Memory Efficient Mining of Periodic-Frequent Patterns in Transactional Databases

A. Anirudh*, R. Uday Kiran†, P. Krishna Reddy*, Masaru Kitsuregawa†
*Kohli Centre on Intelligent Systems (KCIS)
International Institute of Information Technology, Hyderabad, India
alampally.anirudh@research.iiit.ac.in, pkreddy@iiit.ac.in
†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 which exists in a transactional database. A frequent pattern is called periodic-frequent if it appears at regular intervals in a transactional database. In the literature, a model of periodic-frequent patterns was proposed and pattern growth like approaches to extract patterns are being explored. In these approaches, a periodic-frequent pattern tree is built in which a transaction-id list is maintained at each path's tail-node. As the typical size of transactional database is very huge in the modern e-commerce era, extraction of periodic-frequent patterns by maintaining transaction-ids in the tree requires more memory. In this paper, to reduce the memory requirements, we introduced a notion of *period summary* by capturing the periodicity of the patterns in a sequence of transaction-ids. While building the tree, the *period summary* of the transactions is computed and stored at the tail-node of the tree instead of the transaction-ids. We have also proposed a merging framework for *period summaries* for mining periodic-frequent patterns. The performance could be improved significantly as the memory required to store the *period summaries* is significantly less than the memory required to store the transaction-id list. Experimental results show that the proposed approach reduces the memory consumption significantly and also improves the runtime efficiency considerably over the existing approaches.**

I. INTRODUCTION

Periodic-frequent pattern mining [1] is an important knowledge discovery technique in data mining. Its purpose is to discover all patterns that are not only frequent, but also periodic in a transactional database. Periodic-frequent pattern mining has many applications such as detection of unusual activity [2], finding co-occurring genes in biological datasets [3], improving the performance of recommender systems [4] and event detection in social networks [5]. A classic application to demonstrate the usefulness of these patterns is market-basket analysis. It analyzes how regularly a set of items are being purchased by the customers. An example of a periodic-frequent pattern is as follows:

$\{Bread, Battery\}$ $[support = 10\%, periodicity = 1\ hr.]$

The above pattern demonstrates that the items 'Bread' and 'Battery' have been purchased by 10% of the customers, and the maximum time interval between any two consecutive purchases containing both of these items is no more than an hour. This predictive behavior of the customers' purchases can facilitate the users in proper inventory management.

The basic model of periodic-frequent patterns is as follows [1]: Let $I = \{i_1, i_2, \cdots, i_n\}$, $1 \leq n$, be a 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 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 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 a pattern X is the number of transactions containing X in TDB, which is denoted as $Sup(X)$. Therefore, $Sup(X) = |TID^X|$. Let tid_i^X and tid_{i+1}^X, $i \in [1, m-1]$ be two consecutive transaction-ids where X has appeared in TDB. The **period** of a pattern X is the number of transactions or the time difference between tid_{i+1}^X and tid_i^X. Let $P^X = \{p_1^X, p_2^X, \cdots, p_r^X\}$, $r = Sup(X) + 1$, be the complete set of periods of X in TDB. The **periodicity** of a pattern X is the maximum difference between any two adjacent occurrences of X, denoted as $Per(X) = \max(p_1^X, p_2^X, \cdots, p_r^X)$. A pattern X is a periodic-frequent pattern if $Sup(X) \geq minSup$ and $Per(X) \leq maxPer$, where $minSup$ and $maxPer$ represent the user-defined thresholds on $support$ and $periodicity$ respectively. Both $support$ and $periodicity$ of a pattern can be described in percentage of $|TDB|$.

Example 1: Consider the transactional database shown in Table I. Each transaction in this database is uniquely identifiable with a transactional-id (tid), which is also a timestamp of that transaction. Here, $I = \{a, b, c, d, e, f, g\}$. The set of items containing 'a', 'c' and 'd', i.e., 'acd' is a pattern. This pattern contains 3 items. Therefore, it is a 3-pattern. In the database, this pattern occurs in the $tids$ of 1, 3, 4, 6, 7, 8, 9, 11, 13, 15, 17 and 19. Therefore, $TID^{acd} = \{1, 3, 4, 6, 7, 8, 9, 11, 13, 15, 17, 19\}$. The $support$ of this pattern, i.e., $Sup(acd) = |TID^{acd}| = 12$. The periods for this pattern are $1(= 1 - tid_i)$, $2(= 3 - 1)$, $1(= 4 - 3)$, $2(= 6 - 4)$, $1(= 7 - 6)$, $1(= 8 - 7)$, $1(= 9 - 8)$, $2(= 11 - 9)$, $2(= 13 - 11)$, $2(= 15 - 13)$, $2(= 17 - 15)$ and $1(= tid_l - 19)$, where $tid_i = 0$ represents the initial transaction and $tid_l = 20$ represents the last transaction in the transactional database. The $periodicity$

Table I: A running example of a Transactional Database

TID	Items	TID	Items	TID	Items	TID	Items
1	a, c, d, g	6	a, b, c, d	11	a, c, d, e	16	b, e, f, g
2	c, e, f	7	a, c, d, f	12	b, e, g	17	a, b, c, d, e
3	a, c, d	8	a, b, c, d	13	a, c, d, g	18	b, e, g
4	a, b, c, d, e	9	a, c, d, e	14	b, e, f	19	a, c, d, e, g
5	b, f	10	b, e, f, g	15	a, c, d	20	b, g

of acd, $Per(acd) = maximum(1, 2, 1, 2, 1, 1, 1, 2, 2, 2, 1) = 2$. If the user-specified $minSup = 10$ and $maxPer = 4$, the pattern acd is a periodic-frequent pattern because $Sup(acd) \geq minSup$ and $Per(acd) \leq maxPer$.

Tanbeer et al. [1] have described a pattern-growth algorithm, called Periodic-Frequent Pattern-growth (PFP-growth), to find the complete set of periodic-frequent patterns. The algorithm contains two steps: (i) Compress the transactional database into a Periodic-Frequent Pattern Tree (PF-Tree), and (ii) discover the complete set of periodic-frequent patterns by recursively mining the PF-tree.

It can be noted that, for extracting periodic-frequent patterns, the transaction-ids of all transactions have to be stored in the tail-nodes of the PF-tree. As the typical size of transactional database is very huge in this era, extraction of periodic-frequent patterns from voluminous databases, such as Twitter logs and Facebook logs, require ample main memory for building and mining PF-tree.

In this paper, we have proposed an efficient approach to extract periodic-frequent patterns based on the notion of *period summary*. It is based on the observation that it is also possible to extract periodic-frequent patterns by maintaining the *period summary* of the list of transaction-ids. The *period summary* captures the periodicity of a list of transactions and can be stored in a compact manner. In the proposed approach, instead of transaction-ids, we store the *period summary* of the list of transactions in the tree called Period-Summary tree (PS-tree). We have proposed a pattern-growth algorithm on PS-tree, called PS-growth, in which we determine if a pattern is periodic or not by merging *period summary* information of the tail-nodes.

With the proposed approach, it is possible to achieve the improved performance as the memory required to store *period summaries* is significantly less than the memory required to store transaction-ids. As a result, the proposed approach can be extended to mine the periodic-frequent patterns from the data sets of increased size. Experimental results of three types of data sets show that the proposed approach is memory efficient significantly and run time efficient considerably as compared to the existing approaches.

The rest of the paper is organized as follows. In Section 2, we discuss the existing approaches in the field of periodic pattern mining. Section 3 discusses the overview of existing approaches (PFP-growth). A pattern-growth approach based on a tree structure, called Period Summary tree (PS-tree) is discussed in Section 4. We report our experimental results in Section 5. Finally, Section 6 concludes the paper.

II. Related Work

Periodic-frequent patterns were first introduced by Tanbeer et al. [1]. Variations in periodic-frequent patterns has been widely studied in [6]–[9]. These variations are regarding types of innovations in pruning methods. However, in this paper, we have proposed an innovation to improve the memory efficiency. So, the contribution in this paper is orthogonal to above mentioned approaches and the proposed approach can be employed to improve the memory efficiency of these approaches.

To improve the runtime of mining periodic-frequent patterns, Kiran and Kitsuregawa [10] have suggested a *greedy-search technique* to determine the periodic interestingness of a pattern. Amphawan et al. [11] introduced approximate periodicity to reduce the the memory requirements of mining periodic-frequent patterns. In Amphawan's model the transactional timeline is divided into intervals of size *maxPer* and interval information is stored only when a pattern is occurring in that interval. The proposed model is different from this as the size of interval is not restricted to *maxPer* and can expand as long as the transaction-ids merge.

III. Overview of Existing Approaches

The existing PFP-growth algorithm [1] accepts a transactional database, $minSup$ and $maxPer$ as inputs and outputs a complete set of periodic-frequent patterns. It involves the following two steps: (i) Construction of PF-tree and (ii) Mining of PF-tree. Before we discuss these two steps, we describe the structure of PF-tree. As an example, we consider entries in Table I as our transactional database. Let us assume that $minSup = 10$ and $maxPer = 4$.

A. Structure of PF-tree

The structure of PF-tree resembles that of Frequent Pattern-tree (FP-tree) [12]. It contains a PF-list and a prefix-tree. A PF-list consists of three fields - $itemname$ (Item), $support$ (sup) and $periodicity$ (per). The items in PF-tree are sorted in the descending order of $support$ to facilitate high compactness. Two types of nodes are maintained in PF-tree: ordinary node and tail-node. The former is the type of node similar to that used in FP-tree, whereas the latter node explicitly maintains the transaction-ids ($tids$) for each occurrence of that pattern only at the tail-node of every transaction.

B. Construction of PF-tree

The periodic-frequent patterns satisfy the downward closure property [1]. Henceforth, periodic-frequent items (or 1-patterns) play a key role in efficient discovery of periodic-frequent patterns. These items are discovered by scanning the

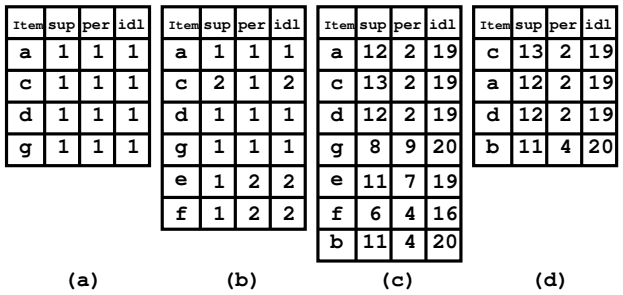

Figure 1: Construction of PF-list/PS-list. (a) After scanning the first transaction (b) After scanning the second transaction (c) After scanning all transactions (d) Final sorted list of the periodic-frequent items of size 1.

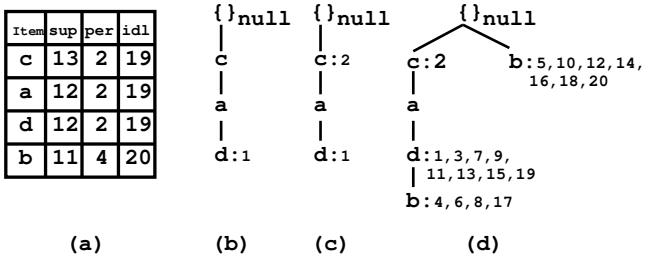

Figure 2: Construction of PF-tree. (a) PF-list (b) After scanning the first transaction (c) After scanning the second transaction (d) After scanning all transactions.

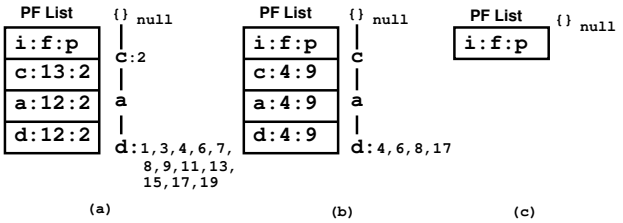

Figure 3: Mining using PFP-growth algorithm. (a) PF-tree after removing 'b' (b) Prefix tree of 'b' (c) Conditional Tree of 'b'.

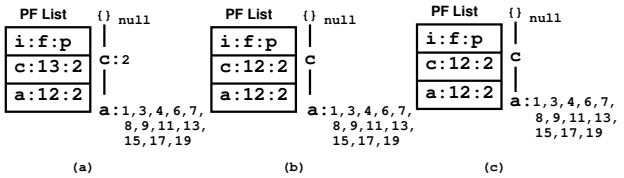

Figure 4: Mining using PFP-growth algorithm. (a) PF-tree after removing 'd' (b) Prefix tree of 'd' (c) Conditional Tree of 'd'.

database and populating the PF-list as per the steps given in Algorithm 1. Figure 1(a), 1(b) and 1(c) show the PF-list generated after scanning the first, second and entire database, respectively. Figure 1(d) shows the final PF-list generated after pruning all items that have failed to satisfy the *minSup* and *maxPer* constraints.

Algorithm 1 PF-list (TDB, $maxPer$, $minSup$)

1: **for** each transaction $t_{\text{cur}} \in TDB$ **do**
2: **for** each item i in t_{cur} **do**
3: **if** t_{cur} is i's first occurrence **then**
4: Set sup^i = 1, $p^i = t^i_{cur}$ and $id^i_l = t^i_{cur}$
5: **else**
6: Set sup^i += 1, $p^i_{cur} = t^i_{cur}$ - id^i_l and $id^i_l = t^i_{cur}$
7: **if** $p^i_{cur} > p$ **then**
8: $p = p^i_{cur}$

In the second database scan, the items in the PF-list will take part in the construction of PF-tree. The tree construction starts by inserting the first transaction, *(1, acdg)*, according to PF-list order, as shown in Figure 2(b). The elements which are not present in the PF-list are not considered while inserting into the prefix tree. The tail-node 'd' carries the transaction-id of the first transaction, 'd : [1]'. The second transaction, *(2, cef)*, is inserted into the tree with node 'c : [2]' as the tail-node (Figure 2). The process is repeated for the remaining transactions in the database. The final PF-tree generated after scanning the entire database is shown in Figure 2(d).

C. Mining of PF-tree

PFP-growth employs the following steps to discover periodic-frequent patterns from PF-tree:

- Choosing the last item i in the PF-list (Figure 2(a)) as an initial suffix item, its prefix-tree (denoted as PT_i) is constructed. This constitutes the prefix sub-paths of the nodes labeled i.
- For each item j in PT_i, we aggregate all of its nodes' transaction-id list to derive the transaction-id list of the pattern ij, i.e., TID_{ij}. Then, 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 to be periodic-frequent in PT_i.
- Choosing every periodic-frequent item j in PT_i, we construct its conditional tree, CT_{ij}, and mine it recursively to discover the patterns.
- After finding all periodic-frequent patterns for a suffix item i, we prune it from the original PF-tree and push the corresponding nodes' transaction-id lists to their parent nodes. We repeat the above steps until the PF-list becomes NULL. Figure 3 and Figure 4 show the mining process for item 'b' and item 'd'.

IV. Proposed Approach

A. Issues with the Existing Approaches

The space complexity for constructing a PF-tree is $O(n + |TDB|)$, where n represents the total number of nodes gen-

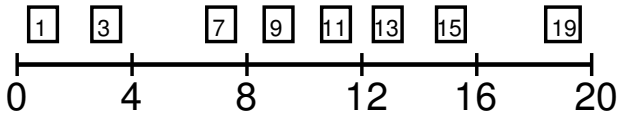

Figure 5: Occurrence timeline for pattern 'cad'

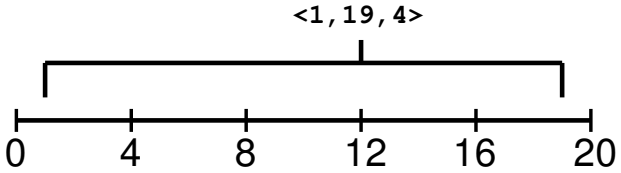

Figure 6: Occurrence timeline for pattern 'cad' using the notion of *period summary*

erated in PF-tree and $|TDB|$ represents the total number of transactions in TDB. The databases of many real-world applications (e.g. eCommerce and Twitter) contain millions of transactions. In other words, $|TDB|$ normally could be a very large number and the size of PF-tree depends on the number of transactions. So, the application of existing algorithm to mine the knowledge of periodic-frequent patterns is constrained due to the size of the transactional database. As we construct a conditional tree for every pattern we extract, the amount of main memory consumed is directly proportional to number of patterns. So, investigation of efficient approaches is a research issue.

B. Basic idea

During the construction of PF-Tree, the entire tid-list of a pattern gets fragmented into multiple distinct sub-lists at different tail-nodes of the PF-Tree.

Example 2: In Table I, the tid-list of pattern 'b' is, $TID^b = \{4, 5, 6, 8, 10, 12, 14, 16, 17, 18, 20\}$. The tid list of '$b$' gets divided into two branches 'b' and '$cadb$' (Figure 2(d)), and the corresponding sub-lists at the tail-node are {4, 6, 8, 17} and {5, 10, 12, 14, 16, 18, 20}.

It can be observed that instead of storing transaction-ids, it is sufficient to store the interval in which the transactional sequence is periodic. Let us say a pattern X is periodic in the range [a,b] such that the difference between any two of its occurrences in that range is no more than $maxPer$. Then, it is enough to store the interval extremes [a,b] denoting that the pattern is periodic in that range.

Based on this idea, we have proposed two concepts. One is the concept of *period summary* which is employed to build the tree and the other is the process of merging *period summaries* for the extraction of periodic-frequent patterns.

The concept of *period summary* is defined as follows.

Definition 1: A ***period summary*** of a pattern X, ps_i^X, captures the interval information in which a pattern has appeared periodically in the data and the $periodicity$ of respective pattern within that interval. That is, $ps_i^X = \langle tid_j^X, tid_k^X, per_i^X \rangle$, where tid_j^X and tid_k^X, $1 \leq j \leq k \leq |TDB|$, represents the first and last $tids$ of that range respectively, in which a pattern has appeared periodically in a subset of database and per_i^X is the $periodicity$ of a pattern within the interval whose tids are within tid_j^X and tid_k^X. Let tid_j^X be denoted as first and tid_k^X be denoted as last in the respective intervals. Let $PS^X = \{ps_1^X, ps_2^X, \cdots, ps_k^X\}$, $1 \leq k \leq |TDB|$, denote the complete set of *period summaries* of X for any tail-node.

Example 3: Continuing with the previous example, the sub-lists of 'b' will result in following *period summaries* at their respective tail-nodes: $PS^{cadb} = \{\langle 4, 8, 2\rangle, \langle 17, 17, 0\rangle\}$ (b along with c, a and d) and $PS^b = \{\langle 5, 5, 0\rangle, \langle 10, 20, 2\rangle\}$ (b in isolation). The first element in PS^{cadb}, says that '$cadb$' has occurred with the $periodicity$ of 2 in the sub-database whose $tids$ are from 4 to 8. The second element in PS^{cadb}, says that '$cadb$' has occurred with $periodicity$ of 0 in the sub-database whose $tids$ are from 17 to 17.

In Figure 5 and Figure 6, we can see how occurrences of pattern 'cad' are stored in the existing methods and proposed method respectively.

The merging process of *period summaries* at different tail-nodes are as follows.

In the mining phase, *period summaries* at different tail-nodes have to be merged to generate the *final period summary* of the pattern to determine whether a pattern X is periodic or not. We encounter the following cases while merging two intervals:

1) One interval is subset of another interval.
2) Both the intervals are overlapping.
3) Both the intervals are non-overlapping, and the difference between them is less than $maxPer$.
4) Both the intervals are non-overlapping, and the difference between them is greater than $maxPer$.

In the first three cases, we merge both the intervals and store the extended interval in *final period summary*. In the fourth case, we store both the intervals separately in *final period summary*. The periodicity of the merged element is calculated as follows:

$$per_k^X = maximum(per_i^X, per_j^X, (tid_i^X(l) - tid_j^X(f)))$$

In the above equation, per_i^X and per_j^X are the periodicity values of i^{th} and j^{th} *period summaries*. The $tids$ $tid_i^X(l)$ and $tid_j^X(f)$ represent the last tid and first tid of the i^{th} and j^{th} *period summaries* which are to be merged.

Example 4: The merging starts with pointers P_1 and P_2 pointing to the start of PS^{cadb} and PS^b respectively. Since $\langle 5, 5, 0\rangle$ of P_2 is subset of $\langle 4, 8, 2\rangle$ ($4 < 5 < 8$), we increment P_2 and point it to $\langle 10, 20, 2\rangle$. Now, intervals $\langle 4, 8, 2\rangle$ and $\langle 10, 20, 2\rangle$ are non-overlapping and the difference between rightmost tid of P_2 and leftmost tid of P_1 is less than $maxPer$ (case 3) i.e., $10-8 = 2 < maxPer(= 4)$. Therefore, we merge the intervals to form $\langle 4, 20, 2\rangle$ and add this element to *final period summary*. The periodicity of the added element

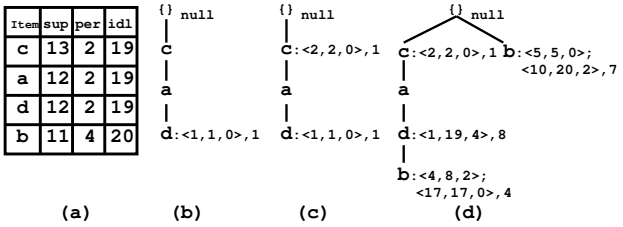

Figure 7: Construction of PS-Tree. (a) PS-list (b) After scanning the first transaction (c) After scanning the second transaction and (d) After scanning all transactions.

is $maximum(2, 2, 2) = 2$. Now, we move our pointer P_1 to $\langle 17, 17, 0\rangle$. P_1 is subset of $\langle 4, 20, 2\rangle$. Hence, the *final period summary* of 'b' gets merged into single element as $\{\langle 4, 20, 2\rangle\}$.

Let PS be the *final period summary*. To check if a pattern is periodic or not, we check for three conditions in PS :

- $PS.size() = 1$
- $PS[0].first \leq maxPer$
- $(|TDB| - PS[0].last) \leq maxPer$

Example 5: Continuing with the previous example, the *final period summary* of 'b' is $\{\langle 4, 20, 2\rangle\}$. Since it satisfies all the three conditions mentioned above (i.e., $size = 1,\ 4 \leq 4$ and $(20 - 20) = 0 \leq 4$), we say pattern 'b' is periodic in the database.

This optimization helps in reducing the memory requirements for constructing a tree. The advantage will be even more if the original tid-list does not split into multiple branches as we just store one element in the tail-node implying that it is periodic in the entire range of database.

The algorithm involves the following two steps: (i) Construction of PS-tree and (ii) Recursively mining PS-tree to discover periodic-frequent patterns. Before we discuss these two steps, we describe the structure of PS-tree.

C. PS-Tree: Structure and Construction

*1) **Structure of PS-Tree**:* Period Summary tree contains PS-list and a summarized prefix-tree. A PS-list consists of three fields: $itemname$ (Item), $support$ (sup) and $periodicity$ (per). Two types of nodes are maintained in PS-tree: ordinary node and tail-node. The former is the type of the node similar to that used in PF-tree, whereas the latter node represents the last item of any sorted transaction. In PS-tree, we maintain the *period summary* of occurrences of that branch's items only at the tail-node of every transaction. The tail-node structure maintains (i) the summarized tid list (*period summaries*), which is of the structure explained in Definition 1 and (ii) the support of that branch's items.

*2) **Construction of PS-Tree**:* In the first database scan, the PS-growth scans the database contents and constructs the PS-list to discover periodic-frequent items of size 1. The construction of PS-list is same as the construction of PF-list (Algorithm 1). The final *PS-list* is shown in Figure 1(d).

In the second database scan, the items in the *PS-list* will take part in the construction of PS-Tree. The tree construction starts by inserting the first transaction, *(1, acdg)*, according to PS-list order, as shown in Figure 7(a). All the items in the transaction are inserted in the same order as in PS-list except 'g'. The tail-node '$d : [\langle 1, 1, 0\rangle]$, 1' carries the summarized transaction-id list and support. In the similar fashion, the second transaction, *(2, cef)*, is also inserted into the tree. The tail-node structure for the second transaction would be '$c : [\langle 2, 2, 0\rangle]$, 1' (see Figure 7). The final PS-tree as shown in Figure 7.

Example 6: Consider the tail-node structure of 'd', from the branch 'cad'. The original occurrences of 'cad' in TDB are {1, 3, 7, 9, 11, 13, 15, 19} and $maxPer = 4$. For the first transaction, the list will be $\{\langle 1, 1, 0\rangle\}$, 1. For the third transaction, it checks for condition $1 + 4(= maxPer) \geq 3$. Since it is satisfied, the list's last occurrence will be updated as $\{\langle 1, 3, 2\rangle\}$, 2. In a similar fashion, the list gets updated for each occurrence of the pattern 'cad'. After scanning all the transactions, the tail-node structure of 'd' in 'cad' will be $\{\langle 1, 19, 4\rangle\}$, 8.

D. Mining of PS-tree

The mining process starts by considering the last item i in the PS-list (least $support$) and constructing its prefix tree (PT_i), which has the prefix sub path of the nodes labeled *i*. We update the $support$ and $periodicity$ values by merging the summarized tid-lists (Algorithm 3) of each item in the PS-list by traversing the item node pointers. All items in PS-list whose $support$ is greater than $minSup$ and $periodicity$ is less than $maxPer$ are used in conditional tree - CT_i (Figure 8(c)).

Algorithm 2 PS-growth (PS-Tree, α)

1: Select the last element in the PS-list.
2: **for** each a_i in header of Tree **do**
3: Generate pattern $\beta = a_i \cup \alpha$.
4: Aggregate all of the a_i's summarized tid-lists into PS^{β}, using Algorithm 3.
5: **if** PS^{β}.support $\geq minSup$ and Check(TS^{β}) **then**
6: Construct β's conditional PS-tree, $Tree_{\beta}$.
7: **if** $Tree_{\beta} \neq \phi$ **then**
8: Call PS-growth ($Tree_{\beta}$, β)
9: Remove a_i from the tree
10: Push a_i's summarized tid-list to its parent nodes using Algorithm 3.

For each item j in PT_i, we aggregate all of its nodes' summarized transaction-id list to derive the summarized transaction-id list of the pattern ij, i.e., summarized TID_{ij}. Next, to check if the final summarized transaction-id list is periodic or not, we use Algorithm 4. If j satisfies Algorithm 4 and $support > minSup$, we consider j as periodic-frequent in PT_i. Choosing every periodic-frequent item j in PT_i , we construct its conditional tree, CT_{ij}, and mine it recursively to discover the patterns.

After finding all periodic-frequent patterns for a suffix item i, we prune i from the original PS-tree and push the corresponding nodes' summarized transaction-id lists to their parent

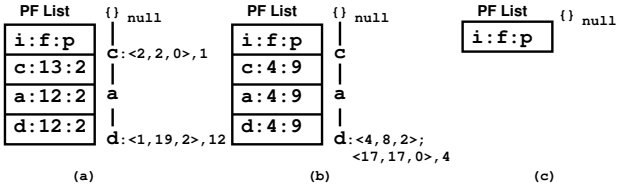

Figure 8: Mining using PS-growth algorithm. (a) PS-Tree after removing 'b' (b) Prefix tree of 'b' (c) Conditional Tree of 'b'.

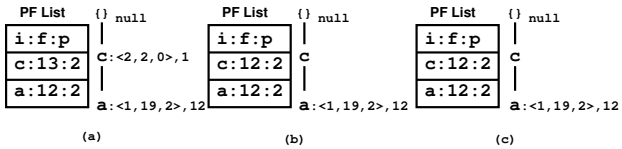

Figure 9: Mining using PS-growth algorithm. (a) PS-Tree after removing 'd' (b) Prefix tree of 'd' (c) Conditional Tree of 'd'.

Algorithm 3 Aggregating Intervals(I_1, I_2)

```
Initialize interval vector I_3
if I_1.first > I_2.first then
    swap (I_1, I_2)
if I_1.last > I_2.last then
    I_3.append(I_1.first, I_1.last)
else if I_1.last + maxPer ≥ I_2.first then
    I_3.append(I_1.first, I_2.last)
else
    I_3.append(I_1.first, I_1.last)
    I_3.append(I_2.first, I_2.last)
return I_3
```

nodes. The above steps are repeated until PS-list becomes NULL. PFP-growth and PS-Growth generate the same set of periodic-frequent patterns.

Algorithm 4 Check (I)

```
if I.size() = 1 then
    if I[0].first <= maxPer then
        if |TDB| − I[0].last <= maxPer then
            return TRUE
return FALSE
```

V. Experimental Results

In this section, we compare the performance of the proposed approach (PS-growth) with the existing approaches (PFP-growth [1], PF-growth++ [10] and ITL-Tree [11]). The structure of PF-tree++ is the same as the structure of PF-tree. Since, PF-growth++ focuses only on improving the runtime for extracting periodic-frequent patterns. Only runtime of PF-growth++ is compared with the proposed approach. All the algorithms are written in GNU C++ and run with CentOS-7.1 on a 3.00GHz machine with 8GB of memory. The runtime specifies the total execution time, i.e., CPU and I/Os. Details from */proc/<pid>/stat* are used to compute the process' CPU usage (RAM).

A. Datasets description

The real-world datasets, mushroom, twitter, and retail, have been used for conducting experiments. The mushroom dataset is obtained from UCI Machine Learning Repository [13]. It is a dense dataset containing 8,124 transactions and 119 distinct items. The twitter dataset is provided by Kiran et al. [5], and contains hashtags appeared on May 1^{st}. The dataset contains 12 hours data with 43,200 transactions and 44,201 items. Retail dataset is large scale sparse dataset containing 88,162 transactions with 16,470 items. The retail database is provided by Brijs et al. [14], and contains market-basket data of an anonymous Belgian retail supermarket store.

B. Performance Evaluation

Performance evaluation was done on three factors: tree memory, total memory and time consumed. Total memory consumed is usually very high compared to tree memory as we construct a conditional tree for every pattern we mine.

Figures 10(1) - 10(3) show the number of periodic-frequent patterns generated in various datasets at different $maxPer$ values. It can be observed that increase in $maxPer$ results in an increase in number of periodic-frequent patterns because more patterns satisfy $periodicity$ condition.

Figures 10(4) - 10(6) show the memory requirements of PS-tree, ITL-Tree and PF-tree (or PF-tree++) for different datasets at different $maxPer$ values. Figures 10(7) - 10(9) show the total memory consumed by PS-growth, ITL-Growth and PFP-growth (or PFP-growth++) algorithms for various datasets at different $maxPer$ values. Figures 10(10) - 10(12) show the total runtime consumed to discover periodic-frequent patterns by PS-growth, ITL-Growth, PFP-growth and PFP-growth++ algorithms at different $maxPer$ values.

As $maxPer$ increases, the two reasons which decide the memory/time consumption are number of *period summaries* stored and number of patterns generated. As $maxPer$ increases, the number of *period summaries* stored (proposed approach) decreases, resulting in lower consumption of memory. At the same time, as $maxPer$ increases, number of patterns also increase, resulting in higher memory consumption. Considering these two factors, memory/time consumption will either increase or decrease (proposed approach), depending on the dataset.

Overall, it can be observed from Figure 10 that PS-growth is memory and runtime efficient as compared to the existing algorithms. In dense datasets, mushroom and twitter, PS-growth outperform PFP-growth (or PFP-growth++) by a wider margin. However, in sparse datasets like retail, PS-growth outperforms PFP-growth (or PFP-growth++) by a narrow margin.

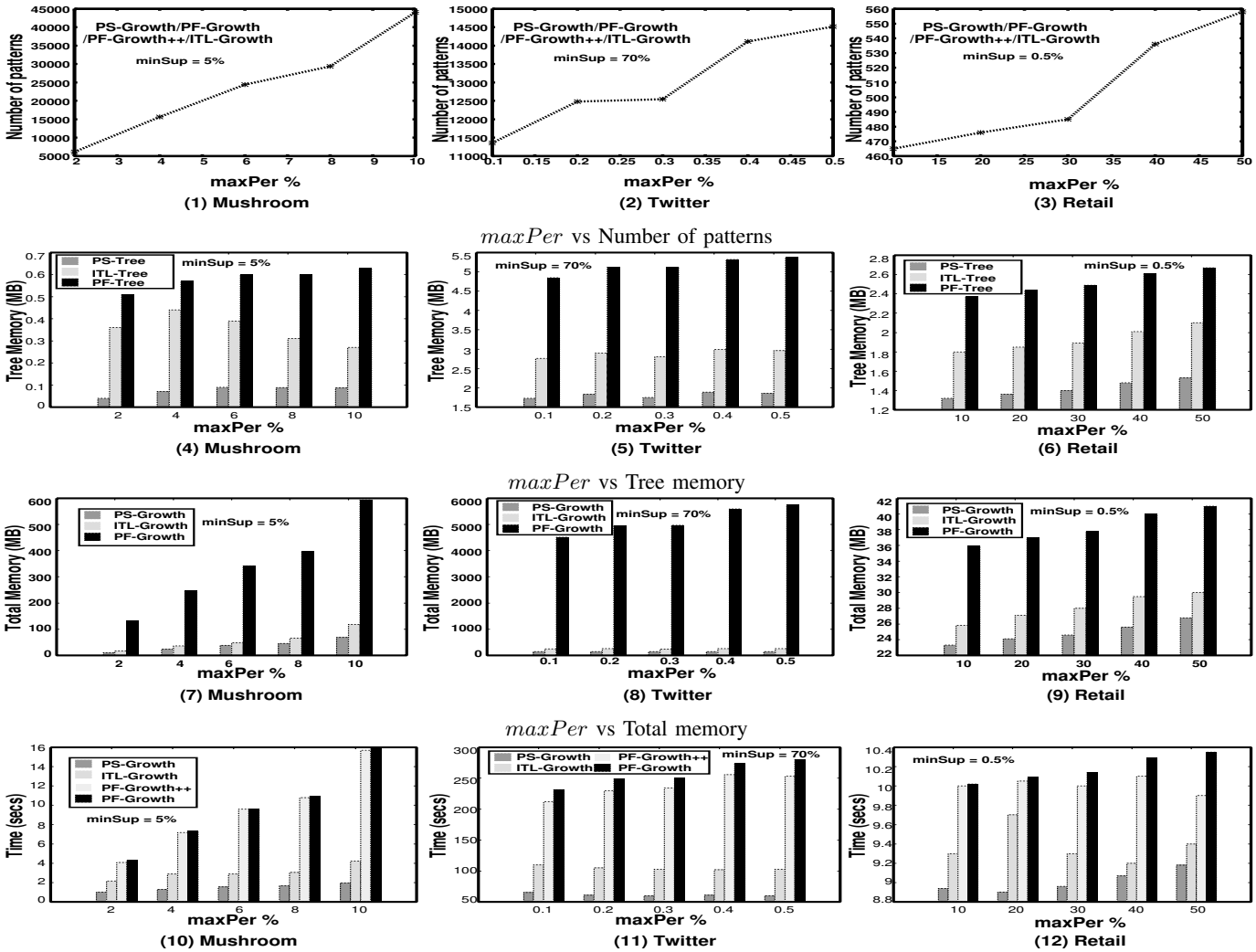

Figure 10: Comparative analysis by varying maxPer

Figure 11 shows how memory/time consumed change at different $minSup$ values. It can be observed that increase in $minSup$ results in decrease of the number of periodic-frequent patterns generated. Thus tree memory, total memory and run time decreases as $minSup$ increases. From Figure 11(8), we can notice that the existing approaches failed to extract periodic-frequent patterns for $minSup < 65\%$ because of over consumption of RAM (exceeded 8 GB).

C. Discussion

For patterns containing millions of transaction-ids, the proposed compact representation of transaction-ids using the notion of *period summaries* will result in a huge reduction in memory and run time requirements. The reduction in memory is because instead of storing thousands of transaction-ids in the tail-node, we store the summary of these transactions in very few intervals. The reduction in run time to extract periodic-frequent patterns is because the existing approaches require the scanning of the entire transaction-id list for determining the periodicity of a pattern whereas in the proposed approach we just have to check for three conditions (Algorithm 4), which can be done in $O(1)$ time.

VI. Conclusions and Future Work

In this paper we have proposed an efficient approach to extract periodic-frequent patterns from large transactional databases by using the notion of *period summaries*. The proposed approach reduces the computational cost without missing any knowledge pertaining to periodic-frequent patterns as compared to the existing approaches. As a part of future work, we are planning to explore efficient approaches to extract periodic-frequent patterns for incremental data sets.

VII. Acknowledgment

This work was supported by the Research and Development on Real World Big Data Integration and Analysis program of the Ministry of Education, Culture, Sports, Science, and Technology, JAPAN.

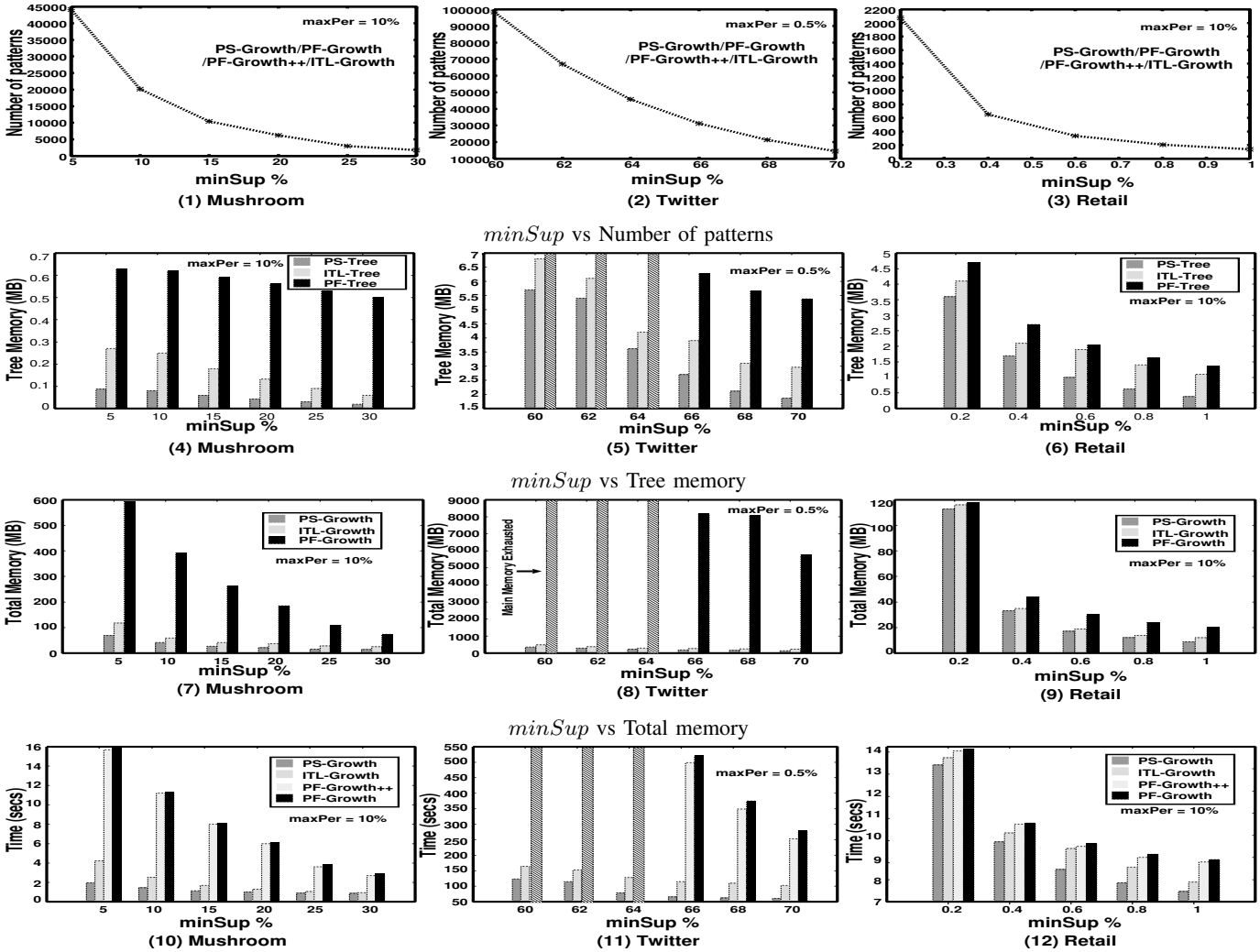

Figure 11: Comparative analysis by varying minSup

REFERENCES

[1] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee, "Discovering periodic-frequent patterns in transactional databases," in *PAKDD*. Springer, 2009, pp. 242–253.

[2] E. Keogh, J. Lin, and A. Fu, "Hot sax: Efficiently finding the most unusual time series subsequence," in *ICDM*, 2005, pp. 226–233.

[3] M. Zhang, B. Kao, D. W. Cheung, and K. Y. Yip, "Mining periodic patterns with gap requirement from sequences," *KDD*, vol. 1, 2007.

[4] H. Stormer, "Improving e-commerce recommender systems by the identification of seasonal products," pp. 92–99.

[5] R. U. Kiran, H. Shang, M. Toyoda, and M. Kitsuregawa, "Discovering recurring patterns in time series," in *EDBT*, 2015, pp. 97–108.

[6] R. Uday Kiran and P. Krishna Reddy, "Towards efficient mining of periodic-frequent patterns in transactional databases," in *DEXA*. Springer, 2010, pp. 194–208.

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

[8] M. M. Rashid, M. R. Karim, B.-S. Jeong, and H.-J. Choi, "Efficient mining regularly frequent patterns in transactional databases," in *DASFAA*. Springer, 2012, pp. 258–271.

[9] R. U. Kiran and P. K. Reddy, "An alternative interestingness measure for mining periodic-frequent patterns," in *DASFAA*, 2011, pp. 183–192.

[10] R. U. Kiran, M. Kitsuregawa, and P. K. Reddy, "Efficient discovery of periodic-frequent patterns in very large databases," *JSS*, vol. 112, pp. 110–121, 2016.

[11] K. Amphawan, P. Lenca, and A. Surarerks, "Mining periodic-frequent itemsets with approximate periodicity using interval transaction-ids list tree," in *WKDD*, 2010, pp. 245–248.

[12] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *International Conference on Management of Data*. ACM, 2000, pp. 1–12.

[13] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[14] T. Brijs, B. Goethals, G. Swinnen, K. Vanhoof, and G. Wets, "A data mining framework for optimal product selection in retail supermarket data: The generalized PROFSET model," *CoRR*, pp. 300–304, 2001.