# Discovering Periodic-Frequent Patterns in Transactional Databases

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

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

**Abstract.** Since mining frequent patterns from transactional databases involves an exponential mining space and generates a huge number of patterns, efficient discovery of user-interest-based frequent pattern set becomes the first priority for a mining algorithm. In many real-world scenarios it is often sufficient to mine a small interesting representative subset of frequent patterns. Temporal periodicity of pattern appearance can be regarded as an important criterion for measuring the interestingness of frequent patterns in several applications. A frequent pattern can be said periodic-frequent if it appears at a regular interval given by the user in the database. In this paper, we introduce a novel concept of mining periodic-frequent patterns from transactional databases. We use an efficient tree-based data structure, called Periodic-frequent pattern tree (PF-tree in short), that captures the database contents in a highly compact manner and enables a pattern growth mining technique to generate the complete set of periodic-frequent patterns in a database for user-given periodicity and support thresholds. The performance study shows that mining periodic-frequent patterns with PF-tree is time and memory efficient and highly scalable as well.

**Keywords:** Data mining, knowledge discovery, frequent pattern, interesting pattern, periodic-frequent pattern.

## 1 Introduction

Mining frequent patterns [1], [2], [4], [6] from transactional databases has been actively and widely studied in data mining and knowledge discovery techniques such as association rule, sequential pattern, classification, and clustering. Since the rationale behind mining the support metric-based frequent patterns is to find the set of patterns that appear frequently in a database, a huge number of patterns are normally generated and most of which might be found insignificant depending on application or user requirement. Moreover, the computation cost in finding such number of patterns may not be trivial. As a result, several techniques to mine constraint-based and/or user interest-based frequent patterns [9], [10], [5] have been proposed recently to reduce the desired result set by effectively

T. Theeramunkong et al. (Eds.): PAKDD 2009, LNAI 5476, pp. 242–253, 2009.

**Table 1.** A transactional database

| Id | Transaction | Id | Transaction | Id | Transaction | Id | Transaction | Id | Transaction |
|---|---|---|---|---|---|---|---|---|---|
| 1 | *a c d e* | 3 | *a c e* | 5 | *a c e f* | 7 | *b c d e* | 9 | *a b c d* |
| 2 | *a d e f* | 4 | *c d e* | 6 | *b f* | 8 | *b c d e* | 10 | *a b e f* |

and efficiently applying early pruning techniques. Uses of several interesting parameters such as closed [3], $K$-most [5], demand-driven [10], maximum length [9] are found useful in literature in discovering frequent patterns of special interest. The other important criterion for identifying the interestingness of frequent patterns might be the shape of occurrence, i.e., whether they occur periodically, irregularly, or mostly in specific time interval in the database.

In a retail market, among all frequently sold products, the user may be interested only on the regularly sold products compared to the rest. Besides, for improved web site design or web administration an administrator may be interested on the click sequences of heavily hit web pages. Also, in genetic data analysis the set of all genes that not only appear frequently but also co-occur at regular interval in DNA sequence may carry more significant information to scientists. As for stock market, the set of high stocks indices that rise periodically may be of special interest to companies and individuals. In the above examples, we observe that the occurrence periodicity plays an important role in discovering some interesting frequent patterns in a wide variety of application areas. We define such a frequent pattern that appears maintaining a similar period/interval in a database as a *periodic-frequent pattern*.

Let us consider the transactional database of Table 1 with ten transactions. The support of the patterns "*e*", "*ae*", "*cd*", "*ce*", "*b*", and "*de*" in the database are respectively 8, 5, 5, 6, 5, and 5. Even though these patterns may be frequent in the database, some of them may not be periodic-frequent because of non-similar occurrence periods. For example, "*b*" and "*ae*" appear more frequently at a certain part of the database (i.e., "*b*" at the end and "*ae*" at the beginning of database) than the rest part. In contrast, patterns "*e*", "*cd*", "*ce*", "*de*" appear at almost regular intervals. Therefore, the latter patterns can be more important frequent patterns in terms of the appearance intervals. On the other hand, although the respective appearance intervals of patterns "*ac*", "*cde*", "*f*" etc. are almost similar, they may not be the patterns of interest due to their relatively low frequency. The traditional frequent pattern mining techniques fail to discover such periodic-frequent patterns because they are only concerned about the occurrence frequency and disregard the pattern occurrence behavior.

Motivated by the above discussion and examples, in this paper, we address a new problem of discovering *periodic-frequent patterns* in a transactional database. We define a new *periodicity* measure for a pattern by the maximum interval at which the same pattern occurs in a database. Therefore, periodic-frequent patterns, defined such way, satisfy the downward closure property [1], i.e., if a frequent pattern is found periodic then all of its non-empty subsets will be periodic. In other words, if a frequent pattern is not periodic then none of its supersets can be periodic. In order to mine periodic-frequent patterns, we

capture the database contents in a highly compact tree structure, called a PF-tree (Periodic-frequent Pattern tree). To ensure that the tree structure is compact and informative, only periodic-frequent length-1 items will have nodes in the tree and to obtain higher prefix sharing, more frequently occurring items are located at the upper part of the tree. We also propose an efficient pattern growth-based mining approach to mine the complete set of periodic-frequent patterns from our PF-tree. The comprehensive performance study on both synthetic and real datasets demonstrates that discovering periodic-frequent patterns from the PF-tree is highly memory and time efficient.

The rest of the paper is organized as follows. In Section 2, we summarize the existing algorithms to mine interesting frequent patterns. Section 3 formally introduces the problem of periodic-frequent pattern mining. The structure and mining of PF-tree are given in Section 4. We report our experimental results in Section 5. Finally, Section 6 concludes the paper.

## 2 Related Work

Since its introduction by Agrawal et al. in 1993 [1], a large number of techniques [2], [4], [6] have been proposed in mining support constraint-based frequent patterns. Han et al. [2] proposed the frequent pattern tree (FP-tree) and the FP-growth algorithm to mine frequent patterns with a memory and time efficient manner. The performance gain achieved by the FP-growth is mainly based on the highly compact nature of the FP-tree, where it stores only the frequent items in a support-descending order. To reduce the size of resultant pattern set and to improve the mining efficiency closed [3] frequent pattern mining has been focused. However, none of the above frequent pattern mining techniques can successfully provide interesting frequent patterns, since their outputs are only based on the support threshold.

Mining interesting frequent patterns of different forms [9], [10], [5], [7] in transactional databases and time-series data has been well-addressed over the last decade. Minh et al. [5] proposed a top-$K$ frequent pattern mining technique that allows the user to control the number of patterns to be discovered without any support threshold. In [9] the authors put efforts to discover the maximum length frequent patterns, rather than finding the complete set of frequent patterns. They have shown the suitability of their method in several real world scenario where long patterns play significant role. Wang et al. [10] mined frequent patterns from relational database. Using the user's query, they find the frequently occurring pattern structures defined by attributes values of items. However, the above models still fail to discover the interesting periodic occurrence characteristics of frequent patterns.

Temporal relationships among pattern occurrences were studied in [7] which focused on discovering the frequently occurring substring patterns in a dimension of multivariate time-series data. Periodic pattern mining has also been studied as a wing of sequential pattern mining [8] in recent years. Although periodic pattern mining is closely related to our work, it cannot be directly applied for

finding the periodic-frequent patterns from a transactional database because of two primary reasons. First, it considers either time-series or sequential data; second, it does not consider the support threshold which is the only constraint to be satisfied by all frequent patterns. Our proposed periodic-frequent pattern mining technique, on the other hand, introduces a new interesting measure of periodicity and provides the set of patterns that satisfy both of the periodicity and support thresholds in a transactional database.

## 3 Problem Definition

In this section, we describe the conceptual framework of the periodic-frequent pattern mining and introduce the basic notations and definitions in this regard.

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of literals, called items that have ever been used as a unit information of an application domain. A set $X = \{i_j, ..., i_k\} \subseteq I$, 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 timestamp) and $Y$ is a pattern. A transactional database $TDB$ over $I$ is a set of transactions $T = \{t_1, ..., t_m\}$, $m = |TDB|$, where $|TDB|$ is the size of $TDB$ in total number of transactions. If $X \subseteq Y$, it is said that $t$ contains $X$ or $X$ occurs in $t$ and such transaction-id is denoted as $t_j^X$, $j \in [1, m]$. Therefore, $T^X = \{t_j^X, ..., t_k^X\}$, $j, k \in [1, m]$ and $j \leq k$ is the set of all transaction-ids where $X$ occurs in $TDB$.

**Definition 1.** (***a period of pattern*** $X$) *Let $t_{j+1}^X$ and $t_j^X$, $j \in [1, (m-1)]$ be two consecutive transaction-ids where $X$ appears. The number of transactions or the time difference between $t_{j+1}^X$ and $t_j^X$ can be defined as a period of $X$, say $p^X$ (i.e., $p^X = t_{j+1}^X - t_j^X$, $j \in [1, (m-1)]$). For the simplicity of period computation, the first and the last transactions (say, $t_f$ and $t_l$) in $TDB$ are respectively identified as "null" (i.e., $t_f = 0$) and $t_m$ (i.e., $t_l = t_m$). For instant, in the $TDB$ of Table 1 the set of transactions where pattern "de" appears is $T^{de} = \{1, 2, 4, 7, 8\}$. Therefore, the periods for this pattern are $1(= 1 - t_f)$, $1(= 2 - 1)$, $2(= 4 - 2)$, $3(= 7 - 4)$, $1(= 8 - 7)$, and $2(= t_l - 8)$, where $t_f = 0$ and $t_l = 10$.*

The occurrence intervals, defined as above, can give the exact information of appearance behavior of a pattern. 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 *periodic* in a $TDB$ (we call it as the periodicity of that pattern) can be defined as follows.

**Definition 2.** (***periodicity of pattern*** $X$) *Let for a $T^X$, $P^X$ be the set of all periods of $X$ i.e., $P^X = \{p_1^X, ..., p_r^X\}$, where $r$ is the total number of periods in $P^X$. Then, the periodicity of $X$ can be denoted as $Per(X) = Max(p_1^X, ..., p_r^X)$. For example, in the $TDB$ of Table 1, $Per(de) = 3$ i.e., $Max(1, 1, 2, 3, 1, 2)$.*

**Definition 3.** (***support of pattern*** $X$) *The number of transactions in a $TDB$ that contain $X$ is called the support of $X$ in $TDB$ and denoted as $Sup(X)$. Therefore, $Sup(X) = |T^X|$, where $|T^X|$ is the size of $T^X$. For example, the support of pattern "de" in the $TDB$ of Table 1 is $Sup(de) = 5$, since $|T^{de}| = 5$.*

A pattern is called a *periodic-frequent pattern* if it satisfies both of the following two criteria: ($i$) its periodicity is no greater than a user-given maximum periodicity threshold say $max\_per$, $\lambda$ and ($ii$) its support is no less than a user-given minimum support threshold, say $min\_sup$, $\alpha$, with $\lambda$, $\alpha$ in percentage of $|TDB|$. Therefore, the *Periodic-frequent pattern mining problem*, given $\lambda$, $\alpha$, and a $TDB$, is to discover the complete set of periodic-frequent patterns in $TDB$ having periodicity no more than $\lambda$ and support no less than $\alpha$. Let $PF_{TDB}$ refer to the set of all periodic-frequent patterns in a $TDB$ for given $\lambda$ and $\alpha$.

## 4 PF-Tree: Design, Construction and Mining

In this section, we describe the construction and mining of Periodic-Frequent Pattern tree (PF-tree). Since periodic-frequent patterns follow the downward closure property, periodic-frequent length-1 items will play an important role in mining periodic-frequent patterns. Therefore, it is necessary to perform one database scan to identify the set of length-1 periodic-frequent items. The objective of this scan is to collect the support count (i.e., frequency) and the periodicity of each item in the database. Consequently, for further processing we can ignore all items that do not satisfy the periodicity and support thresholds. Let $PF$ be the set of all items that are found periodic-frequent at this stage.

### 4.1 Structure of the PF-Tree

The structure of the PF-tree includes a prefix-tree and a periodic-frequent item list, called the PF-list, consisting of each distinct item with relative support, periodicity and a pointer pointing to the first node in the PF-tree carrying the item. To facilitate high degree of compactness, items in a PF-tree are arranged in support-descending item order. It has been proved in [6] that such tree can provide a highly compact tree structure (as FP-tree in [2] and CP-tree in [6]) and an efficient mining phase using FP-growth mining technique. Before discussing the tree construction process, we provide the PF-list construction technique and the node structures of a PF-tree.

**Construction of the PF-list.** Each entry in a PF-list consists of three fields - item name ($i$), total support ($f$), and the periodicity of $i$ ($p$). The $tid$s of the last occurring transactions of all items in the PF-list are explicitly recorded for each item in a temporary array, called $id_l$. Let $t_{cur}$ and $p_{cur}$ respectively denote the $tid$ of current transaction and the most recent period for an item $i$. The PF-list is, therefore, maintained according to the process given in Fig. 1.

In Fig. 2 we show how the PF-list is populated for the $TDB$ of Table 1. With the scan of the first transaction $\{a\ c\ d\ e\}$ (i.e., $t_{cur} = 1$), the items '$a$', '$c$', '$d$', and '$e$' in the list are initialized as shown in Fig. 2(a) (lines 1 and 2 in Fig. 1). The next transaction $\{a\ d\ e\ f\}$ with $t_{cur} = 2$ initializes PF-list entries for item '$f$' and updates values $\{f; p\}$ and $id_l$ (lines 3 - 6 in Fig. 1) respectively to $\{2; 1\}$ and $\{2\}$ for items '$a$', '$d$', and '$e$' (Fig. 2(b)). As shown in Fig. 2(c), the periodicity ($p$)

1. **If** $t_{cur}$ is $i$'s first occurrence
2. $f = 1$, $id_l = t_{cur}$, $p = t_{cur}$;
3. **Else** $f = f + 1$;
4. $p_{cur} = t_{cur} - id_l$, $id_l = t_{cur}$;
5. **If** ($p_{cur} > p$)
6. $p = p_{cur}$;
7. At the end of *TDB*, calculate $p_{cur}$ for each item by considering $t_{cur}$ = the *tid* of the last transaction in *TDB*, and update the respective $p$ value according to step 5 and 6;

**Fig. 1.** PF-list maintenance algorithm

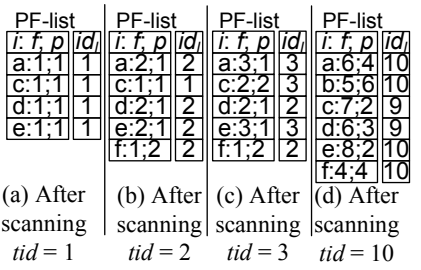



**Fig. 2.** PF-list population after the first scan of the *TDB* in Table 1

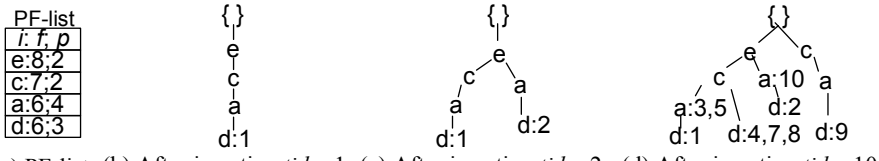
of '*c*' changes from 1 to 2, since after scanning $tid = 3$ the value of $p_{cur}$ for it is found greater than its previous periodicity (lines 5 and 6 in Fig. 1). The PF-list after scanning all ten transactions is given in Fig. 2(d). To reflect the correct periodicity for each item in the list, the whole PF-list is refreshed as mentioned in line 7 of Fig. 1 which results the final PF-list of Fig. 2(d). Therefore, once the PF-list is built, we generate the $PF$ by removing items that do not satisfy the user-given periodicity and support thresholds from it.

**PF-tree Node Structures.** An important feature of a PF-tree is that, it explicitly maintains the occurrence information for each transaction in the tree structure by keeping an occurrence transaction-id list, called *tid*-list, only at the last node of every transaction. Hence, there are two types of nodes maintained in a PF-tree; ordinary node and *tail*-node. The former is the type of nodes similar to that used in FP-tree, whereas the latter is the node that represents the last item of any sorted transaction. Therefore, the structure of a *tail*-node is $N[t_1, t_2, ..., t_n]$, where $N$ is the node's item name and $t_i$, $i \in [1, n]$, ($n$ be the total number of transactions from the *root* up to the node) is a transaction-id where item $N$ is the last item. Like the FP-tree [2], each node in a PF-tree maintains parent, children, and node traversal pointers. However, irrespective of the node type, no node in a PF-tree maintains support count value in it.

### 4.2 Construction of the PF-Tree

With the second database scan, the PF-tree is constructed in such a way that, it only contains nodes for items in $PF$. We use an example to illustrate the construction of a PF-tree.

Consider the transactional database of Table 1. In Fig. 3, we show the PF-tree construction steps for $\lambda = 4$ and $\alpha = 5$. At first, the support-descending PF-list (Fig. 3(a)) for all periodic-frequent items is constructed from the PF-list of Fig. 2(d). Next, using the FP-tree [2] construction technique, only the items in this list take part in PF-tree construction. For the simplicity of figures, we do not show the node traversal pointers in trees, however, they are maintained in a fashion like FP-tree does. The tree construction starts with inserting the




(a) PF-list (b) After inserting *tid* = 1 (c) After inserting *tid* = 2 (d) After inserting *tid* = 10

**Fig. 3.** Construction of a PF-tree for the *TDB* in Table 1 with $\alpha = 5$ and $\lambda = 4$

first transaction $\{a\ c\ d\ e\}$ (i.e., $tid = 1$) according to PF-list order, as shown in Fig. 3(b), since all the items in the transactions are periodic-frequent. The *tail*-node "$d : 1$" carries the *tid* of the transaction. After removing the non-periodic-frequent item '$f$', the second transaction is inserted into the tree in the form and an order of $\{e\ a\ d\}$ with node "$d : 2$" as the *tail*-node for it (Fig. 3(c)). After inserting all the transactions in similar fashion we get the final PF-tree for the database as shown in Fig. 3(d).

Based on the PF-list population technique discussed in Section 4.1 and the above example, we have the following property and lemmas of a PF-tree. For each transaction $t$ in a $TDB$, $PF(t)$ is the set of all periodic-frequent items in $t$, i.e., $PF(t) = item(t) \bigcap PF$, and is called the periodic-frequent item projection of $t$.

**Property 1.** *A PF-tree maintains a complete set of periodic-frequent item projection for each transaction in a TDB only once.*

**Lemma 1.** *Given a transactional database TDB, a max_per, and a min_sup, the complete set of all periodic-frequent item projections of all transactions in a TDB can be derived from the PF-tree for both of the max_per and min_sup.*

**Proof:** Based on Property 1, $PF(t)$ of each transaction $t$ is mapped to only one path in the 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).

**Lemma 2.** *The size of a PF-tree (without the root node) on a transactional database TDB for a max_per, and a min_sup is bounded by* $\sum_{t \in TDB} |PF(t)|$.

**Proof:** According to the PF-tree construction process and Lemma 1, each transaction $t$ contributes at best one path of the size $|PF(t)|$ to a PF-tree. Therefore, the total size contribution of all transactions can be $\sum_{t \in TDB} |PF(t)|$ at best. However, since there are usually a lot of common prefix patterns among the transactions, the size of a PF-tree is normally much smaller than $\sum_{t \in TDB} |PF(t)|$.

One can assume that the structure of a PF-tree may not be memory efficient, since it explicitly maintains *tid*s of each transaction. But, we argue that the PF-tree achieves the memory efficiency by keeping such transaction information only at the *tail*-nodes and avoiding the support count field at each node. Moreover, keeping the *tid* information in tree can also been found in literature for efficient frequent pattern mining [3], [4].

Therefore, the highly compact PF-tree structure maintains the complete information for all periodic-frequent patterns. Once the PF-tree is constructed, we use an FP-growth-based pattern growth mining technique to discover the complete set of periodic-frequent patterns from it.

### 4.3 Mining Periodic-Frequent Pattern

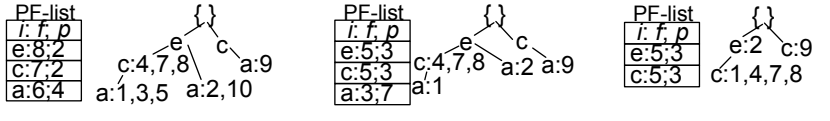
Even though both of the PF-tree and FP-tree arrange items in support-descending order, we can not directly apply the FP-growth mining on a PF-tree. The reason is that, PF-tree does not maintain the support count at each node, and it handles the *tid*-lists at *tail*-nodes. Therefore, we devise a pattern growth-based bottom-up mining technique that can handle the additional features of the PF-tree. The basic operations in mining a PF-tree for periodic-frequent patterns are $(i)$ counting length-1 periodic-frequent items, $(ii)$ constructing the prefix-tree for each periodic-frequent itemset, and $(iii)$ constructing the conditional tree from each prefix-tree. The PF-list provides the length-1 periodic-frequent items. Before discussing the prefix-tree construction process we explore the following important property and lemma of a PF-tree.

**Property 2.** *A tail-node in a PF-tree maintains the occurrence information for all the nodes in the path (from that tail-node to the root) at least in the transactions in its tid-list.*

**Lemma 3.** *Let $Z = \{a_1, a_2, ..., a_n\}$ be a path in a PF-tree where node $a_n$ is the tail-node carring the tid-list of the path. If the tid-list is pushed-up to node $a_{n-1}$, then $a_{n-1}$ maintains the occurrence information of the 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, $a_n$ maintains the occurrence information of the path $Z'$ at least in the transactions in its *tid*-list. Therefore, the same *tid*-list at node $a_{n-1}$ exactly maintains the same transaction information for $Z'$ without any lose.

Using the features revealed by the above property and lemma, we proceed to construct each prefix-tree starting from the bottom-most item, say $i$, of the PF-list. Only the prefix sub-paths of nodes labeled $i$ in the PF-tree are accumulated as the prefix-tree for $i$, say $PT_i$. Since $i$ is the bottom-most item in the PF-list, each node labeled $i$ in the PF-tree must be a *tail*-node. While constructing the $PT_i$, based on Property 2 we map the *tid*-list of every node of $i$ to all items




(a) PF-tree after removing item '*d*' (b) Prefix-tree for '*d*' (c) Conditional tree for '*d*'

**Fig. 4.** Prefix-tree and conditional tree construction with PF-tree

in the respective path explicitly in a temporary array (one for each item). It facilitates the periodicity and support calculation for each item in the PF-list of $PT_i$. Moreover, to enable the construction of the prefix-tree for the next item in the PF-list, based on Lemma 3 the *tid*-lists are pushed-up to respective parent nodes in the original PF-tree and in $PT_i$ as well. All nodes of $i$ in the PF-tree and $i$'s entry in the PF-list are deleted thereafter. Figure 4(a) shows the status of the PF-tree of Fig. 3(d) after removing the bottom-most items '*d*'. Besides, the prefix-tree for '*d*', $PT_d$ is shown in Fig. 4(b).

The conditional tree $CT_i$ for $PT_i$ is constructed by removing all non-periodic-frequent nodes from the $PT_i$. If the deleted node is a *tail*-node, its *tid*-list is pushed-up to its parent node. Figure 4(c), for instance, shows the conditional tree for '*d*', $CT_d$ constructed from the $PT_d$ of Fig. 4(b). The contents of the temporary array for the bottom item $j$ in the PF-list of $CT_i$ represent the $T^{ij}$ (i.e., the set of all *tid*s where items $i$ and $j$ occur together). Therefore, it is rather simple calculation to compute $Per(ij)$ and $Sup(ij)$ from $T^{ij}$ by generating $P^{ij}$. Then the pattern "$ij$" is generated as a periodic-frequent pattern with the periodicity and support values of $Per(ij)$ and $Sup(ij)$, respectively. The same process of creating prefix-tree and its corresponding conditional tree is repeated for further extensions of "$ij$". The whole process of mining for each item is repeated if PF-list $\neq \emptyset$.

The above bottom-up mining technique on support-descending PF-tree is efficient, because it shrinks the search space dramatically with the progress of mining process. In the next section, we present the experimental results of finding periodic-frequent patterns from the PF-tree.

## 5 Experimental Results

Since there is no existing approach to discover periodic-frequent patterns, we only investigate PF-tree's performance. All programs are written in Microsoft Visual C++ 6.0 and run with Windows XP on a 2.66 GHz machine with 1GB memory. The runtime specifies the total execution time, i.e., CPU and I/Os. The experiments are pursued on several synthetic (*T10I4D100K*) and real datasets (*chess*, *mushroom*, and *kosarak*) respectively developed at IBM Almaden Quest research group (`http://www.almaden.ibm.com/cs/quest`) and obtained from UCI Machine Learning Repository (University of California - Irvine, CA). *T10I4D100K* is a large sparse dataset with 100,000 transactions and 870 distinct items. The dense datasets *chess* and *mushroom* contain 3,196 and 8,124 transactions, and 75 and 119 distinct items respectively. In the first experiment, we study the compactness of the PF-tree on different datasets.

### 5.1 Compactness of the PF-Tree

The memory consumptions of PF-tree on the variations of *max_per* and *min_sup* values over several datasets are reported in Table 2. The first and second columns of the table respectively show the dataset-dependent different *max_per* and

**Table 2.** Memory requirements for the PF-tree

| **Dataset** (*max_per* **values**) | $\alpha$(%) | **Memory(MB)** | | |
|---|---|---|---|---|
| | | $\lambda_1$ | $\lambda_2$ | $\lambda_3$ |
| *mushroom* | 15 | 0.068 | 0.088 | 0.107 |
| $\lambda_1 = 2.0\%$, $\lambda_2 = 4.0\%$, $\lambda_3 = 6.0\%$ | 35 | 0.049 | 0.050 | 0.052 |
| *chess* | 55 | 0.015 | 0.017 | 0.019 |
| $\lambda_1 = 0.5\%$, $\lambda_2 = 0.6\%$, $\lambda_3 = 0.7\%$ | 85 | 0.014 | 0.016 | 0.016 |
| *T10I4D100K* | 1.5 | 0.288 | 5.090 | 7.349 |
| $\lambda_1 = 0.2\%$, $\lambda_2 = 0.4\%$, $\lambda_3 = 0.6\%$ | 4.5 | 0.241 | 0.281 | 0.281 |

*min_sup* values we used in the experiment. The size of the PF-tree is, therefore, shown in the last three columns for the respective thresholds.

The data in the table demonstrate that, keeping the *min_sup* fixed the memory consumption of PF-tree increases with the increase of *max_per* for almost all of the datasets. In contrast, for fixed *max_per* the tree size becomes smaller with increasing values of *min_sup*. The reason of such threshold-dependent tree size variation is that, more and more patterns become periodic-frequent with the increase of *max_per* and the decrease of *min_sup* values. Therefore, the PF-tree size increases to represent the increasing pattern set. However, it is clear from the Table 2 that, the structure of the PF-tree can easily be handled in a memory efficient manner irrespective of the dataset type (dense or sparse) or size (large or small) and threshold values. In the next experiment, we show the execution time performance of PF-tree in mining periodic-frequent patterns.

### 5.2 Execution Time of the PF-Tree

The changes on the periodicity and the support thresholds show the similar effect on execution time as of the size of PF-tree structure. Because of the space limitations, we report the results, in Fig. 5, only on *T10I4D100K* and *mushroom* datasets. The execution time shown in the graphs encompasses all phases of PF-list and PF-tree constructions, and the corresponding mining operation.

We varied the values of both thresholds as we demonstrated in the previous experiment. It can be noticed from the graphs in Fig. 5 that, for both sparse

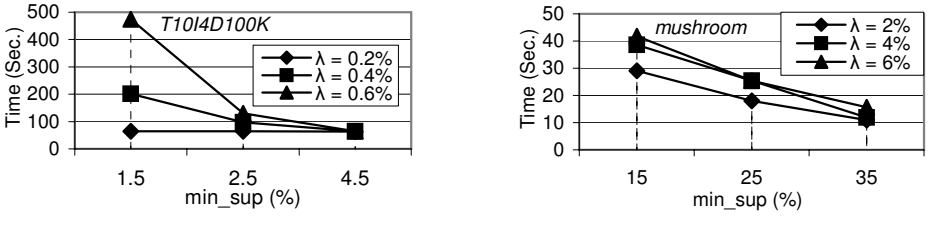



**Fig. 5.** Execution time on the PF-tree

and dense datasets PF-tree takes similar amount of time for relatively higher support threshold values for the variation of the periodicity thresholds. However, as the support thresholds go down, the gaps become wider. From another point of view, keeping the *max_per* fixed, the execution time increases (mainly for higher *max_per*s) with lowering the *min_sup*. The reason of such performance variation is that, for a fixed *min_sup* value the number and the lengths of periodic-frequent patterns increase for higher values of *max_per*. For a fixed *max_per* value, on the other hand, the same effect we get for lower *min_sup* values. In general, when mining for lower *min_sup* and higher *max_per* values, the PF-tree requires more execution time. However, as per as the database size and reasonably high *max_per* and low *min_sup* values are concerned, we see that mining periodic-frequent patterns from the corresponding PF-tree is rather time efficient for both sparse and dense datasets. The scalability study on PF-tree, discussed in the next subsection, also reflects this scenario.

### 5.3 Scalability of the PF-Tree

We study the scalability of our PF-tree on execution time and required memory by varying the number of transactions in database. We use real *kosarak* dataset for the scalability experiment, since it is a huge sparse dataset with a large number of distinct items (41,270) and transactions (990,002). We divided the dataset into five portions of 0.2 million transactions in each part. Then we investigated the performance of PF-tree after accumulating each portion with previous parts with performing periodic-frequent pattern mining each time. We fix the *max_per* to 50% and the *min_sup* to 2% of |*kosarak*| for each experiment. The experimental results are shown in Fig. 6. The time and memory in *y*-axes of the left and right graphs in Fig. 6 respectively specify the total execution time and required memory with the increase of database size. It is clear from the graphs that as the database size increases, overall tree construction and mining time, and memory requirement increase. However, PF-tree shows stable performance of about linear increase of runtime and memory consumption with respect to the database size. Therefore, it can be observed from the scalability test that PF-tree can mine the $PF_{TDB}$ over large datasets and distinct items with considerable amount of runtime and memory.

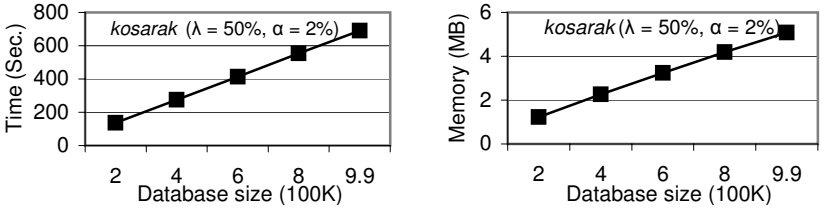



**Fig. 6.** Scalability of the PF-tree

## 6 Conclusions

In this paper, we have introduced a new interesting measure, called temporal periodicity of occurrence behavior, for frequently recurring patterns in transactional databases. We have defined such patterns as the periodic-frequent patterns under the user-given periodicity and support thresholds. This paper also shows the significance of discovering such patterns in a wide range of real-world application areas. We have provided the PF-tree, a highly compact tree structure to capture the database content, and a pattern growth-based mining technique to discover the complete set of periodic-frequent patterns on the user-given maximum periodicity and minimum support thresholds over a transactional database. The experimental results demonstrate that our PF-tree can provide the time and memory efficiency during mining the periodic-frequent pattern set. Moreover, it is highly scalable in terms of runtime and memory consumption.

## References

1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining Association Rules Between Sets of Items in Large Databases. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 207–216 (1993)
2. 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)
3. Zaki, M.J., Hsiao, C.-J.: Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure. IEEE Trans. on Knowl. and Data Eng. 17(4), 462–478 (2005)
4. Zhi-Jun, X., Hong, C., Li, C.: An Efficient Algorithm for Frequent Itemset Mining on Data Streams. In: Int. Conf. on Management of Data, pp. 474–491 (2006)
5. Minh, Q.T., Oyanagi, S., Yamazaki, K.: Mining the K-Most Interesting Frequent Patterns Sequentially. In: Corchado, E., Yin, H., Botti, V., Fyfe, C. (eds.) IDEAL 2006. LNCS, vol. 4224, pp. 620–628. Springer, Heidelberg (2006)
6. Tanbeer, S.K., Ahmed, C.F., Jeong, B.-S., Lee, Y.-K.: CP-tree: A Tree Structure for Single-Pass Frequent Pattern Mining. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS(LNAI), vol. 5012, pp. 1022–1027. Springer, Heidelberg (2008)
7. Tatavarty, G., Bhatnagar, R., Young, B.: Discovery of Temporal Dependencies between Frequent Patterns in Multivariate Time Series. In: The 2007 IEEE Symposium on Computational Intelligence and Data Mining, pp. 688–696 (2007)
8. Maqbool, F., Bashir, S., Baig, A.R.: E-MAP: Efficiently Mining Asynchronous Periodic Patterns. Int. J. of Comp. Sc. and Net. Security 6(8A), 174–179 (2006)
9. Hu, T., Sung, S.Y., Xiong, H., Fu, Q.: Discovery of Maximum Length Frequent Itemsets. Information Sciences 178, 69–87 (2008)
10. Wang, H., Perng, C.-S., Ma, S., Yu, P.S.: Demand-driven Frequent Itemset Mining Using Pattern Structures. Knowledge and Information Systems 8, 82–102 (2005)