaK 2004. LNCS, vol. 3181, pp. 401–410. Springer, Heidelberg (2004)
6. Agrawal, R., Srikant, R.: Fast algorithms for Mining Association Rules in Large Databases. In: VLDB, pp. 487–499 (1994)
7. Ozden, B., Ramaswamy, S., Silberschatz, A.: Cyclic Association Rules. In: 14th ICDE, pp. 412–421 (1998)
8. Zaki, M.J., Hsiao, C.-J.: Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure. IEEE Trans. Knowl. Data Eng. 17(4), 462–478 (2005)
9. Toroslu, I.H., Kantarcioglu, M.: Mining Cyclically Repeated Patterns. In: Kambayashi, Y., Winiwarter, W., Arikawa, M., et al. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 83–92. Springer, Heidelberg (2001)
10. Tanbeer, S.K., Ahmed, C.F., Jeong, B.-S., Lee, Y.-K.: Sliding Window-based Frequent Pattern Mining over Data Streams. Information Sciences 179, 3843–3865 (2009)
11. Leung, C.K.-S., Khan, Q.I.: DSTree: A Tree Structure for the Mining of Frequent Sets from Data Streams. In: ICDM, pp. 928–932 (2006)
12. Li, H.-F., Lee, S.-Y.: Mining Frequent Itemsets over Data Streams Using Efficient Window Sliding Techniques. Expert Systems with Applications 36, 1466–1477 (2009)
13. IBM, QUEST Data Mining Project, `http://www.almaden.ibm.com/cs/quest`
14. Frequent Itemset Mining Dataset Repository, `http://fimi.cs.helsinki.fi/data/`