Novel Techniques to Reduce Search Space in Periodic-Frequent Pattern Mining

R. Uday Kiran and Masaru Kitsuregawa

Institute of Industrial Science,
The University of Tokyo, Tokyo, Japan
{uday_rage,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. Periodic-frequent patterns are an important class of regularities that exist in a transactional database. Informally, a frequent pattern is said to be periodic-frequent if it appears at a regular interval specified by the user (i.e., periodically) in a database. A pattern-growth algorithm, called PFP-growth, has been proposed in the literature to discover the patterns. This algorithm constructs a *tid*-list for a pattern and performs a complete search on the *tid*-list to determine whether the corresponding pattern is a periodic-frequent or a non-periodic-frequent pattern. In very large databases, the *tid*-list of a pattern can be very long. As a result, the task of performing a complete search over a pattern's *tid*-list can make the pattern mining a computationally expensive process. In this paper, we have made an effort to reduce the computational cost of mining the patterns. In particular, we apply greedy search on a pattern's *tid*-list to determine the periodic interestingness of a pattern. The usage of greedy search facilitate us to prune the non-periodic-frequent patterns with a sub-optimal solution, while finds the periodic-frequent patterns with the global optimal solution. Thus, reducing the computational cost of mining the patterns without missing any knowledge pertaining to the periodic-frequent patterns. We introduce two novel pruning techniques, and extend them to improve the performance of PFP-growth. We call the algorithm as PFP-growth++. Experimental results show that PFP-growth++ is runtime efficient and highly scalable as well.

Keywords: Data mining, pattern mining and periodic behaviour.

1 Introduction

Since the introduction of periodic-frequent patterns in [1], the problem of finding these patterns has received a great deal of attention in data mining [2,3,4,5]. The classic application is market basket analysis, where these patterns can provide useful information pertaining to the sets of items that were not only sold frequently, but also purchased regularly by the customers. The basic model of periodic-frequent patterns is as follows [1].

Let $I = \{i_1, i_2, \cdots, i_n\}$ be the set of items. A set $X = \{i_j, \cdots, i_k\} \subseteq I$, 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

S.S. Bhowmick et al. (Eds.): DASFAA 2014, Part II, LNCS 8422, pp. 377–391, 2014.

Y is a pattern. A transactional database TDB over I is a set of transactions, i.e., $TDB = \{t_1, t_2, \cdots, t_m\}$, $m = |TDB|$, where $|TDB|$ represents 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 tid_j^X, $j \in [1, m]$. Let $TID^X = \{tid_j^X, \cdots, tid_k^X\}$, $j, k \in [1, m]$ and $j \leq k$, be the set of all transaction-ids where X occurs in TDB. The **support** of pattern X, denoted as $S(X)$, represents the number of transactions containing X in TDB, i.e., $S(X) = |TID^X|$. Let tid_p^X and tid_q^X, $p, q \in [1, m]$ and $p < q$, be the two consecutive transaction-ids where X has appeared in TDB. The number of transactions (or the time difference) between tid_p^X and tid_q^X can be defined as a **period** of X, say p_i^X. That is, $p_i^X = tid_q^X - tid_p^X$. Let $P^X = \{p_1^X, p_2^X, \cdots, p_r^X\}$, $r = |TID^X| + 1$, be the complete set of all periods of X in TDB. The **periodicity** of X, denoted as $Per(X) = max(p_1^X, p_2^X, \cdots, p_r^X)$. (It was argued in the literature that the largest *period* of a pattern can provide the upper limit of its periodic occurrence characteristic.) The pattern X is said to be **frequent** if $S(X) \geq minSup$, where $minSup$ represents the user-defined *minimum support*. The frequent pattern X is said to be **periodic-frequent** if $Per(X) \leq maxPer$, where $maxPer$ represents the user-defined *maximum periodicity*. **Please note that the *support* and *periodicity* of a pattern can also be expressed in percentage of** $|TDB|$. We now explain the model using the transactional database shown in Table 1.

Table 1. Transactional database

TID	Items	TID	Items	TID	Items	TID	Items	TID	Items
1	a, b	3	c, e, f, j	5	b, c, d	7	a, b, i	9	a, e, f, g
2	a, c, d, i	4	a, b, f, g, h	6	d, e, f	8	c, d, e	10	a, b, c

Example 1. The database shown in Table 1 contains 10 transactions. Therefore, $|TDB| = 10$. Each transaction in this database is uniquely identifiable with a transaction-id (*tid*), which also represents the timestamp of corresponding transaction. The set of items, $I = \{a,\ b,\ c,\ d,\ e,\ f,\ g,\ h,\ i,\ j\}$. The set of items '$a$' and '$b$', i.e., $\{a,\ b\}$ is a pattern. It is a 2-pattern. For the purpose of simplicity, we represent this pattern as 'ab'. The pattern 'ab' occurs in *tids* of $1, 4, 7$ and 10. Therefore, the list of *tids* containing 'ab' (or *tid*-list of 'ab'), i.e., $TID^{ab} = \{1, 4, 7, 10\}$. The *support* of '$ab$', i.e., $S(ab) = |TID^{ab}| = 4$. The complete set of all periods for this pattern are: $p_1 = 1$ $(= 1 - tid_i)$, $p_2 = 3$ $(= 4-1)$, $p_3 = 3$ $(= 7-4)$, $p_4 = 3$ $(= 10-7)$ and $p_5 = 0$ $(= 10-tid_l)$, where $tid_i = 0$ represents the initial transaction and $tid_l = |TDB| = 10$ represents the *tid* of last transaction in the transactional database. The *periodicity* of 'ab', denoted as $Per(ab) = max(1, 3, 3, 3, 0) = 3$. If the user-specified $minSup = 2$, then 'ab' is a frequent pattern because $S(ab) \geq minSup$. If the user-specified $maxPer = 3$, then the frequent pattern 'ab' is a periodic-frequent pattern because $Per(ab) \leq maxPer$.

The periodic-frequent patterns satisfy the *anti-monotonic property*. That is, "all non-empty subsets of a periodic-frequent pattern are also periodic-frequent."

Tanbeer et al. [1] have proposed a pattern-growth algorithm, called Periodic-Frequent Pattern-growth (PFP-growth), to mine the patterns. Briefly, this algorithm compresses the database into a Periodic-Frequent tree (PF-tree), and mines it recursively to discover the patterns. The nodes in PF-tree do not maintain the support count as in FP-tree. Instead, they maintain a list of *tids* (or a *tid*-list) in which the corresponding item has appeared in a database. These *tid*-lists are later aggregated to derive the final *tid*-list of a pattern (i.e., TID^X for pattern X). A complete search on this *tid*-list provides the *support* and *periodicity*, which are later used to determine whether the corresponding pattern is a periodic-frequent or a non-periodic-frequent pattern. In other words, the PFP-growth performs a complete search on a pattern's *tid*-list to determine whether it is a periodic-frequent or a non-periodic-frequent pattern.

In very large databases, the *tid*-list of a pattern can be very long. As a result, the task of performing a complete search on a pattern's *tid*-list can make the pattern mining a computationally expensive process (or the PFP-growth a computationally expensive algorithm).

In this paper, we have made an effort to reduce the computational cost of mining the periodic-frequent patterns. The contributions of this paper are as follows:

1. In this paper, we apply greedy search on a pattern's *tid*-list to determine whether it is a periodic-frequent or a non-periodic-frequent pattern.
2. A novel concept known as *local-periodicity* has been proposed in this paper. For a pattern, the *local-periodicity* corresponds to a sub-optimal solution (i.e., maximum *period* found in a subset of *tid*-list), while the *periodicity* corresponds to the global optimal solution. If the *local-periodicity* of a pattern fails to satisfy the $maxPer$, then we immediately determine the corresponding pattern as a non-periodic-frequent pattern and avoid further search on the *tid*-list to measure its *periodicity*. This results in reducing the computational cost of mining the patterns.
3. Using the concept of *local-periodicity*, we introduce two novel pruning techniques and extend them to improve the performance of PFP-growth. We call the algorithm as PFP-growth++. The proposed techniques facilitate the PFP-growth++ to prune the non-periodic-frequent patterns with a sub-optimal solution, while finds the periodic-frequent patterns with a global optimal solution. Thus, we do not miss any knowledge pertaining to periodic-frequent patterns.
4. Experimental results show that PFP-growth++ is runtime efficient and highly scalable as well.

Since the real-world is non-uniform, it was observed that mining periodic-frequent patterns with a single $minSup$ and $maxPer$ constraint leads to the "rare item problem." At high $minSup$, we miss the patterns involving rare items, and at low $minSup$, combinatorial explosion can occur producing too many patterns. To confront this problem, an effort has been made in [4] to mine the patterns using multiple $minSup$ and $maxPer$ thresholds. Amphawan et al. [5] have extended PFP-growth algorithm to mine top-k periodic-frequent patterns

in a database. As the real-world is imperfect, it was observed that the periodic-frequent pattern mining algorithms fail to discover those interesting frequent patterns whose appearances were almost periodic in the database. Uday and Reddy [2] have introduced *periodic-ratio* to capture the almost periodic behavior of the frequent patterns. Alternatively, Rashid et al. [3] have employed *standard deviation* to assess the periodic behavior of the patterns. The discovered patterns are known as regular frequent patterns. The algorithms used in all of the above works are the extensions of PFP-growth, and therefore, perform a complete search on the *tid*-list of a pattern. Thus, all these algorithms are computationally expensive to use in very large databases. The pruning techniques that are going to be discussed in this paper can be extended to improve the performance of all these algorithms. In this paper, we confine our work to finding periodic-frequent patterns using $minSup$ and $maxPer$ thresholds.

The rest of the paper is organized as follows. Section 2 describes the PFP-growth algorithm and its performance issues. Section 3 describes the basic idea and introduces the proposed PFP-growth++ algorithm. Section 4 presents the experimental evaluation on both PFP-growth and PFP-growth++ algorithms. Finally, Section 5 concludes the paper.

2 PFP-Growth and Its Performance Issues

2.1 PFP-Growth

The PFP-growth involves two steps: (i) Construction of PF-tree and (ii) Recursive mining of PF-tree to discover the patterns. Before explaining these two steps, we describe the structure of PF-tree as we also employ similar *tree* structure to discover the patterns.

Structure of PF-tree. The structure of PF-tree contains PF-list and prefix-tree. The PF-list consists of three fields – item name (i), support (f) and periodicity (p). The structure of prefix-tree is same as that of the prefix-tree in FP-tree. However, please note that the nodes in the prefix-tree of PFP-tree do not maintain the support count as in FP-tree. Instead, they explicitly maintain the occurrence information for each transaction in the tree by keeping an occurrence transaction-id, called *tid*-list, only at the last node of every transaction. Two types of nodes are 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. The *tail*-node structure is of form $I[tid_1, tid_2, \cdots, tid_n]$, where I is the node's item name and tid_i, $i \in [1, n]$, (n be the total number of transactions from the *root* up to the node) is a *tid* where item I is the last item.

Construction of PF-tree. The PFP-growth scans the database and discover periodic-frequent items (or 1-patterns) using Algorithm 1. Figure 1(a), (b) and (c) respectively show the PF-list generated after scanning the first, second and every transaction in the database (lines 2 to 11 in Algorithm 1). To reflect

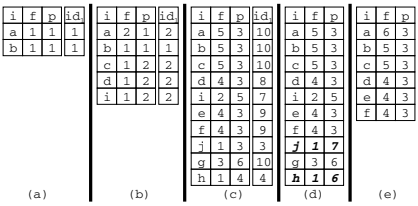

Fig. 1. Construction of PF-list. (a) After scanning first transaction (b) After scanning second transaction (c) After scanning every transaction (d) Updated *periodicity* of items and (e) Sorted list of periodic-frequent items.

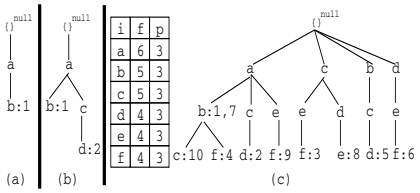

Fig. 2. Construction of PF-tree. (a) After scanning first transaction (b) After scanning second transaction and (c) After scanning every transaction.

the correct *periodicity* for an item, the p_{cur} value of every item in PF-list is re-calculated by setting $t_{cur} = |TDB|$ (line 12 in Algorithm 1). Figure 1(d) shows the updated *periodicity* of items in PF-tree. It can be observed that the *periodicity* of 'j' and 'h' items have been updated from 3 and 4 to 7 and 6, respectively. The items having $f < minSup$ or $p > maxPer$ are considered as non-periodic-frequent items and pruned from the PF-list. The remaining items are considered as periodic-frequent items and sorted in descending order of their f (or support) value (line 13 in Algorithm 1). Figure 1(e) shows the sorted list of periodic-frequent items. Let PI denote the sorted set of periodic-frequent items.

Using the FP-tree construction technique, only the items in the PI will take part in the construction of PF-tree. The *tree* construction starts by inserting the first transition, '$1 : a, b$', according to PF-list order, as shown in Figure 2(a). The tail-node '$b : 1$' carries the *tid* of the transaction. After removing the non-periodic-frequent item 'i', the second transaction is inserted into the *tree* with node '$d : 2$' as the tail-node (see Figure 2(b)). After inserting all the transactions in the database, we get the final PF-tree as shown in Figure 2(c). 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.

Mining PF-tree. To discover the patterns from PF-tree, PFP-growth employs the following steps:

i. Choosing the last item 'i' in the PF-tree as an initial suffix item, its prefix-tree (denoted as PT_i) constituting with the prefix sub-paths of nodes labeled 'i' is constructed. Figure 3(a) shows the prefix-tree for the item 'f', say PT_f.

ii. For each item 'j' in PT_i, we aggregate all of its node's *tid*-list to derive the *tid*-list of the pattern 'ij', i.e., TID^{ij}. Next, we perform a complete search on TID^{ij} to measure the *support* and *periodicity* of the pattern 'ij'. Next, we determine whether 'ij' is a periodic-frequent pattern or not by comparing its *support* and *periodicity* against *minSup* and *maxPer*, respectively. If 'ij' is a periodic-frequent pattern, then we consider 'j' is periodic-frequent in PT_i.

Algorithm 1. PF-list (TDB: transactional database, $minSup$: minimum support and $maxPer$: maximum periodicity)

1. Let t_{cur} denote the *tid* of current transaction. Let id_l be a temporary array that explicitly records the *tids* of last occurring transactions of all items in the PF-list.
2. **for** each transaction t_{cur} in TDB **do**
3. **if** t_{cur} is i's first occurrence **then**
4. Set $f = 1$, $id_l = t_{cur}$ and $p = t_{cur}$.
5. **else**
6. Set $f = f + 1$, $p_{cur} = t_{cur} - id_l$ and $id_l = t_{cur}$.
7. **if** $p_{cur} > p$ **then**
8. $p = p_{cur}$.
9. **end if**
10. **end if**
11. **end for**
12. Calculate p_{cur} value of every item in the list as $|TDB| - id_l$. Next, update the p value of every item with p_{cur} if $p_{cur} > p$.
13. Prune the items in the PF-list that have $f < minSup$ or $p > maxPer$. Consider the remaining items as periodic-frequent items and sort them with respect to their f value.

Example 2. Let us consider the last item 'e' in the PT_f. The set of *tids* containing 'e' in PT_f is $\{3, 6, 9\}$. Therefore, the *tid*-list of the pattern 'ef', i.e., $TID^{ef} = \{3, 6, 9\}$. A complete search on TID^{ef} gives $S(ef) = 3$ and $Per(ef) = 3$. Since $S(ef) \geq minSup$ and $Per(ef) \leq maxPer$, the pattern 'ef' is considered as a periodic-frequent pattern. In other words, 'e' is considered as a periodic-frequent item in PT_f. Similar process is repeated for the other items in PT_f. The PF-list in Figure 3(a) shows the *support* and *periodicity* of each item in PT_f.

iii. Choosing every periodic-frequent item 'j' in PT_i, we construct its conditional tree, CT_i, and mine it recursively to discover the patterns.

Example 3. Figure 3(b) shows the conditional-tree, CT_f, derived from PT_f. It can be observed that the items 'a', 'b', 'c' and 'd' in PT_f are not considered in the construction of CT_f. The reason is that they are non-periodic-frequent items in PT_f.

iv. After finding all periodic-frequent patterns for a suffix item 'i', we prune it from the original PF-tree and push the corresponding nodes' *tid*-lists to their parent nodes. Next, once again we repeat the steps from i to iv until the $PF\text{-}list = \emptyset$.

Example 4. Figure 3(c) shows the PF-tree generated after pruning the item 'f' in Figure 2(c). It can be observed that the *tid*-list of all the nodes containing 'f' have been pushed to their corresponding parent nodes.

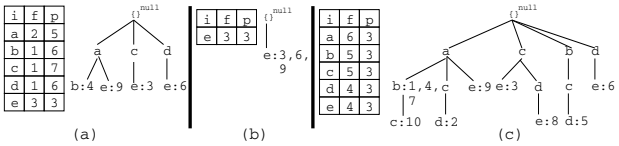

Fig. 3. Mining periodic-frequent patterns using 'f' as a suffix item. (a) Prefix-tree of f, i.e., PT_f (b) Conditional tree of 'f', i.e., CT_f and (c) PF-tree after removing item 'f'.

2.2 Performance Issues

We have observed that PFP-growth suffers from the following two performance issues:

1. The PFP-growth scans the database and constructs the PF-list with every item in the database. The non-periodic-frequent items are pruned from the list only after the scanning of database. We have observed that this approach can cause performance problems, which involves the increased updates and search costs for the items in the PF-list.

   Example 5. In Table 1, the items 'g' and 'h' have initially appeared in the transaction whose $tid = 4$. Thus, their first period is going to be 4, which is greater than the $maxPer$. In other words, these two items were non-periodic-frequent by the time they were first identified in the database. Thus, these two items need not have been included in the construction of PF-list. However, PFP-growth considers these items in the construction of PF-list. This results in the performance problems, which involves increased updates and search cost for the items in the PF-list.

2. Another performance issue of PFP-growth lies at the Step ii of mining PF-tree. That is, for every item 'j' in PT_i, PFP-growth performs a complete search on its tid-list to determine whether it is periodic-frequent or not. In very large databases, the tid-list of a pattern (or for an item 'j' in PT_i) can be generally long. In such cases, performing a complete search on a pattern's tid-list to determine whether it is periodic-frequent or not can be a computationally expensive process. Thus, PFP-growth is a computationally expensive algorithm.

   Example 6. Let us consider the item 'a' in PT_f. The tid-list of 'a' in PT_f, i.e., $TID^{af} = \{4, 9\}$. Its periods are: $p_1^{af} = 4$ $(= 4 - tid_i)$, $p_2^{af} = 5$ $(= 9 - 4)$ and $p_3^{af} = 1$ $(= tid_l - 9)$. The PFP-growth measures $periodicity = 5$ $(= max(4, 5, 1))$, and then determines 'a' is not a periodic-frequent item in PT_f. In other words, PFP-growth performs a complete search on the tid-list of 'af' to determine it is not a periodic-frequent pattern. However, such a complete search was not necessary to determine 'af' as a non-periodic-frequent pattern. It is because its first period, i.e., $p_1^{af} > maxPer$.

In the next section, we discuss our approach to address the above two performance issues of PFP-growth.

3 Proposed Algorithm

In this section, we first describe our basic idea. Next, we explain our PFP-growth++ algorithm to discover the patterns.

3.1 Basic Idea: The Local-Periodicity of a Pattern

Our idea to reduce the computational cost of mining the patterns is as follows.

> " Apply greedy search on a pattern's *tid*-list to derive its *local-periodicity*. For a pattern, the *local-periodicity* represents a sub-optimal solution, while the *periodicity* corresponds to the global optimal solution. If the *local-periodicity* of a pattern fails to satisfy the *maxPer*, then we immediately determine it as a non-periodic-frequent pattern, and avoid further search on the *tid*-list to measure its actual *periodicity*. Thus reducing the computational cost of mining the patterns."

Definition 1 defines the *local-periodicity* of a pattern X. Example 7 illustrates the definition. The correctness of our idea is shown in Lemma 1, and illustrated in Example 8.

Definition 1. ***(Local-periodicity of pattern** X**.)*** *Let* $P^X = \{p_1^X, p_2^X, \cdots, p_n^X\}$, $n = S(X) + 1$, *denote the complete set of periods for* X *in* TDB. *Let* $\widehat{P^X} = \{p_1^X, p_2^X, \cdots, p_k^X\}$, $1 \leq k \leq n$, *be an ordered set of periods of* X *such that* $\widehat{P^X} \subseteq P^X$. *The local-periodicity of* X, *denoted as loc-per*(X), *refers to the maximum period in* $\widehat{P^X}$. *That is, loc-per*$(X) = max(p_1^X, p_2^X, \cdots, p_k^X)$.

Example 7. Continuing with Example 1, the set of all *periods* for 'ab', i.e., $P^{ab} = \{1, 3, 3, 0\}$. Let $\widehat{P^{ab}} = \{1, 3\} \subset P^{ab}$. The local-periodicity of 'ab', denoted as $loc\text{-}per(ab) = max(p_j^{ab} | \forall p_j^{ab} \in \widehat{P^{ab}}) = max(1, 3) = 3$.

Lemma 1. *For the pattern* X, *if loc-per*$(X) > maxPer$, *then* X *is a non-periodic-frequent pattern.*

Proof. For the pattern X, $Per(X) \geq loc\text{-}per(X)$ as $\widehat{P^X} \subseteq P^X$.Therefore, if $loc\text{-}per(X) > maxPer$, then $Per(X) > maxPer$. Hence proved.

Example 8. In Table 1, the pattern 'af' occurs in *tids* of 4 and 9. Therefore, $TID^{af} = \{4, 9\}$. The first *period* of 'af,' i.e., $p_1^{af} = 4$ $(= tid_i - 4)$. At this point, the $loc\text{-}per(af) = p_1^{af} = 4$. Since $loc\text{-}per(af) > maxPer$, it is clear that '$af$' is a non-periodic-frequent pattern as $Per(af) \geq loc\text{-}per(af) > maxPer$.

If X is a periodic-frequent pattern, then its *local-periodicity* equals the *periodicity*. Thus, we do not miss any knowledge pertaining to periodic-frequent patterns. The correctness of this argument is based on Property 1, and shown in Lemma 2.

Property 1. For a pattern X, $loc\text{-}per(X) \leq Per(X)$ as $\widehat{P^X} \subseteq P^X$.

Lemma 2. *If X is a periodic-frequent pattern, then loc-per$(X) = Per(X)$.*

Proof. If X is a periodic-frequent pattern, then we perform a complete search on its *tid*-list, i.e., TID^X. Thus, $\widehat{P^X} = P^X$. From Property 1, it turns out that *loc-per*$(X) = Per(X)$. Hence proved.

Overall, our idea of using the greedy search technique facilitates the user to find the periodic-frequent patterns with an optimal solution, while pruning the non-periodic-frequent patterns with a sub-optimal solution. Thus, our idea reduces the computational cost of mining the patterns without missing any knowledge pertaining to periodic-frequent patterns.

Two novel pruning techniques have been developed based on the concept of *local-periodicity*. The first pruning technique addresses the issue of pruning non-periodic-frequent items (or 1-patterns) effectively. The second pruning technique addresses the issue of pruning non-periodic-frequent k-patterns, $k \geq 2$. These two techniques have been discussed in subsequent subsection.

3.2 PFP-Growth++

The proposed algorithm also involves the following steps: (i) construction of PF-tree++ and (ii) Mining PF-tree++ recursively to discover the patterns. We now discuss each of these steps.

Construction of PF-tree++. The structure of PF-tree++ consists of two components: (i) PF-list++ and (ii) prefix-tree. The PF-list++ consists of three fields – item name (i), total support (f) and *local-periodicity* (p^l). **Please note that PF-list++ do not explicitly store the** *periodicity* **of an item** i **as in the PF-list.** The structure of prefix-tree in PF-tree++, however, remains the same as in PF-tree. The structure of prefix-tree has been discussed in Section 2.

Since periodic-frequent patterns satisfy the anti-monotonic property, periodic-frequent items (or 1-patterns) play a key role in discovering the patterns effectively. To discover these items, we employ a pruning technique which is based on the concepts of 2-Phase Locking [6] (i.e., 'expanding phase' and 'shrinking phase'). We now discusses the pruning technique:

- **Expanding phase:** In this phase, we insert every new item found in a transaction into the PF-list++. Thus, expanding the length of PF-list++. This phase starts from the first transaction (i.e., $tid = 1$) in the database and ends when the *tid* of the transaction equals to $maxPer$ (i.e., $tid = maxPer$). If the *tid* of a transaction is greater than the $maxPer$, then we do not insert any new items found in the corresponding transaction into the PF-list++. It is because these items are non-periodic-frequent items as their first *period* (or local-periodicity) fails to satisfy the user-defined $maxPer$ threshold.
- **Shrinking phase:** In this phase, we delete the non-periodic-frequent items from the PF-list++. Thus, shrinking the length of PF-list++. The non-periodic-frequent items are those items that have a *period* (or *local-periodicity*)

greater than the *maxPer*. This phase starts right after the completion of expansion phase (i.e., when the *tid* of a transaction equal to $maxPer + 1$) and continues until the end of database.

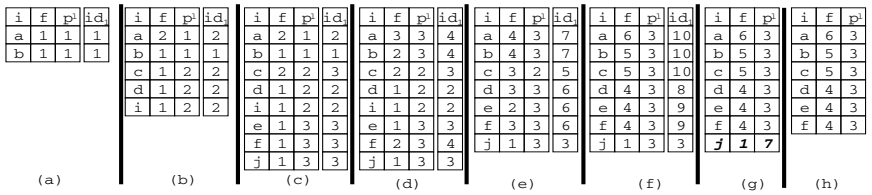

Fig. 4. The construction of PF-list++. (a) After scanning the first transaction (b) After scanning the second transaction (c) After scanning the third transaction (d) After scanning the fourth transaction (e) After scanning the seventh transaction (f) After scanning all transactions (g) Measuring actual *periodicity* for items and (h) The sorted list of periodic-frequent items.

Algorithm 2 shows the construction of PF-list++ using the above two phases. We illustrate the algorithm using the database shown in Table 1. The 'expanding phase' starts from the first transaction. The scan on the first transaction with $t_{cur} = 1$ results in inserting the items '*a*' and '*b*' in to the list with $f = 1$, $p^l = 1$ and $id_l = 1$ (lines 4 to 6 in Algorithm 2). The resultant PF-list++ is shown in Figure 4(a). The scan on the second transaction with $t_{cur} = 2$ results in adding the items '*c*', '*d*' and '*i*' into the PF-list++ with $f = 1$, $id_l = 2$ and $p^l = 2$. Simultaneously, the f and id_l values of '*a*' are updated to 2 and 2, respectively (lines 7 and 8 in Algorithm 2). The resultant PF-list++ was shown in Figure 4(b). Similarly, the scan on the third transaction with $t_{cur} = 3$ results in adding the items '*e*', '*f*' and '*j*' into the PF-list++ with $f = 1$, $id_l = 3$ and $p^l = 3$. In addition, the f, p and id_l values of '*c*' are updated to 2, 2 and 3, respectively. Figure 4(c) shows the PF-list++ generated after scanning the third transaction. Since $maxPer = 3$, the expanding phase ends at $t_{cur} = 3$. The 'shrinking phase' begins from $t_{cur} = 4$. The scan on the fourth transaction, '$4 : a, b, f, g, h$', updates the f, p^l and id_l of the items '*a*', '*b*' and '*f*' accordingly as shown in Figure 4(d). Please observe that we have **not added** the new items '*g*' and '*h*' in the PF-list++ as their *period* (or *local-periodicity*) is greater than the *maxPer* (lines 11 to 16 in Algorithm 2). Similar process is repeated for the other transactions in the database, and the PF-list++ is constructed accordingly. Figure 4(e) shows the PF-list++ constructed after scanning the seventh transaction. It can be observed that the non-periodic-frequent item '*i*' was pruned from the PF-list++. It is because its *local-periodicity* (or $p^l = 5$ $(= t_{cur} - id_l)$) has failed to satisfy the *maxPer* (line 15 in Algorithm 2). Figure 4(f) shows the **initial PF-list++** constructed after scanning the entire database. Figure 4(g) shows the updated *periodicity* of all items in the PF-list++ (line 19 in Algorithm 2). It can be observed that the p^l value of '*j*' has been updated from 3 to 7. Figure 4(h) shows

the set of periodic-frequent items sorted in descending order of their frequencies. Let PI denote this sorted list of periodic-frequent items.

Using FP-tree construction technique, we perform another scan on the database and construct prefix-tree in PI order. The construction of prefix-tree has been discussed in Section 2. Figure 5 shows the PF-tree++ generated after scanning the database shown in Table 1. Since the *local-periodicity* of a periodic-frequent item (or 1-pattern) is same as its *periodicity* (see Lemma 2), the constructed PF-tree++ resembles that of the PF-tree in PFP-growth. However, there exists a key difference between these two *trees*, which we will discuss while mining the patterns from PF-tree++.

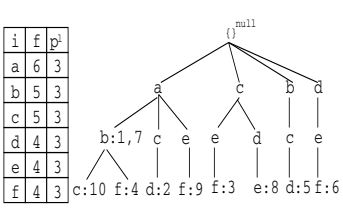

Fig. 5. The PF-tree++ generated after scanning every transaction in the database

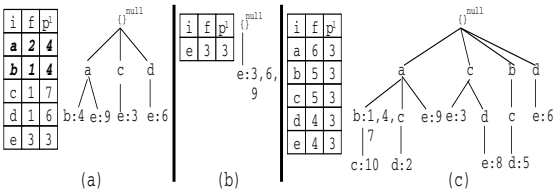

Fig. 6. Mining periodic-frequent patterns using 'f' as suffix item. (a) Prefix-tree of suffix item 'f', i.e., PT_f (b) Conditional tree of suffix item 'f', i.e., CT_f and (c) PF-tree++ after pruning item 'f'.

Mining PF-tree++. Choosing the last item 'i' in the PF-list++, we construct its prefix-tree, say PT_i, with the prefix sub-paths of nodes labeled 'i' in the PF-tree++. 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, we map the *tid*-list of every node of 'i' to all items in the respective path explicitly in a temporary array. It facilitates the construction of *tid*-list for each item 'j' in the PF-list++ of PT_i, i.e., TID^{ij}. The length of TID^{ij} gives the *support* of 'ij'. Algorithm 3 is used to measure the *local-periodicity* of 'ij.' This algorithm applies greedy search on the *tid*-list to identify whether j is periodic-frequent or not in PT_i. The pruning technique used in this algorithm is as follows:

"If $loc\text{-}per(X) > maxPer$, then X is a non-periodic-frequent pattern."

If j is periodic-frequent in PT_i, then its p^l denotes its actual *periodicity* in the database. However, if 'j' is non-periodic-frequent in PT_i, then its p^l denotes the *local-periodicity* (or the *period*) which has failed to satisfy the $maxPer$. The correctness of this technique has already been shown in Lemma 2.

Figure 6(a) shows the prefix-tree of item 'f', PT_f. The set of items in PT_f are 'a', 'b', 'c', 'd' and 'e.' Let us consider an item 'a' in PT_f. The *tid*-list of the nodes containing 'a' in PT_f gives $TID^{af} = \{4, 9\}$. The *support* of 'af', $S(af) = 2(= |TID^{af}|)$. As $S(af) \geq 2$ $(= minSup)$, we determine 'af' as a frequent pattern. Next, we pass TID^{af} as an array in Algorithm 3 to find whether 'af' is

a periodic-frequent or a non-periodic-frequent pattern. The first period, $p_{cur} = 4\ (= 4 - 0)$ (line 1 in Algorithm 3). As $p_{cur} > maxPer$, we determine 'af' as a non-periodic-frequent pattern and return $p^l = p_{cur} = 4$ (lines 2 to 4 in Algorithm 3). Thus, we prevent the complete search on the *tid*-list of a non-periodic-frequent pattern. Similar process is applied for the remaining items in the PT_f. The PF-list++ in Figure 6(a) shows the *support* and *local-periodicity* of items in PT_f. It can be observed that the p^l value of non-periodic-frequent items, 'a' and 'b', in PT_f are set to 4 and 4, respectively. Please note that these values are not their actual *periodicity* values. The actual *periodicity* of 'a' and 'b' in PT_f are 5 and 6, respectively (see Figure 3(a)). This is the key difference between the PFP-tree++ and PFP-tree.

The conditional tree, CT_i, is constructed by removing all non-periodic-frequent items from the PT_i. If the deleted node is a *tail*-node, its *tid*-list is pushed up to its parent node. Figure 6(b), for instance, shows the conditional tree for 'f', say CT_f, from PT_f. The same process of creating prefix-tree and its corresponding conditional tree is repeated for the further extensions of 'ij'. Once the periodic-frequent patterns containing 'f' are discovered, the item 'f' is removed from the original PF-tree++ by pushing its node's *tid*-list to its respective parent nodes. Figure 6(c) shows the resultant PF-tree++ after pruning the item 'f'. The whole process of mining for each item in original PF-tree++ is repeated until its 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.

4 Experimental Results

The algorithms, PFP-growth and PFP-growth++, are written in Java and run with Ubuntu 10.04 operating system on a 2.66 GHz machine with 4GB memory. The runtime specifies the total execution time, i.e., CPU and I/Os. We pursued experiments on synthetic (T10I4D100K and T10I4D1000K) and real-world (Retail and Kosarak) datasets. The T10I4D100K dataset contains 100,000 transactions with 1000 items. The T10I4D1000K dataset contains 1,000,000 transactions with 1000 items. The Retail dataset [7] contains 88,162 transactions with 16,470 items. The Kosarak dataset is a very large dataset containing 990,002 transactions and 41,270 distinct items.

The *maxPer* threshold varies from 0% to 100%. In this paper, we vary the *maxPer* threshold from 1% to 10%. The reason is as follows. Very few (almost nil) periodic-frequent patterns are discovered in these databases for the *maxPer* values less than the 1%. Almost all frequent patterns are discovered as periodic-frequent patterns when the *maxPer* values are greater than the 10%.

4.1 Discovering Periodic-Frequent Patterns

Figure 7(a) and (b) respectively show the number of periodic-frequent patterns generated in T10I4D100K and Retail datasets at different *maxPer* thresholds.

Algorithm 2. PF-list++ (TDB: transactional database, $minSup$: minimum support and $maxPer$: maximum periodicity)

1. **for** each transaction $t \in TDB$ **do**
2. **if** $t_{cur} < maxPer$ **then**
3. /*Expanding phase*/
4. **if** t_{cur} is i's first occurrence **then**
5. Insert i into the list and set $f = 1$, $id_l = t_{cur}$ and $p^l = t_{cur}$.
6. **else**
7. Calculate $p_{cur} = t_{cur} - id_l$. Set $f = f + 1$, $id_l = t_{cur}$ and $p^l = (p_{cur} > p^l)?p_{cur} : p^l$.
8. **end if**
9. **else**
10. /*Shrinking phase*/
11. Calculate $p_{cur} = t_{cur} - id_l$.
12. **if** $p_{cur} < maxPer$ **then**
13. Set $f = f + 1$, $id_l = t_{cur}$ and $p^l = (p_{cur} > p^l)?p_{cur} : p^l$.
14. **else**
15. Remove i from PF-list++.
16. **end if**
17. **end if**
18. **end for**
19. For each item in the PF-list++, we re-calculate p_{cur} value as $|TDB| - id_l$, and prune the non-periodic-frequent items.

The $minSup$ values are set at 0.01% and 0.01% in T10I4D100K and Retail datasets, respectively. The usage of a low $minSup$ value in these datasets facilitate us to discover periodic-frequent patterns involving both frequent and rare items. It can be observed that the increase in $maxPer$ has increased the number of periodic-frequent patterns. It is because some of the periods of a patterns that are earlier considered aperiodic have been considered periodic with the increase in $maxPrd$ threshold.

4.2 Runtime for Mining Periodic-Frequent Patterns

Figure 8(a) and (b) shows the runtime taken by PFP-growth and PFP-growth++ algorithms to discover periodic-frequent patterns at different $maxPer$ thresholds in T10I4D100K and Retail datasets, respectively. The following three observations can be drawn from these figures: (i) Increase in $maxPer$ threshold has increased the runtime for both the algorithms. It is because of the increase in number of periodic-frequent pattern with the increase in $maxPer$ threshold. (ii) At any $maxPer$ threshold , the runtime of PFP-growth++ is no more than the runtime of PFP-growth. It is because of the greedy search technique employed by the PFP-growth++ algorithm. (iii) At a low $maxPer$ value, the PFP-growth++ algorithm has outperformed the PFP-growth by an order of magnitude. It is because the PFP-growth++ has performed only partial search on the tid-lists of non-periodic-frequent patterns.

Algorithm 3. CalculateLocalPeriodicity (TID: an array of tid's containing X.)

1. Set $p^l = -1$ and $p_{cur} = TID[0]$ $(= TID[1] - 0)$.
2. **if** $p_{cur} > maxPer$ **then**
3. return p_{cur}; /*(as p^l value).*/
4. **end if**
5. **for** $i = 1$; $i < TID.length - 1$;++i **do**
6. Calculate $p_{cur} = TID[i+1] - TID[i]$.
7. $p^l = (p_{cur} > p^l)?p_{cur} : p^l$
8. **if** $p^l > maxPer$ **then**
9. return p^l;
10. **end if**
11. **end for**
12. Calculate $p_{cur} = |TDB| - TID[TID.length]$, and repeat the steps numbered from 7 to 10.

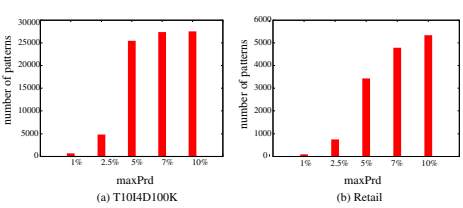

Fig. 7. Periodic-frequent patterns discovered at different $maxPer$ thresholds in various datasets.

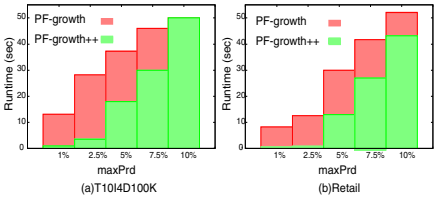

Fig. 8. Runtime comparison of PF-growth and PF-growth++ algorithms at different $maxPer$ thresholds in various datasets.

4.3 Scalability Test

We study the scalability of PFP-growth and PFP-growth++ algorithms on execution time by varying the number of transactions in $T10I4D1000K$ and $Kosarak$ datasets. In the literature, these two datasets were widely used to study the scalability of algorithms. The experimental setup was as follows. Each dataset was divided into five portions with 0.2 million transactions in each part. Then, we investigated the performance of both algorithms after accumulating each portion with previous parts. We fixed the $minSup = 1\%$ and $maxPer = 2.5\%$.

Figure 9(a) and (b) shows the runtime requirements of PFP-growth and PFP-growth++ algorithms in $T10I4D1000K$ and $Kosarak$ datasets, respectively. It can be observed that the increase in dataset size has increased the runtime of both the algorithms. However, the proposed PFP-growth++ has taken relatively less

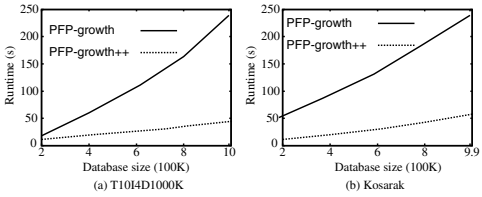

Fig. 9. Scalability of PFP-growth and PFP-growth++ algorithms

runtime than the PFP-growth. In particular, as the dataset size increases, the proposed PFP-growth++ algorithm has outperformed PFP-growth by an order of magnitude. The reason is as follows. The *tid*-list of a pattern gets increased with the increase in the database size. The complete search on the *tid*-list of both periodic-frequent and non-periodic-frequent patterns by PFP-growth has increased its runtime requirements. The partial search on the *tid*-list of a non-periodic-frequent pattern has facilitated the PFP-growth++ to reduce its runtime requirements.

5 Conclusions and Future Work

In this paper, we have employed greedy search technique to reduce the computational cost of mining the periodic-frequent patterns. The usage of greedy search technique facilitated the user to find the periodic-frequent patterns with the global optimal solution, while pruning the non-periodic-frequent patterns with a sub-optimal solution. Thus, reducing the computational cost of mining the patterns without missing any useful knowledge pertaining to periodic-frequent patterns. Two novel pruning techniques have been introduced to discover the patterns effectively. A pattern-growth algorithm, known as PFP-growth++, has been proposed to discover the patterns. Experimental results show that the proposed PFP-growth++ is runtime efficient and scalable as well.

As a part of future work, we would like to extend the proposed concepts to mine partial periodic-frequent patterns in a database. In addition, we would like to investigate alternative search techniques to further reduce the computational cost of mining the patterns.

References

1. Tanbeer, S.K., Ahmed, C.F., Jeong, B.-S., Lee, Y.-K.: Discovering periodic-frequent patterns in transactional databases. In: Theeramunkong, T., Kijsirikul, B., Cercone, N., Ho, T.-B. (eds.) PAKDD 2009. LNCS, vol. 5476, pp. 242–253. Springer, Heidelberg (2009)
2. Kiran, R.U., Reddy, P.K.: An alternative interestingness measure for mining periodic-frequent patterns. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 183–192. Springer, Heidelberg (2011)
3. Rashid, M. M., Karim, M. R., Jeong, B.-S., Choi, H.-J.: Efficient mining regularly frequent patterns in transactional databases. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part I. LNCS, vol. 7238, pp. 258–271. Springer, Heidelberg (2012)
4. Uday Kiran, R., Krishna Reddy, P.: Towards efficient mining of periodic-frequent patterns in transactional databases. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part II. LNCS, vol. 6262, pp. 194–208. Springer, Heidelberg (2010)
5. Amphawan, K., Lenca, P., Surarerks, A.: Mining top-k periodic-frequent pattern from transactional databases without support threshold. In: Papasratorn, B., Chutimaskul, W., Porkaew, K., Vanijja, V. (eds.) IAIT 2009. CCIS, vol. 55, pp. 18–29. Springer, Heidelberg (2009)
6. Gray, J.: Notes on data base operating systems. In: Advanced Course: Operating Systems, pp. 393–481 (1978)
7. Brijs, T., Goethals, B., Swinnen, G., Vanhoof, K., Wets, G.: A data mining framework for optimal product selection in retail supermarket data: The generalized profset model. In: KDD, pp. 300–304 (2000)