Efficient Discovery of Periodic-Frequent Patterns in Very Large Databases

R. Uday Kiran[a,b], Masaru Kitsuregawa[a,c], P. Krishna Reddy[d]

[a]*The University of Tokyo, Japan*
[b]*National Institute of Communication Technology, Japan*
[c]*National Institute of Informatics, Japan*
[d]*International Institute of Information Technology-Hyderabad, India*

Abstract

Periodic-frequent patterns (or itemsets) are an important class of regularities that exist in a transactional database. Most of its mining algorithms discover all those frequent patterns that satisfy the user-specified maximum periodicity constraint. This constraint controls the maximum inter-arrival time of a pattern in a database. The time complexity to measure *periodicity* of a pattern is $O(n)$, where n represents the number of timestamps at which the corresponding pattern has appeared in a database. As n usually represents a high value in voluminous databases, determining the *periodicity* of every candidate pattern in the itemset lattice makes the periodic-frequent pattern mining a computationally expensive process. This paper introduces a novel approach to address this problem. Our approach determines the periodic interestingness of a pattern by adopting greedy search. The basic idea of our approach is to discover all periodic-frequent patterns by eliminating aperiodic patterns based on suboptimal solutions. The best and worst case time complexities of our approach to determine the periodic interestingness of a frequent pattern are $O(1)$ and $O(n)$, respectively. We introduce two pruning techniques and propose a pattern-growth algorithm to find these patterns efficiently. Experimental results show that our algorithm is runtime efficient and highly scalable as well.

Email addresses: uday_rage@tkl.iis.u-tokyo.ac.jp (R. Uday Kiran), kitsure@tkl.iis.u-tokyo.ac.jp (Masaru Kitsuregawa), pkreddy@iiit.ac.in (P. Krishna Reddy)

URL: http://researchweb.iiit.ac.in/~uday_rage/index.html (R. Uday Kiran), http://www.tkl.iis.u-tokyo.ac.jp/Kilab/Members/memo/kitsure_e.html (Masaru Kitsuregawa), http://faculty.iiit.ac.in/~pkreddy/index.html (P. Krishna Reddy)

Keywords:
Data mining, knowledge discovery, frequent patterns, periodicity, greedy search

1. Introduction

Frequent pattern (or itemset) mining is an important knowledge discovery technique. It typically involves finding all patterns that are occurring frequently within a transactional database. These patterns play a key role in discovering associations [1], correlations [6, 30], episodes [26], multi-dimensional patterns [24], diverse patterns [34], emerging patterns [9], and so on. The popular adoption and successful industrial application of frequent patterns has been hindered by a major obstacle: "*frequent pattern mining often generates too many patterns, and majority of them may be found insignificant depending on application or user requirements.*" When confronted with this problem in real-world applications, researchers have tried to reduce the desired set by finding user interest-based frequent patterns such as maximal frequent patterns [13], demand driven patterns [**?**], utility patterns [41], constraint-based patterns [32], diverse-frequent patterns [37], top-k patterns [18] and periodic-frequent patterns [38]. This paper focuses on efficient discovery of periodic-frequent patterns.

An important criterion to assess the interestingness of a frequent pattern is its temporal occurrences within a database. That is, whether a frequent pattern is occurring periodically, irregularly, or mostly at specific time intervals in a database. The class of frequent patterns that are occurring periodically within a database are known as **periodic-frequent patterns**. These patterns are ubiquitous and play a key role in many applications such as finding co-occurring genes in biological datasets [42], improving performance of recommender systems [35], intrusion detection in computer networks [25] and discovering events in Twitter [22]. A classic application to illustrate the usefulness of these patterns is market-basket analysis. It analyzes how regularly the set of items are being purchased by the customers. An example of a periodic-frequent pattern is as follows:

$$\{Bed,\ Pillow\} \quad [support = 10\%,\ \ periodicity = 1\ hour].$$

The above pattern says that 10% of customers have purchased the items '*Bed*' and '*Pillow*' at least once in every hour. The basic model of periodic-frequent patterns is as follows [38]:

Let $I = \{i_1, i_2, \cdots, i_n\}$, $1 \leq n$, be the set of items. Let $X \subseteq I$ be a **pattern** (or an itemset). A pattern containing β number of items is called a β**-pattern**. A **transaction**, $tr_k = (ts_k, Y)$, $1 \leq k$, is a tuple, where $ts_k \in \mathbb{R}$ represents the timestamp of

Y pattern. For a transaction $tr_k = (ts_k, Y)$, such that $X \subseteq Y$, it is said that X occurs in tr_k and such timestamp is denoted as ts_k^X. A **transactional database** TDB over I is a set of transactions, $TDB = \{tr_1, \cdots, tr_m\}$, $m = |TDB|$, where $|TDB|$ can be defined as the number of transactions in TDB. Let $TS^X = \{ts_j^X, \cdots, ts_k^X\}$, $j,k \in [1,m]$ and $j \leq k$, be an **ordered set of timestamps** where X has occurred in TDB. In this paper, we call this list of timestamps of X as **ts-list** of X. The number of transactions containing X in TDB is defined as the **support** of X and denoted as $Sup(X)$. That is, $Sup(X) = |TS^X|$. Let ts_q^X and ts_r^X, $j \leq q < r \leq k$, be the two consecutive timestamps in TS^X. The time difference (or an inter-arrival time) between ts_r^X and ts_q^X can be defined as a ***period*** of X, say p_a^X. That is, $p_a^X = ts_r^X - ts_q^X$. Let $P^X = \{p_1^X, p_2^X, \cdots, p_r^X\}$ be the set of periods for pattern X. The ***periodicity*** of X, denoted as $Per(X) = maximum(p_1^X, p_2^X, \cdots, p_r^X)$. The pattern X is a **frequent pattern** if $Sup(X) \geq minSup$, where *minSup* refers to the user-specified *minimum support* constraint. The frequent pattern X is said to be **periodic-frequent** if $Per(X) \leq maxPer$, where *maxPer* refers to the user-specified *maximum periodicity* constraint. The problem definition of periodic-frequent pattern mining involves discovers all those patterns in TDB that satisfy the user-specified *minSup* and *maxPer* constraints. Please note that both *support* and *periodicity* of a pattern can be described in percentage of $|TDB|$.

Table 1: Transactional Database

ts	Items
1	*ab*
2	*acdi*
3	*cefj*
4	*abfgh*
5	*bcd*

ts	Items
6	*def*
7	*abi*
8	*cde*
9	*abef*
10	*acg*

Table 2: Periodic-frequent patterns discovered from Table 1

P	*Sup*	*Per*
a	6	3
b	5	3
c	5	3
d	4	3
e	4	3

P	*Sup*	*Per*
f	4	3
ab	4	3
cd	3	3
ef	3	3

EXAMPLE 1. Consider the transactional database shown in Table 1. This database contains 10 transactions. Therefore, $|TDB| = 10$. Each transaction in this database is uniquely identifiable with a timestamp (*ts*). The set of all items in TDB, i.e., $I = \{a,\ b,\ c,\ d,\ e,\ f,\ g,\ h,\ i,\ j\}$. The set of items '*a*' and '*b*', i.e., $\{a,\ b\}$ is a pattern. For brevity, we represent this pattern as '*ab*'. This pattern contains two items. Therefore, it is a 2-pattern. The pattern '*ab*' appears at the timestamps of 1,4,7 and 9. Therefore, the list of timestamps containing '*ab*' (or *ts*-list of '*ab*') is $\{1,4,7,9\}$. In other words, $TS^{ab} = \{1,4,7,9\}$. The *support* of '*ab*' is the

size of TS^{ab}. Therefore, $Sup(ab) = |TS^{ab}| = 4$. The periods for this pattern are: $p_1^{ab} = 1$ $(= 1 - ts_{initial})$, $p_2^{ab} = 3$ $(= 4 - 1)$, $p_3^{ab} = 3$ $(= 7 - 4)$, $p_4^{ab} = 2$ $(= 9 - 7)$ and $p_5^{ab} = 1$ $(= ts_{final} - 9)$, where $ts_{initial} = 0$ represents the timestamp of initial transaction and $ts_{final} = |TDB| = 10$ represents the timestamp of final transaction in the database. The *periodicity* of *ab*, i.e., $Per(ab) = maximum(1,3,3,2,1) = 3$. If the user-defined $minSup = 3$, then '*ab*' is a frequent pattern because $Sup(ab) \geq minSup$. If the user-defined $maxPer = 3$, then the frequent pattern '*ab*' is said to be a periodic-frequent pattern because $Per(ab) \leq maxPer$. The complete set of periodic-frequent patterns discovered from Table 1 are shown in Table 2.

The space of items in a transactional database gives rise to an itemset lattice. Figure 1 shows the itemset lattice for the items '*a*', '*b*' and '*c*.' This lattice is a conceptualization of search space while finding the user interest-based patterns. Tanbeer et al. [38] have introduced a pattern-growth algorithm, called Periodic-Frequent Pattern-growth (PFP-growth), to mine the periodic-frequent patterns. This algorithm generates periodic-frequent patterns by applying depth-first search in the itemset lattice. From a singleton periodic-frequent pattern *i*, successively larger periodic-frequent patterns are discovered by adding one item at a time.

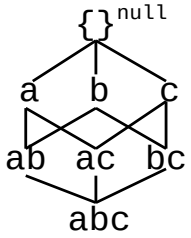


Figure 1: Itemset lattice generated for the items '*a*,' '*b*' and '*c*'

The measure, *periodicity*, plays a key role in periodic-frequent pattern mining. This measure ensures that the anti-monotonic property (see Property 1) of frequent patterns still holds for periodic-frequent patterns. Measuring the *periodicity* of a pattern requires a complete scan on its ts-list. As a result, the time complexity of finding *periodicity* of a pattern is $O(n)$, where n represents the number of timestamps at which the corresponding pattern has appeared within a database. **As n typically represents a very large number in voluminous databases, measuring the *periodicity* of every candidate pattern in the huge itemset lattice makes the periodic-frequent pattern mining a computationally expensive process or impracticable in real-world applications.**

EXAMPLE 2. Let '*xy*' be a frequent pattern in a very large transactional database appearing randomly at the timestamps, say 5, 9, 20, 23, 27, 50 and so on. The existing state-of-the-art algorithms measure the *periodicity* of this pattern by performing a complete search on its huge list of timestamps. In the next step, they determine '*xy*' as either periodic or aperiodic by comparing its *periodicity* against the user-specified *maxPrd*. In other words, current approaches determine the periodic interestingness of this pattern by performing a complete search on its huge list of timestamps. This approach of determining the periodic interestingness of every candidate pattern in the itemset lattice makes the periodic-frequent pattern mining a computationally expensive process.

Property 1. *(Anti-monotonic property of periodic-frequent patterns.) For a pattern X, if* $Sup(X) \geq minSup$ *and* $Per(X) \leq maxPer$*, then* $\forall Y \subset X$ *and* $Y \neq \emptyset$*,* $Sup(Y) \geq minSup$ *and* $Per(X) \leq maxPer$*.*

Reducing the computational cost of periodic-frequent pattern mining is a non-trivial and challenging task. The main reason is that we cannot sacrifice any information pertaining to periodic-frequent patterns in order to reduce the computational cost.

With this motivation, we propose an approach to reduce the computational cost of finding the periodic-frequent patterns. Our approach determines the periodic interestingness of a pattern by adopting greedy search on its ts-list. The usage of greedy search achieves two important tasks. First, reduces the need of complete search on the ts-lists of aperiodically occurring patterns by identifying them based on sub-optimal solution. Second, finds global optimal solution (i.e., *periodicity*) for every periodic-frequent pattern. As a result, our approach reduces the computational cost without missing any information about periodic-frequent patterns. The best and the worst case time complexities of our approach are $O(1)$ and $O(n)$, respectively.

Our contributions are as follows:

1. A novel concept known as *local-periodicity* has been proposed in this paper. For a pattern, *local-periodicity* and *periodicity* correspond to sub-optimal solution and global optimal solution, respectively. If the *local-periodicity* of a pattern fails to satisfy the user-defined *maximum periodicity* threshold value, then we immediately determine the corresponding pattern as an aperiodic-frequent pattern and avoid further search on its *ts*-list. As a result, the usage of *local-periodicity* reduces the computational cost of finding the periodic-frequent patterns.

2. Two novel pruning techniques have been introduced based on the concept of *local-periodicity*. First pruning technique is based on the concepts of 2-phase locking [14], and is used to discover periodic-frequent patterns containing only one item. Second pruning technique is used to discover periodic-frequent patterns containing more than one item efficiently.
3. A pattern-growth algorithm, called Periodic-Frequent pattern-growth++ (PFP-growth++), has been proposed using our pruning techniques.
4. A performance study has been conducted to compare the performance of PFP-growth++ against PFP-growth. Our study shows that PFP-growth++ is runtime efficient and highly scalable than the PFP-growth.

The PFP-growth++ was first proposed briefly in [19]. In this paper, we provide theoretical correctness for PFP-growth++ and evaluate the performance of our pruning techniques by conducting extensive experiments on both synthetic and real-world databases.

The rest of the paper is organized as follows. Section 2 describes the related work on finding periodically occurring patterns. Section 3 describes the working of PFP-growth and its performance issues. Section 4 introduces our PFP-growth++. Section 5 reports the experimental evaluation of PFP-growth and PFP-growth++ algorithms. Finally, Section 6 concludes the paper with future research directions.

2. Related Work

A time series is a collection of events obtained from sequential measurements over time. Periodic pattern mining involves discovering all those patterns that are exhibiting either complete or partial cyclic repetitions in a timeseries. The problem of finding periodic patterns has been widely studied in [3, 4, 7, 15, 16, 40]. The basic model used in all these studies, however, remains the same. That is, this model takes into account a timeseries as a symbolic sequence and finds all periodic patterns by employing the following two steps:

1. Partition the symbolic sequence into distinct subsets (or period-segments) of a fixed length (or *period*).
2. Discover all periodic patterns that satisfy the user-defined *minimum support* (*minSup*). *Minsup* controls the minimum number of period-segments in which a pattern must appear.

EXAMPLE 3. Given the timeseries $TS = a\{bc\}baebace$ and the user-defined *period* as 3, TS is divided into three periodic-segments: $TS_1 = a\{bc\}b$, $TS_2 = aeb$ and $TS_3 = ace$. Let $\{a \star b\}$ be a pattern, where '$\star$' denotes a wild character that can represent any single set of events. This pattern appears in the periodic-segments of TS_1 and TS_2. Therefore, its *support* count is 2. If the user-defined *minSup* is 2, then $\{a \star b\}$ represents a periodic pattern.

A major limitation of the above studies is that they do not take into account the actual temporal information of events within a timeseries.

Ozden et al. [31] have enhanced the transactional database by a time attribute that describes the time when a transaction has appeared, investigated the periodic behavior of the patterns to discover cyclic association rules. In this study, a database is fragmented into non-overlapping subsets with respect to time. The association rules that are appearing in at least a certain number of subsets are discovered as cyclic association rules. By fragmenting the data and counting the number of subsets in which a pattern occurs greatly simplifies the design of the mining algorithm. However, the drawback is that patterns (or association rules) that span multiple windows cannot be discovered.

Tanbeer et al. [38] have proposed a simplified model to discover periodically appearing frequent patterns without any data fragmentation. A pattern-growth algorithm, PFP-growth, has been proposed to discover periodic-frequent patterns. Uday and Reddy [20] and Akshat et al. [36] have enhanced Tanbeer's model to address the *rare item problem* [39], which involves discovering periodic-frequent patterns involving both frequent and rare items. Amphawan et al. [2] have focussed on finding top-k periodic-frequent patterns. As the real-world is imperfect, Uday and Reddy [21] have extended [38] to discover those frequent patterns that have exhibited partial periodic behavior within a database. Alternatively, Rashid et al. [33] have employed standard deviation of *periods* as an alternative criterion to assess the periodic behavior of frequent patterns. The discovered patterns are known as regular frequent patterns. The algorithms employed to discover periodic-frequent patterns in all of the above studies are the extensions of PFP-growth; therefore, determine the periodic interestingness of a pattern by performing a complete search on its ts-list. As a result, all of these algorithms are computationally expensive to use in very large databases. Some of our pruning techniques that we are going to discuss in this paper can be extended to improve the performance of all these algorithms. However, we confine our work to reduce the computational cost of finding periodic-frequent patterns as defined in [38].

Table 3: Comparison of issues addressed by our approach and related current approaches. The issues 1, 2 and 3 represent the 'inability to capture temporal information of the items,' 'fragmentation of data,' and 'computational expensiveness,' respectively. The symbols '✓' and '×' represent the issue addressed and have not addressed by the corresponding work, respectively

	Issue 1	Issue 2	Issue 3
Han et al. [15]	×	×	✓
Ozden et al. [31]	✓	×	×
Tanbeer et al. [38]	✓	✓	×
Our approach	✓	✓	✓

The problem of finding sequential patterns [28] and frequent episodes [26, 23] has received a great deal of attention. However, it should be noted that these studies do not take into account the *periodicity* of a pattern.

Finding partial periodic patterns [11], motifs [29], and recurring patterns [27] has also been studied in time series; however, the focus was on finding numerical curve patterns rather than symbolic patterns.

Table 3 compares the issues addressed by our approach against the related current approaches. It can be observed that our approach tries to address all of the above issues, while related approaches address a portion of them.

3. Periodic-Frequent Pattern-growth: Working and Performance Issues

In this section, we describe the working of PFP-growth along with its performance issues.

3.1. PFP-growth

The PFP-growth involves two steps: (*i*) Compress the database into a *tree* known as Periodic-Frequent tree (PF-tree) and (*ii*) Recursively mine PF-tree to discover all periodic-frequent patterns. Before explaining these two steps, we describe the structure of PF-tree as we will be using similar *tree* structure for our algorithm.

3.1.1. 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 Frequent Pattern-tree (FP-tree) [17]. 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 timestamp, called *ts*-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 $i_j[ts_1, ts_2, \cdots, ts_n]$, where $i_j \in I$ is the node's item name and $ts_i \in R$, is a timestamp in which i_j is the last item. The conceptual structure of a prefix-tree in PF-tree is shown in Figure 2. Like an FP-tree, each node in a PF-tree maintains parent, children, and node traversal pointers. Please note that no node in a PF-tree maintains the support count as in FP-tree. To facilitate a high degree of compactness, items in the prefix-tree are arranged in support-descending order.

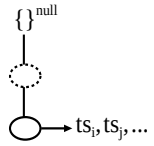


Figure 2: Conceptual structure of prefix-tree in PF-tree. Dotted ellipse represents ordinary node, while other ellipse represents tail-node of sorted transactions with timestamps $ts_i, ts_j \in R$.

3.1.2. Construction of PF-tree

The periodic-frequent patterns discovered using *minSup* and *maxPer* satisfy the anti-monotonic property (see Property 1). Therefore, finding periodic-frequent 1-patterns (or items) plays an important role in efficient discovery of periodic-frequent k-patterns, $k > 1$. Algorithm 1 describes the steps followed by PFP-growth to discover periodic-frequent items. Let *PI* denote the set of periodic-frequent items discovered by this algorithm.

Figure 3(a), (b) and (c) respectively show the PF-list generated after scanning the first, second and every transaction in the database (lines 2 to 8 in Algorithm 1). To reflect the correct *periodicity* for an item, the *periodicity* value of every item in PF-list is re-calculated by setting $ts_{cur} = |TDB|$ (lines 9 to 11 in Algorithm 1). Figure 3(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 *support* less than *minSup* or *periodicity* more than *maxPer* are considered as uninteresting items and removed from the PF-list (lines 12 to 16 in Algorithm 1). The remaining items are considered as periodic-frequent items and sorted in descending order of their f (or *support*) value (line

17 in Algorithm 1). Figure 3(e) shows the sorted list of periodic-frequent items.

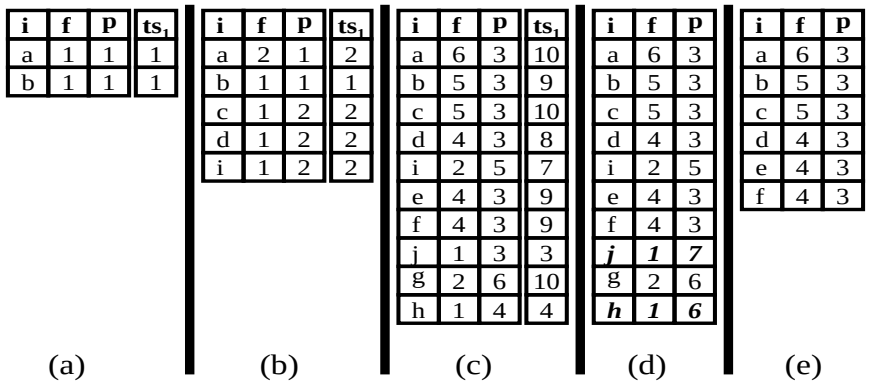


Figure 3: Construction of PF-list. (a) After scanning first transaction (b) After scanning second transaction (c) After scanning every transaction (d) Updated *periodicity* of items and (e) Sorted list of periodic-frequent items

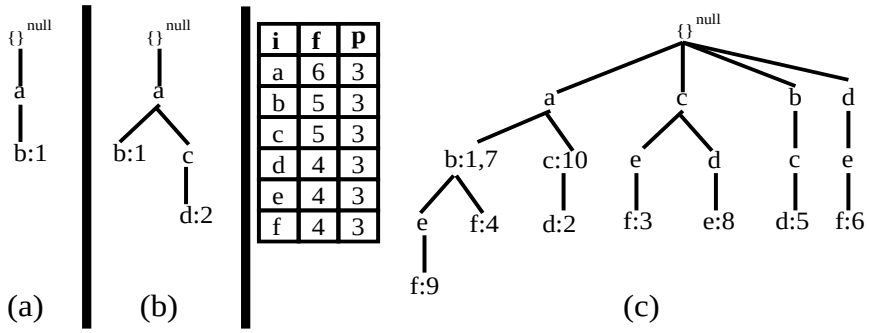


Figure 4: Construction of PF-tree. (a) After scanning first transaction (b) After scanning second transaction and (c) After scanning every transaction

Using the FP-tree construction technique [17], only the items in *PI* will take part in the construction of PF-tree. The *tree* construction starts by inserting the first transition, '1 : *ab*', according to PF-list order, as shown in Figure 4(a). The tail-node '*b* : 1' carries the timestamp of the transaction. After removing the uninteresting item '*i*' from the second transaction '2 : *acdi*', a branch as shown in Figure 4(b) is created with node '*d* : 2' as the tail-node. After inserting all the transactions in the database, we get the final PF-tree as shown in Figure 4(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.

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

1: Let ts_l be a temporary array that explicitly records the timestamps of last occurring transactions of all items in the PF-list. Let $t = (ts_{cur}, X)$ denote a transaction with ts_{cur} and X representing the timestamp of current transaction and pattern, respectively. Let P_{cur} be a temporary variable that records the current period of an item.
2: **for** each transaction $t \in TDB$ **do**
3: **if** ts_{cur} is i's first occurrence **then**
4: Set $f^i = 1$, $ts^i_l = ts_{cur}$ and $p^i = ts_{cur}$.
5: **else**
6: Calculate $p_{cur} = ts_{cur} - ts^i_l$. Set f^i++, $p^i = (p_{cur} > p^i)?p_{cur} : p^i$ and $ts^i_l = ts_{cur}$.
7: **end if**
8: **end for**
9: **for** each item i in PF-list **do**
10: Calculate $p_{cur} = |TDB| - ts^i_l$. Set $p^i = (p_{cur} > p^i)?p_{cur} : p^i$.
11: **end for**
12: **for** each item i in PF-list **do**
13: **if** $f^i < minSup$ or $p^i > maxPer$ **then**
14: Remove i from PF-list.
15: **end if**
16: **end for**
17: Consider the remaining items in PF-list as periodic-frequent items and sort them with respect to their f value.

3.1.3. Mining PF-tree

The PFP-growth finds periodic-frequent patterns by employing the following steps:

1. 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 5(a) shows the prefix-tree for the item 'f', say PT_f.
2. For each item 'j' in PT_i, aggregate all of its node's ts-list to derive the ts-list of 'ij', i.e., TS^{ij}. Next, perform a complete search on TS^{ij} to measure the *support* and *periodicity* of 'ij'. Now determine the interestingness of

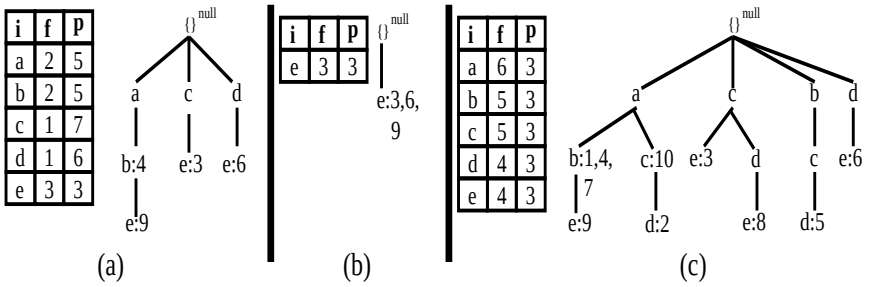


Figure 5: Mining periodic-frequent patterns for the suffix item 'f.' (a) Prefix-tree of f, i.e., PT_f (b) Conditional tree of 'f', i.e., CT_f and (c) PF-tree after removing item 'f'.

'ij' by comparing its *support* and *periodicity* against *minSup* and *maxPer*, respectively. If 'ij' is a periodic-frequent pattern, then we consider 'j' as a periodic-frequent item in PT_i.

EXAMPLE 4. Let us consider the last item 'e' in the PT_f. The set of timestamps containing 'e' in PT_f is $\{3,6,9\}$. Therefore, the ts-list of the pattern 'ef', i.e., $TS^{ef} = \{3,6,9\}$. A complete search on TS^{ef} gives $Sup(ef) = 3$ and $Per(ef) = 3$. Since $Sup(ef) \geq minSup$ 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 5(a) shows the *support* and *periodicity* of each item in PT_f.

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

   EXAMPLE 5. Figure 5(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 aperiodic-frequent items in PT_f.

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

   EXAMPLE 6. Figure 5(c) shows the PF-tree generated after pruning the item 'f' in Figure 4(c). It can be observed that ts-list of all the nodes containing 'f' in Figure 4(c) have been pushed to their corresponding parent nodes.

3.2. Performance Issues

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

1. The PFP-growth prunes uninteresting items from the PF-list only after the initial scan on a database. We have observed that this approach leads to increase in memory, search and update requirements of PF-list.

   EXAMPLE 7. In Table 1, the items 'g' and 'h' have initially appeared in the transaction whose timestamp is 4 (i.e., $ts = 4$). Thus, their first period is going to be 4, which is greater than the *maxPer*. In other words, these two items were aperiodic by the time when they were first identified in the database. Thus, there is no need to consider these two items in the construction of PF-list. However, PFP-growth considers these items in the construction of PF-list. This results in performance issues as described above.

2. PFP-growth determines the periodic interestingness of every item 'j' in PT_i by performing a complete search on its ts-list. In voluminous databases, the ts-list of a pattern (or for an item 'j' in PT_i) can be long. In such cases, the task of performing a complete search on a pattern's ts-list can make the periodic-frequent mining a computationally expensive process (or PFP-growth a computationally expensive algorithm).

   EXAMPLE 8. Let us consider the item 'a' in PT_f. The ts-list of 'a' in PT_f contains 4 and 9. Therefore, $TS^{af} = \{4, 9\}$. All *periods* of 'af' are: $p_1^{af} = 4$ $(= 4 - ts_i)$, $p_2^{af} = 5$ $(= 9 - 4)$ and $p_3^{af} = 1$ $(= ts_f - 9)$. The PFP-growth measures $periodicity = 5$ $(= max(4, 5, 1))$, compares it against *maxPer*, and then determines 'a' as an aperiodic-frequent item in PT_f. In other words, PFP-growth determines the interestingness of 'af' by performing a complete search on its ts-list. However, such a complete search is not necessary to determine 'af' as an aperiodic-frequent pattern. The reason is that the first period of 'af' itself fails to satisfy the *maxPer* as $p_1^{af} \not\leq maxPer$.

In the next section, we discuss our approach that addresses the performance issues of PFP-growth.

4. Proposed Algorithm

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

4.1. Basic Idea: local-periodicity

Our idea to reduce the computational cost of mining periodic-frequent patterns is as follows. "*Apply greedy search on a pattern's ts-list to derive its local-periodicity. For a pattern, local-periodicity and periodicity correspond to sub-optimal solution and global optimal solution, respectively. If the local-periodicity of a pattern fails to satisfy the maxPer, then we immediately determine it as an aperiodic-frequent pattern and avoid further search on the ts-list. Thus, reducing the computational cost of mining these patterns.*"

The *local-periodicity* of a pattern X is defined in Definition 1. Example 9 illustrates this definition. The correctness of our idea is based on Property 2 and shown in Lemma 1. Example 10 illustrates our idea of finding aperiodic-frequent patterns using *local-periodicity*.

Definition 1. ***(Local-periodicity of pattern** X.)* *Let $P^X = \{p^X_1, p^X_2, \cdots, p^X_n\}$, $n = Sup(X)+1$, denote the complete set of periods for X in TDB. Let $\widehat{P^X} = \{p^X_1, p^X_2, \cdots, p^X_k\}$, $1 \leq k \leq n$, be an ordered set of periods of X such that $\widehat{P^X} \subseteq P^X$. The local-periodicity of X, denoted as $loc\text{-}per(X)$, refers to the maximum period in $\widehat{P^X}$. That is, $loc\text{-}per(X) = max(p^X_1, p^X_2, \cdots, p^X_k)$.*

EXAMPLE 9. 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 $loc\text{-}per(ab) = max(p^{ab}_j | \forall p^{ab}_j \in \widehat{P^{ab}}) = max(1,3) = 3$.

Property 2. *For a pattern X, $Per(X) \geq loc\text{-}per(X)$ as $\widehat{P^X} \subseteq P^X$.*

Lemma 1. *For the frequent pattern X, if $loc\text{-}per(X) > maxPer$, then X is an aperiodic-frequent pattern.*

PROOF. From Property 2, it turns out that if $loc\text{-}per(X) > maxPer$, then $Per(X) > maxPer$. Thus, X is an aperiodic-frequent pattern. Hence proved.

EXAMPLE 10. In Table 1, the pattern 'af' occurs at the timestamps of 4 and 9. Therefore, $TS^{af} = \{4,9\}$. The first *period* of 'af,' i.e., $p^{af}_1 = 4$ $(= ts_i - 4)$. At this point, the $loc\text{-}per(af) = p^{af}_1 = 4$. As $loc\text{-}per(af) > maxPer$, we can conclude in prior that $Per(af) > maxPer$. Thus, 'af' is an uninteresting pattern.

Another important advantage of adopting greedy search to determine the periodic interestingness of a pattern is that both *local-periodicity* and *periodicity* remains the same for periodic-frequent patterns. Thus, we do not miss any information pertaining to periodic-frequent patterns. The correctness of this argument is based on Property 3, and shown in Lemma 2.

Property 3. *For a pattern X, loc-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. For a periodic-frequent pattern X, $\widehat{P^X} = P^X$. From Property 3, it turns out that $loc\text{-}per(X) = Per(X)$. Hence proved.

Overall, our approach facilitates an algorithm to find the periodic-frequent patterns with a global optimal solution, while pruning the aperiodic-frequent patterns with a sub-optimal solution. Thus, our approach reduces the computational cost of mining the patterns without missing any information pertaining to periodic-frequent patterns.

In the next subsection, we describe PFP-growth++ that implements the above approach to find periodic-frequent patterns effectively.

4.2. PFP-growth++

Our PFP-growth algorithm involves the following two steps:

1. Compress the database into a tree structure known as PF-tree++ and
2. Recursively mine PF-tree++ to discover the complete set of periodic-frequent patterns.

We now discuss each of these steps.

4.2.1. 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 (see Section 3.2.1).

The periodic-frequent items (or 1-patterns) play a key role in discovering periodic-frequent k-patterns, $k > 1$. In order to discover these items, we employ a pruning technique that is based on the concepts of 2-Phase Locking [14] (i.e., 'expanding phase' and 'shrinking phase'). The pruning technique is as follows:

- **Expanding phase:** In this phase, we insert every new item found in a transaction into the PF-list++. Thus, expanding the size of PF-list++. This phase starts from the first transaction (i.e., $ts = 1$) in the database and ends when the timestamp of a transaction equals to *maxPer*. If the *timestamp* of a transaction is greater than *maxPer*, then we do not insert any new item found in the corresponding transaction into the PF-list++. It is because such items are aperiodic as their first *period* (or *local-periodicity*) fails to satisfy *maxPer*.

- **Shrinking phase:** In this phase, we remove those items from the PF-list++ that are occurring aperiodically within a database. Thus, shrinking the length of PF-list++. The aperiodic items are those items that have *period* (or *local-periodicity*) greater than the *maxPer*. This phase starts right after the completion of expansion phase (i.e., when the *timestamp* of a transaction equal to $maxPer + 1$) and continues until the end of database.

The items remaining in the PF-list++ are called as **candidate items**. We will perform another scan on the candidate items to discover periodic-frequent items. The relationship between all items (I), candidate items (CI) and periodic-frequent items (PI) in TDB is $I \supseteq CI \supseteq PI$.

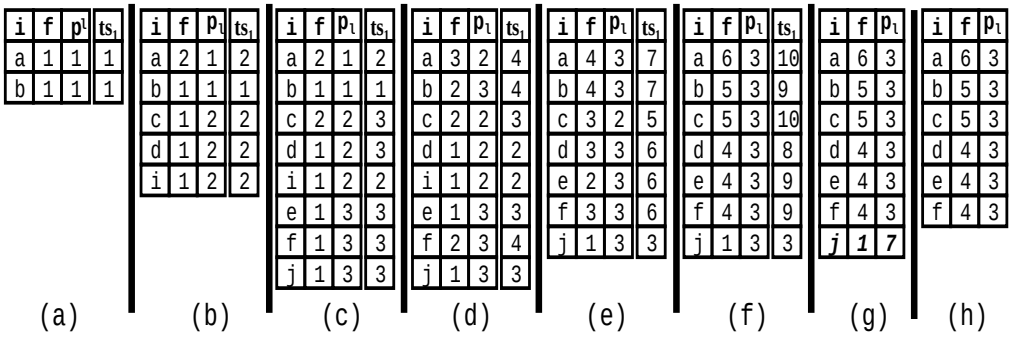


Figure 6: The construction of PF-list++. (a) After scanning the first transaction (b) After scanning the second transaction (c) After scanning the third transaction (d) After scanning the fourth transaction (e) After scanning the seventh transaction (f) After scanning all transactions (g) Measuring the actual *periodicity* for all candidate 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 this algorithm using the database shown in Table 1. The 'expanding phase' starts from first transaction. The scan on first transaction, "$1 : ab$," with $ts_{cur} = 1$ results in inserting the items 'a' and 'b' in to the PF-list++ with $f = 1$, $p_l = 1$ and $ts_l = 1$ (lines 4 to 6 in Algorithm 2). The resultant PF-list++ is shown in Figure 6(a). The scan on the second transaction, "$2 : acdi$," with $ts_{cur} = 2$

results in adding the items 'c', 'd' and 'i' into the PF-list++ with $f = 1$, $ts_l = 2$ and $p_l = 2$. Simultaneously, the f and ts_l values of an already existing item 'a' are updated to 2 and 2, respectively (lines 7 and 8 in Algorithm 2). The resultant PF-list++ was shown in Figure 6(b). Similarly, the scan on the third transaction with $ts_{cur} = 3$ results in adding the items 'e', 'f' and 'j' into the PF-list++ with $f = 1$, $ts_l = 3$ and $p_l = 3$. In addition, the f, p and ts_l values of 'c' are updated to 2, 2 and 3, respectively. Figure 6(c) shows the PF-list++ generated after scanning the third transaction. Since $maxPer = 3$, the expanding phase ends at $ts_{cur} = 3$. The 'shrinking phase' begins from $ts_{cur} = 4$. The scan on the fourth transaction, "$4 : abfgh$," updates the f, p_l and ts_l of the items 'a', 'b' and 'f' accordingly as shown in Figure 6(d). It can be observed that our algorithm **do not add** the new items 'g' and 'h' into the PF-list++ as their *period* (or *local-periodicity*) is greater than the *maxPer* (lines 11 to 18 in Algorithm 2). Similar process is repeated for the other transactions in the database, and the PF-list++ is constructed accordingly. Figure 6(e) shows the PF-list++ constructed after scanning the seventh transaction. It can be observed that aperiodic-frequent item 'i' has been pruned from the PF-list++ as its *local-periodicity* (or $p_l = 5$ $(= ts_{cur} - ts_l)$) has failed to satisfy the *maxPer* (lines 15 to 17 in Algorithm 2). Figure 6(f) shows the **initial PF-list++** constructed after scanning the entire database. The items in PF-list++ denote candidate items. Figure 6(g) shows the updated *periodicity* of all candidate items in the PF-list++ (lines 21 to 32 in Algorithm 2). It can be observed that p_l value of the candidate item 'j' has been updated from 3 to 7. Figure 6(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 (line 33 in Algorithm 2).

Algorithms 3 and 4 show the construction of prefix-tree in PFP-growth++. The working of these two algorithms has already been discussed in Section 3.2.2. Figure 7 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++.

4.2.2. Mining PF-tree++.

Algorithm 5 describes the procedure to mine periodic-frequent patterns from PF-tree++. Briefly, the working of this algorithm is as follows. 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

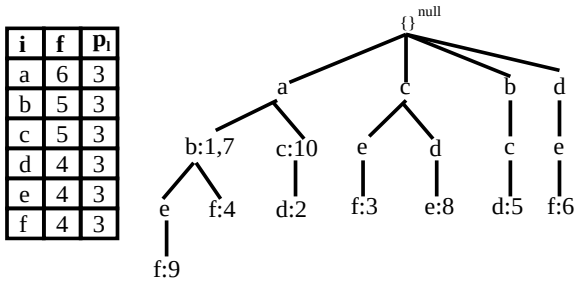


Figure 7: PF-tree++ generated after scanning every transaction in the database

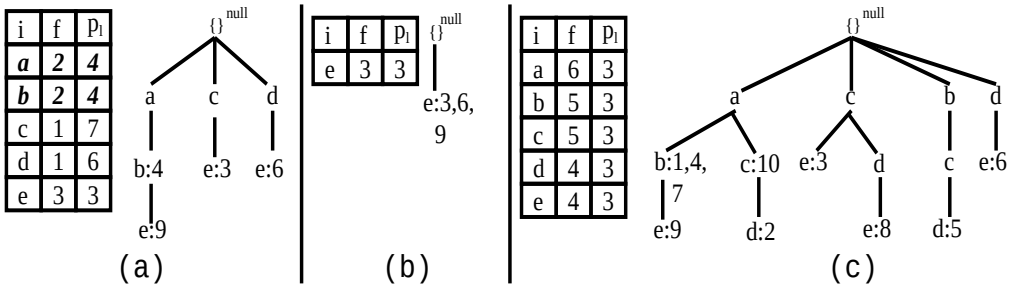


Figure 8: Mining periodic-frequent patterns using 'f' as suffix item. (a) Prefix-tree of suffix item 'f', i.e., PT_f (b) Conditional tree of suffix item 'f', i.e., CT_f and (c) PF-tree++ after pruning item 'f'

the PF-list++, each node labeled 'i' in the PF-tree++ must be a *tail*-node. While constructing the PT_i, we map the ts-list of every node of 'i' to all items in the respective path explicitly in a temporary array. It facilitates the construction of ts-list for each item 'j' in the PF-list++ of PT_i, i.e., TS^{ij} (line 2 in Algorithm 5). The length of TS^{ij} gives the *support* of 'ij'. Algorithm 6 is used to measure the *local-periodicity* of 'ij.' This algorithm applies the following pruning technique to determine the interestingness of j in PT_i. The pruning technique is as follows: "*If* $loc\text{-}per(X) > maxPer$*, then X is an aperiodic-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 aperiodic-frequent in PT_i, then its p_l denotes the $local\text{-}periodicity$ (or the *period*) that has failed to satisfy the *maxPer*. The correctness of this technique has already been shown in Lemma 2.

Figure 8(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 ts-list of the nodes containing 'a' in PT_f gives $TS^{af} = \{4, 9\}$. The *support* of 'af', $S(af) = 2(= |TS^{af}|)$. As $S(af) \geq 2$ $(= minSup)$, we determine 'af' as a frequent pattern.

Next, we pass TS^{af} as an array in Algorithm 6 to find whether 'af' is a periodic-frequent or an aperiodic-frequent pattern. The first period, $p_{cur} = 4$ $(= 4 - 0)$ (line 1 in Algorithm 6). As $p_{cur} > maxPer$, we determine 'af' as an aperiodic-frequent pattern and return $p_l = p_{cur} = 4$ (lines 2 to 4 in Algorithm 6). Thus, we prevent the complete search on the ts-list of an aperiodic-frequent pattern. Similar process is applied for the remaining items in the PT_f. The PF-list++ in Figure **??**(a) shows the *support* and *local-periodicity* of items in PT_f. It can be observed that the p_l value of aperiodic-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 aperiodic-frequent items from the PT_i. If the deleted node is a *tail*-node, its ts-list is pushed up to its parent node. Figure 8(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 ts-list to its respective parent nodes. Figure 8(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 correctness of PFP-growth++ is shown in Lemma 3.

Lemma 3. *Let* α *be a pattern in PF-tree++. Let minSup and maxPer be the user-defined minimum support and maximum periodicity, respectively. Let B be the* α*-conditional pattern base, and* β *be an item in B. Let* TS^{β} *be an array of timestamps containing* β *in B. If* α *is a periodic-frequent pattern,* $TS^{\beta}.length \geq minSup$ *and* $Periodicity(TS^{\beta}) \leq maxPer$*, then* $< \alpha\beta >$ *is also a periodic-frequent pattern.*

PROOF. According to the definition of conditional pattern base and compact PF-tree++, each subset in B occurs under the condition of the occurrence of α in the transactional database. If an item β in B at the timestamps $ts_i, ts_j, \cdots$, then β appears with α at $ts_i, ts_j, \cdots$. Thus, TS^{β} in B is same as $TS^{\alpha\beta}$. From the definition of periodic-frequent pattern in our model, if $TS^{\beta}.length \geq minSup$ and $Periodicity(TS^{\beta}) \leq maxPer$, then $< \alpha\beta >$ is also a periodic-frequent pattern. Hence proved.

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.

Table 4: Statistics of the databases. The terms T_{min}, T_{max} and T_{avg} respectively represent the minimum, maximum and average number of items within a transaction

Database	Type	T_{min}	T_{max}	T_{avg}	Size	Total Items
T10I4D100K	synthetic	1	29	10.1	100,000	870
T20I6D100K	synthetic	1	47	19.89	99,522	893
T10I4D1000K	synthetic	1	31	10.1	983,155	30,387
T25I6D1000K	synthetic	1	55	24.9	999960	1007
Shop-4	Real	1	82	2.36	59,240	155
Retail	Real	1	76	10.3	88,162	16,470
Kosarak	Real	1	2,498	8	990,002	41,270

5. Experimental Results

This section compares the performance of PFP-growth and PFP-growth++ algorithms on various databases. We show that our PFP-growth++ is runtime efficient and highly scalable than the PFP-growth. Both of these algorithms are written in GNU C++ and run with Ubuntu 14.4 on a 2.66 GHz machine with 8 GB of memory.

5.1. Experimental Setup

For our experiments, we have selected both synthetic ($T10I4D100K$, $T20I6D100K$ and $T10I4D1000K$) and real-world (*Retail*, *Shop*-14 and *Kosarak*) databases. The synthetic databases are generated by using the IBM data generator [1, 8], which is widely used for evaluating the association rule mining algorithms. The *Retail* database is constructed by Brijs et al. [5] from an anonymous Belgian retail supermarket store. The *Kosarak* database is provided by Ferenc Bodon and contains click-stream data of a hungarian on-line news portal. We have constructed the Shop-4 database from the clickstream data of product categories visited by the users in "Shop 4" (www.shop4.cz) provided in the ECML/PKDD 2005 Discovery challenge [10]. Each transaction in this database represents the set of web pages (or product categories) visited by the people at a particular *minute interval*. The statistical details of all of these databases has been shown in Table 4. The databases, *Retail* and *Kosarak*, are available for download at Frequent Itemset MIning repository [12].

5.2. Finding Periodic-Frequent Patterns

Figure 9 (*a*), (*b*), (*c*) and (*d*) show the number of periodic-frequent patterns discovered at various *minSup* values in $T10I4D100K$, $T20I6D100K$, *Retail* and

Shop-4 databases, respectively. The *X*-axis represents the maximum periodicity used to discover periodic-frequent patterns. The *Y*-axis represents the number of periodic-frequent patterns discovered at a given *maxPer* value. The following two observations can be drawn from these figures:

- At a fixed *maxPer*, the increase in *minSup* may decrease the number of periodic-frequent patterns being generated. This because many patterns fail to satisfy the increased *minSup* threshold values.
- At a fixed *minSup*, the increase in *maxPer* typically increases the number of periodic-frequent patterns. The reason is that increase in *maxPer* facilitates some of the frequent patterns that were having longer inter-arrival times to be periodic-frequent patterns.

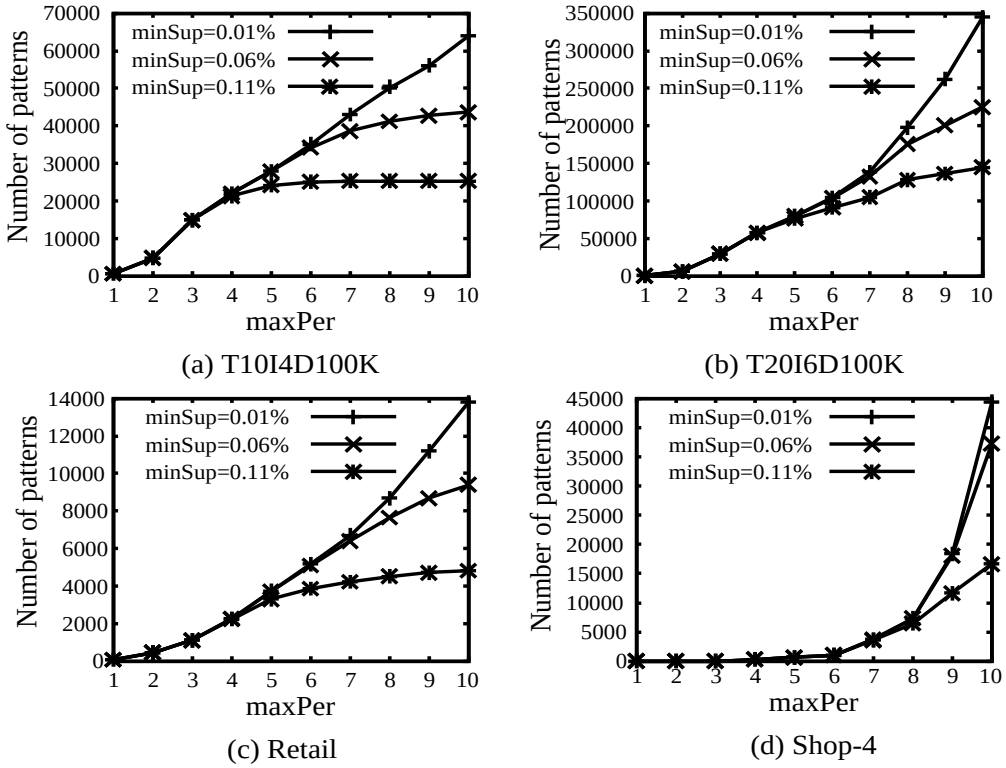


Figure 9: Number of periodic-frequent patterns discovered at different *minSup* and *maxPer* values

5.3. Runtime comparison of PFP-growth and PFP-growth++

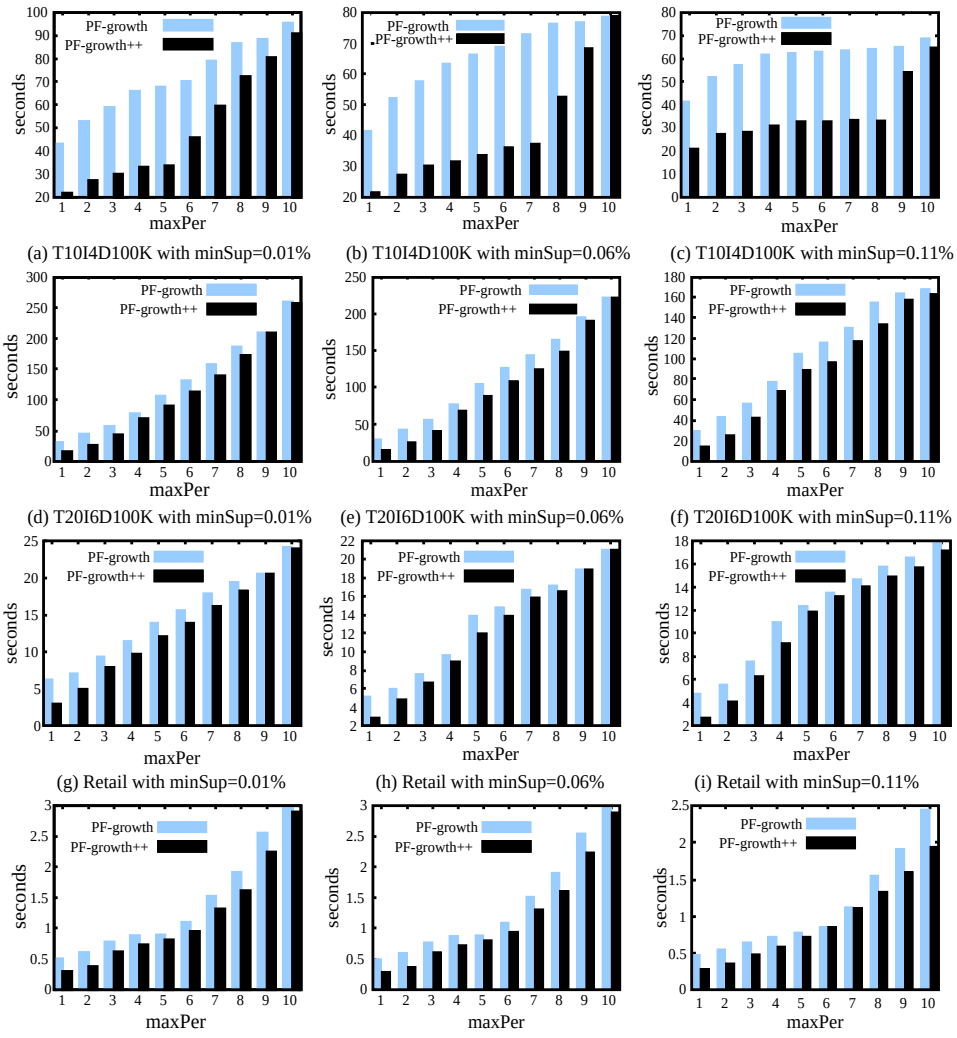
Figure 10 shows the runtime taken by PFP-growth and PFP-growth++ algorithms on various databases at different *minSup* and *maxPer* thresholds. The X-axis represents the maximum periodicity used to discover periodic-frequent patterns. The Y-axis represents the runtime taken by PFP-growth and PFP-growth++ algorithms to discover periodic-frequent patterns. The unit of runtime is seconds. The following three observations can be drawn from these figures:

- Increase in *maxPer* threshold has increased the runtime for both the algorithms. This is because of the increase in number of periodic-frequent pattern with the increase in *maxPer* threshold.
- 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.
- At a low *maxPer* value, the PFP-growth++ algorithm has outperformed the PFP-growth by a very large margin. It is because the PFP-growth++ has performed only partial search on the ts-lists of aperiodic-frequent patterns.

5.4. 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 database were widely used to study the scalability of algorithms. The experimental setup was as follows. Each database 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 11(a) (b) and (c) shows the runtime requirements of PFP-growth and PFP-growth++ algorithms on $T10I4D1000K$, $T25I6D1000K$ and *Kosarak* database, respectively. It can be observed that the increase in database size has increased the runtime of both the algorithms. However, the proposed PFP-growth++ has taken relatively less runtime than the PFP-growth. In particular, as the database size increases, the proposed PFP-growth++ algorithm has outperformed PFP-growth by an order of magnitude. The reason is as follows: *Increase in database size generally increases the size of ts-list of a pattern. The complete search on these huge ts-lists for both periodic and aperiodically occurring patterns by PFP-growth increases its runtime requirements. However, partial search on the ts-list of an*



(a) T10I4D100K with minSup=0.01% (b) T10I4D100K with minSup=0.06% (c) T10I4D100K with minSup=0.11%

(d) T20I6D100K with minSup=0.01% (e) T20I6D100K with minSup=0.06% (f) T20I6D100K with minSup=0.11%

(g) Retail with minSup=0.01% (h) Retail with minSup=0.06% (i) Retail with minSup=0.11%

Figure 10: Runtime comparison of PFP-growth and PFP-growth++ algorithms

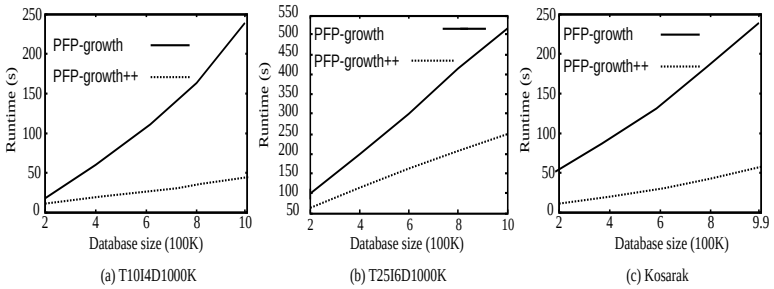


Figure 11: Scalability of PFP-growth and PFP-growth++ algorithms.

aperiodically occurring pattern has facilitated by our PFP-growth++ to reduce its runtime requirements. complete search on the tid-list of both periodic-frequent and aperiodic-frequent patterns by PFP-growth has increased its runtime requirements. The partial search on the ts-list of an aperiodically occurring pattern has facilitated the PFP-growth++ to reduce its runtime requirements.

5.5. A case study on Shop-4 database

In this subsection, we demonstrate the usefulness of periodic-frequent patterns by considering shop-4 database. Table 5 presents some of the patterns discovered in this database at $minSup = 0.1\%$ and $maxPrd = 10\%$. The first two patterns, i.e., {{Built-in dish washers}, {Refrigerators, freezers, show cases}} and {{Washer dryers}, {Refrigerators, freezers, show cases}, {built-in ovens, hobs, grills}}, are appearing not only frequently within the database but also appearing at regular intervals within the database. The remaining two patterns, i.e., {{Speakers for home cinemas}, {Home cinema systems-components}} and {{Tv's}, {Analog camcorders}}, are not appearing as frequently as the former two patterns; however, they are still appearing periodically within the database. This clearly shows that periodic-frequent pattern model can discover knowledge pertaining to both frequent and rare items.

6. Conclusions and Future Work

Determining the *periodicity* of every candidate pattern in an itemset lattice makes the periodic-frequent pattern mining a computationally expensive process. In this paper, we have proposed a novel approach to reduce the computational cost of mining the periodic-frequent patterns. Our approach facilitates the user to find the periodic-frequent patterns by pruning the aperiodically occurring patterns

Table 5: Some of the interesting periodic-frequent patterns discovered in Shop-4 database

S. No.	Pattern	*Support*	*Periodicity*
1	{{Built-in dish washers}, {Refrigerators, freezers, show cases}}	35.4%	0.77%
2	{{Washer dryers}, {Refrigerators, freezers, show cases}, {built-in ovens, hobs, grills}}	15.2%	1.34%
3	{{Speakers for home cinemas}, {Home cinema systems-components}}	0.11%	8.4%
4	{{Tv's}, {Analog camcorders}}	0.1%	9.5%

based on sub-optimal solutions. As a result, our approach reduces the computational cost 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 our PFP-growth++ is runtime efficient and scalable as well.

As a part of future work, we would like to extend our pruning techniques 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.

7. References

[1] Agrawal, R., Imieliński, T., Swami, A., 1993. Mining association rules between sets of items in large databases. In: SIGMOD. pp. 207–216.

[2] Amphawan, K., Lenca, P., Surarerks, A., 2009. Mining top-k periodic-frequent pattern from transactional databases without support threshold. In: Advances in Information Technology. pp. 18–29.

[3] Aref, W. G., Elfeky, M. G., Elmagarmid, A. K., mar 2004. Incremental, online, and merge mining of partial periodic patterns in time-series databases. IEEE TKDE 16 (3), 332–342.

[4] Berberidis, C., Vlahavas, I., Aref, W., Atallah, M., Elmagarmid, A., 2002. On the discovery of weak periodicities in large time series. In: PKDD. pp. 51–61.

[5] Brijs, T., Goethals, B., Swinnen, G., Vanhoof, K., Wets, G., 2000. A data mining framework for optimal product selection in retail supermarket data: the generalized profset model. In: KDD. pp. 300–304.

[6] Brin, S., Motwani, R., Silverstein, C., 1997. Beyond market baskets: generalizing association rules to correlations. In: SIGMOD. pp. 265–276.

[7] Cao, H., Cheung, D., Mamoulis, N., 2004. Discovering partial periodic patterns in discrete data sequences. In: Advances in Knowledge Discovery and Data Mining. Vol. 3056. pp. 653–658.

[8] Cheng, H., 1994. IBM Quest Market-Basket Synthetic Data Generator. Available at http://www.philippe-fournier-viger.com/spmf/datasets/IBM_Quest_data_generator.zip .

[9] Dong, G., Li, J., 2009. Emerging patterns. In: Encyclopedia of Database Systems. pp. 985–988.

[10] ECML PKDD Challenge, 2005. ClickStream data. Available at http://lisp.vse.cz/challenge/CURRENT/ .

[11] Esling, P., Agon, C., Dec. 2012. Time-series data mining. ACM Computing Surveys 45 (1), 12:1–12:34.

[12] Goethals, B., 2005. Frequent Itemset Mining repository. Available at http://fimi.ua.ac.be/data/ .

[13] Gouda, K., Zaki, M. J., 2001. Efficiently mining maximal frequent itemsets. In: ICDM. pp. 163–170.

[14] Gray, J., 1978. Notes on data base operating systems. In: Operating Systems, An Advanced Course. Vol. 60 of Lecture Notes in Computer Science. pp. 393–481.

[15] Han, J., Dong, G., Yin, Y., 1999. Efficient mining of partial periodic patterns in time series database. In: ICDE. pp. 106–115.

[16] Han, J., Gong, W., Yin, Y., 1998. Mining segment-wise periodic patterns in time-related databases. In: KDD. pp. 214–218.

[17] Han, J., Pei, J., Yin, Y., Mao, R., Jan. 2004. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Min. Knowl. Discov. 8 (1), 53–87.

[18] Han, J., Wang, J., Lu, Y., Tzvetkov, P., 2002. Mining top.k frequent closed patterns without minimum support. In: ICDM. pp. 211–218.

[19] Kiran, R. U., Kitsuregawa, M., 2014. Novel techniques to reduce search space in periodic-frequent pattern mining. In: DASFAA (2). pp. 377–391.

[20] Kiran, R. U., Reddy, P. K., 2010. Towards efficient mining of periodic-frequent patterns in transactional databases. In: DEXA (2). pp. 194–208.

[21] Kiran, R. U., Reddy, P. K., 2011. An alternative interestingness measure for mining periodic-frequent patterns. In: DASFAA (1). pp. 183–192.

[22] Kiran, R. U., Shang, H., Toyoda, M., Kitsuregawa, M., 2015. Discovering recurring patterns in time series. In: EDBT. pp. 97–108.

[23] Laxman, S., Sastry, P. S., Unnikrishnan, K. P., 2007. A fast algorithm for finding frequent episodes in event streams. In: KDD. pp. 410–419.

[24] Lent, B., Swami, A., Widom, J., 1997. Clustering association rules. In: ICDE. pp. 220–231.

[25] Ma, S., Hellerstein, J., 2001. Mining partially periodic event patterns with unknown periods. In: ICDE. pp. 205–214.

[26] Mannila, H., 1997. Methods and problems in data mining. In: The International Conference on Database Theory. pp. 41–55.

[27] Mohammad, Y. F. O., Nishida, T., 2013. Approximately recurring motif discovery using shift density estimation. In: IEA/AIE. pp. 141–150.

[28] Mooney, C. H., Roddick, J. F., mar 2013. Sequential pattern mining – approaches and algorithms. ACM Comput. Surv. 45 (2), 19:1–19:39.

[29] Oates, T., 2002. Peruse: An unsupervised algorithm for finding recurrig patterns in time series. In: ICDM. pp. 330–337.

[30] Omiecinski, E. R., January 2003. Alternative interest measures for mining associations in databases. IEEE Trans. on Knowl. and Data Eng. 15, 57–69.

[31] Özden, B., Ramaswamy, S., Silberschatz, A., 1998. Cyclic association rules. In: ICDE. pp. 412–421.

[32] Pei, J., Han, J., Lakshmanan, L. V., 2004. Pushing convertible constraints in frequent itemset mining. Data Mining and Knowledge Discovery 8, 227–252.

[33] Rashid, M. M., Karim, M. R., Jeong, B. S., Choi, H. J., 2012. Efficient mining regularly frequent patterns in transactional databases. In: DASFAA (1). pp. 258–271.

[34] Srivastava, S., Kiran, R. U., Reddy, P. K., 2011. Discovering diverse-frequent patterns in transactional databases. In: COMAD. pp. 69–78.

[35] Stormer, H., 2007. Improving e-commerce recommender systems by the identification of seasonal products. In: Twenty second Conference on Artificial Intelligence. pp. 92–99.

[36] Surana, A., Kiran, R. U., Reddy, P. K., 2011. An efficient approach to mine periodic-frequent patterns in transactional databases. In: PAKDD Workshops. pp. 254–266.

[37] Swamy, M. K., Reddy, P. K., Srivastava, S., 2014. Extracting diverse patterns with unbalanced concept hierarchy. In: PAKDD (I). pp. 15–27.

[38] Tanbeer, S. K., Ahmed, C. F., Jeong, B. S., Lee, Y. K., 2009. Discovering periodic-frequent patterns in transactional databases. In: PAKDD. pp. 242–253.

[39] Weiss, G. M., 2004. Mining with rarity: A unifying framework. SIGKDD Explorations 6 (1), 7–19.

[40] Yang, R., Wang, W., Yu, P., 2002. Infominer+: mining partial periodic patterns with gap penalties. In: ICDM. pp. 725–728.

[41] Yao, H., Hamilton, H. J., Butz, C. J., 2004. A foundational approach to mining itemset utilities from databases. In: SDM. SIAM, pp. 482–486.

[42] Zhang, M., Kao, B., Cheung, D. W., Yip, K. Y., aug 2007. Mining periodic patterns with gap requirement from sequences. ACM Trans. Knowl. Discov. Data 1 (2).

Algorithm 2 PF-list++

1: **for** each transaction $t \in TDB$ **do**
2: **if** $ts_{cur} < \beta$ **then**
3: /*Expanding phase*/
4: **if** ts_{cur} is i's first occurrence **then**
5: Insert i into the list and set $f^i = 1$, $ts^i_l = ts_{cur}$ and $p^i_l = ts_{cur}$.
6: **else**
7: Calculate $p_{cur} = ts_{cur} - ts^i_l$.
8: Set $f^+ +$, $ts^i_l = ts_{cur}$ and $p^i_l = (p_{cur} > p^i_l)?p_{cur} : p^i_l$.
9: **end if**
10: **else**
11: /*Shrinking phase*/
12: Calculate $p_{cur} = ts_{cur} - ts^i_l$.
13: **if** $p_{cur} < maxPer\beta$ **then**
14: Set $f^i + +$, $ts^i_l = ts_{cur}$ and $p^i_l = (p_{cur} > p^i_l)?p_{cur} : p^i_l$.
15: **else**
16: prune the i from the list;
17: **end if**
18: **end if**
19: **end for**
20: Let CI denote the set of candidate items remaining in PF-list++.
21: **for** each item i in PF-list++ **do**
22: **if** $f^i \geq minSup$ **then**
23: Calculate $p_{cur} = |TDB| - ts^i_l$.
24: **if** $p_{cur} \leq maxPer$ **then**
25: Set $p^i = (p_{cur} > p^i)?p_{cur} : p^i$.
26: **else**
27: Remove i from the PF-list++.
28: **end if**
29: **else**
30: Remove i from the PF-list++
31: **end if**
32: **end for**
33: Consider the remaining items in PF-list++ as periodic-frequent items and sort them with respect to their *support*. Let *PI* denote this sorted list of items in PF-list++.

Algorithm 3 PF-Tree++(TDB, PF-list++)

1: Create the root of an PF-tree++, T, and label it "*null*".
2: **for** each transaction $t \in TDB$ **do**
3: Set the timestamp of the corresponding transaction as ts_{cur}.
4: Select and sort the candidate items in t according to the order of PI. Let the sorted candidate item list in t be $[p|P]$, where p is the first item and P is the remaining list.
5: Call $insert\_tree([p|P], ts_{cur}, T)$.
6: **end for**
7: call PFP-growth++ ($Tree$, *null*);

Algorithm 4 insert_tree($[p|P]$, ts_{cur}, T)

1: **while** P is non-empty **do**
2: **if** T has a child N such that $p.itemName \neq N.itemName$ **then**
3: Create a new node N. Let its parent node be linked to T. Let its node-link be linked to nodes with the same itemName via the node-link structure. Remove p from P.
4: **end if**
5: **end while**
6: Add ts_{cur} to the leaf node.

Algorithm 5 PFP-growth++($Tree$, α)

1: **for** each a_i in the header of Tree **do**
2: Generate pattern $\beta = a_i \cup \alpha$. Collect all of the a_i's ts-lists into a temporary array, TS^{β}.
3: **if** $TS^{\beta}.length \geq minSup$ and $CalculateLocalPeriodicity(TS^{\beta}) < maxPer$ **then**
4: Construct β's conditional pattern base then β's conditional PF-tree++ $Tree_{\beta}$.
5: **if** $Tree_{\beta} \neq \emptyset$ **then**
6: call PFP-growth++($Tree_{\beta}$, β);
7: **end if**
8: **end if**
9: Remove a_i from the $Tree$ and push the a_i's ts-list to its parent nodes.
10: **end for**

Algorithm 6 CalculateLocalPeriodicity (TS^X: an array of timestamps containing X.)

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