PAPER *Special Section on Knowledge, Information and Creativity Support System*

Mining Regular Patterns in Transactional Databases

Syed Khairuzzaman TANBEER[†a)], **Chowdhury Farhan AHMED**[†], ***Nonmembers***, **Byeong-Soo JEONG**[†*], ***Member***, *and* **Young-Koo LEE**[†], ***Nonmember***

SUMMARY The frequency of a pattern may not be a sufficient criterion for identifying meaningful patterns in a database. The temporal regularity of a pattern can be another key criterion for assessing the importance of a pattern in several applications. A pattern can be said *regular* if it appears at a regular user-defined interval in the database. Even though there have been some efforts to discover *periodic* patterns in time-series and sequential data, none of the existing studies have provided an appropriate method for discovering the patterns that occur regularly in a transactional database. Therefore, in this paper, we introduce a novel concept of mining *regular* patterns from transactional databases. We also devise an efficient tree-based data structure, called a Regular Pattern tree (RP-tree in short), that captures the database contents in a highly compact manner and enables a pattern growth-based mining technique to generate the complete set of *regular* patterns in a database for a user-defined *regularity* threshold. Our performance study shows that mining *regular* patterns with an RP-tree is time and memory efficient, as well as highly scalable.

key words: *data mining, pattern mining, regular pattern, periodic pattern*

1. Introduction

Mining patterns that appear frequently in transactional databases [1], [2], [7], [14] has been widely studied for over a decade. The rationale behind mining frequent patterns is that only patterns occurring at a high frequency are of interest to users. However, the utility of frequent patterns is dependent on several application-specific criteria. For example, the cost, profit, or price of an item might be more important factors for business data analysis than the frequency of an item in sales transactions. Another important criterion for determining the importance of a pattern might be the shape of occurrence, i.e., whether the pattern occurs periodically, irregularly, or mostly in a specific time interval.

In a retail market, some products may be sold more regularly than other products. Thus, even though both of the products are sold frequently over the entire selling history or for a specific time period (e.g., for a year), the products still need to be managed independently. That is, it is necessary to identify a set of items that are sold together at a regular interval for a specified time period. Also, to improve web site design, a site administrator may be interested in regularly visited web page sequences rather than web pages that are heavily hit only for a specific period. As for genetic data analysis, scientists might be interested in identifying the set of all genes that co-occur at a fixed interval in the DNA sequence.

In the above examples, we can see that users may be interested on the appearance behavior (*regularity*) of patterns rather than just the frequency of occurrence. The pattern regularity can also be a useful metric in applications related to network monitoring or the stock market, among other applications. Therefore, temporal regularity plays an important role in the discovery of interesting patterns for a wide variety of applications. We define regularly appearing patterns in a database as *regular* patterns.

Consider the transactional database in Table 1, in which the patterns "*a*", "*d*" and "*be*" have support values of 5, 5 and 4, respectively, and occur more frequently in certain parts of the database (i.e., "*a*" at the beginning, "*d*" at the end, and "*be*" in the middle of database). Such patterns may be frequent across the whole database. However, their appearance behaviors do not follow a temporal regularity. In contrast, the appearances of patterns "*c*", "*bc*", "*ce*", "*ef*" etc., are almost evenly distributed throughout the database. Although these patterns are relatively less frequent in a database, they may be important patterns in terms of their regularity of appearance. Traditional frequent pattern mining techniques fail to uncover such *regular* patterns because they focus only on the high frequency patterns.

Motivated by these examples and discussion, in this paper, we address the problem of discovering *regular* patterns in a transactional database. We define a new *regularity* measure for a pattern determined by the maximum interval at which the same pattern occurs in a database. Therefore, *regular* patterns, defined in such way, satisfy the downward closure property [1], i.e., if a pattern is found to be *regular*, then all of its non-empty subsets will be *regular*. Thus, if a pattern is not *regular*, then none of its supersets can be *regular*. In order to mine *regular* patterns based on the downward closure property, we propose a novel tree structure, called an RP-tree (Regular Pattern tree), to capture the

Table 1 A transactional database.

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

Manuscript received April 9, 2008.
Manuscript revised July 4, 2008.

[†]The authors are with the Department of Computer Engineering, Kyung Hee University, 1 Sochun-ri, Kihung-eup, Youngin-si, Kyonggi-do, 446–701, South Korea.
[*]Corresponding author, E-mail: jeong@khu.ac.kr
a) E-mail: tanbeer@khu.ac.kr

DOI: 10.1093/ietisy/e91–d.11.2568

database contents in a highly compact manner. To ensure that the tree structure is compact and informative, only *regular* length-1 items will have nodes in the tree and more frequently occurring items are located at the upper part of the tree to have a better chance of prefix sharing. We use a pattern growth approach to mine *regular* patterns from our RP-tree. Our extensive performance study on both real and synthetic datasets shows that discovering *regular* patterns with an RP-tree is highly memory and time efficient.

The rest of the paper is organized as follows: in Sect. 2, we summarize the existing algorithms to mine *periodic* and *cyclic* patterns that are mostly related to our work; Section 3 introduces the problem definition of *regular* pattern mining; Section 4 provides the details of the structure of RP-tree and the *regular* pattern mining algorithms; Section 5 reports our experimental results; Section 6 is a discussion of some adaptability and limitation issues of the proposed method; and finally, Sect. 7 concludes the paper.

2. Related Work

Han et al. [2] proposed the frequent pattern tree (FP-tree) and the FP-growth algorithm to mine frequent patterns in a memory and time efficient manner. The performance gain achieved by FP-growth is mainly based on the highly compact nature of the FP-tree, which stores only the frequent items in a support-descending order. However, the support metric-based frequent pattern mining models are not appropriate for discovering the special occurrence (i.e., *periodic* or *cyclic*) characteristics of patterns. Mining frequent patterns, *periodic* patterns and *cyclic* patterns in a static database have been well-addressed over the last decade.

Periodic pattern mining problems for time-series data can be categorized into two types; (*i*) *full periodic patterns mining* [3], [8], where every point in time contributes to the cyclic behavior of the time-series and (*ii*) *partial periodic patterns mining* [4], [10], the more general type, which specifies the behavior of the time-series at some point but not at all points in the time-series. However, although *periodic* pattern mining is closely related to our work, it cannot be directly applied in the discovery of *regular* patterns from a transactional database because it considers time-series data.

Periodic pattern mining has also been studied as part of sequential pattern mining [11]–[13], [9] in recent years. In [11], the authors extended the basic form of sequential patterns to cyclically repeated patterns, which is a generalization of the former mining approach. A progressive time list-based verification method to mine *periodic* patterns from a sequence of event sets was proposed in [9]. The technique presented in [13] also generalizes the sequential pattern mining by considering temporal information from the transactions. The above *periodic* pattern mining approaches are also not appropriate for finding *regular* patterns from a transactional database.

Ozden et al. [3] proposed a method to discover the association rules [1] occurring cyclically in a transactional database. This method decomposes the whole database into several non-overlapping segments of fixed time intervals defined by the user. Then it outputs the set of rules that maintains a *cyclic* appearance behavior among all the database segments. The main limitation of this method results from segmenting the database into a series of fixed-sized segments, which may suffer from the *border effect*. That is, if a sufficient number of occurrences of a pattern (to become frequent) occur at the borders of two consecutive segments, the pattern might be ignored to generate association rules.

3. Problem Definition

In this section, we give the preliminaries to formally explain the concept of *regular* pattern mining and we provide related definitions.

Let $L = \{i_1, i_2, \ldots, i_n\}$ be a set of literals, which are items that have been used as a unit of information for an application domain. A set $X = \{i_j, \ldots, i_k\} \subseteq L$, where $j \leq k$ and $j, k \in [1, n]$, is called a pattern (or an *itemset*), or an l-itemset if it contains l items. A transaction $t = (tid, Y)$ is a couple where *tid* represents a transaction-id (or time of transaction occurrence), and Y is a pattern. A transactional database DB over L is a set of transactions $T = \{t_1, \ldots, t_m\}$, $m = |DB|$, where $|DB|$ be the size of DB, i.e., total number of transactions in DB. If $X \subseteq Y$, then t contains X or X occurs in t and is denoted as t_j^X, $j \in [1, m]$. Therefore, $T^X = \{t_j^X, \ldots, t_k^X\}$, $j \leq k$ and $j, k \in [1, m]$ is the set of all transactions where pattern X occurs.

Definition 1 (**a period of pattern** X): Let t_j^X and t_{j+1}^X, $j \in [1, (m-1)]$ be two transactions in which pattern X appears consecutively. The number of transactions or the time difference between t_{j+1}^X and t_j^X can be defined as a period of pattern X, say p^X. That is, $p^X = t_{j+1}^X - t_j^X$, $j \in [1, (m-1)]$. For simplicity in period computation, we consider that there has been a '*null*' transaction with no item at the beginning of DB, i.e., $t_{first} = null$, where t_{first} is the first transaction to be considered. At the same time, t_{last}, the last transaction to be considered, is the mth transaction in DB, i.e., $t_{last} = t_m$. For instance, in Table 1 the set of transactions where pattern "*bf*" appears is $T^{bf} = \{3, 5\}$. Therefore, the periods for this pattern are 3 (i.e., $3 - t_{first}$), 2 (i.e., $5-3$), and 4 (i.e., $t_{last} - 5$), where $t_{first} = 0$ and $t_{last} = 9$.

Definition 2 (***max_period***): For a T^X, let P^X be the set of all periods of pattern X, i.e., $P^X = \{p_1^X, \ldots, p_r^X\}$, where r is the total number of periods of X in T^X. Then, *max_period* of X, is the period with the largest value among all p_k, $k \in [1, r]$. So, $p_{max}^X = Max(t_{j+1}^X - t_j^X)$, $j \in [1, (m-1)]$. Therefore, in the database of Table 1, $P^{bf} = \{3, 2, 4\}$, and $p_{max}^{bf} = 4$.

A pattern will not be *regular* if, at any stage in the database, it fails to satisfy the user-defined *regularity* threshold. The *max_period* of a pattern carries the information about the highest interval (in the number of transactions or in the time period) between two consecutive occurrences in a database. Therefore, we use *max_period* to determine if a pattern is *regular* or not. So, the *regularity* of a pattern X in a DB, denoted as *reg*(X), is the *max_period* of that pattern i.e., $reg(X) = p^X_{max}$. A pattern is called a *regular* pattern if its *regularity* is no more than a user-defined maximum *regularity* threshold *max_period*, λ, with $1 \leq \lambda \leq |DB|$. Thus, pattern X is a *regular* pattern if $reg(X) \leq \lambda$. The *regularity* threshold can also be given as the percentage of database size e.g., $max\_reg = 10\%$ of $|DB|$ may indicate $\lambda = 0.1 \times |DB|$. The *regular pattern mining problem*, given a λ and a *DB*, is to discover the complete set of *regular* patterns in the *DB* having *regularity* of no more than λ. R_{DB} refers to a set of all *regular* patterns in a *DB* for a given *max_reg*.

4. RP-Tree: Design, Construction and Mining

In this section, we first describe the structure of the RP-tree, which was created for efficient *regular* pattern mining. Then, we provide the details of the mining procedure that can be applied to generate a complete set of *regular* patterns from the RP-tree.

Since *regular* patterns follow the downward closure property, *regular* length-1 items will play an important role in *regular* pattern mining. Therefore, it is necessary to perform one database scan to identify the set of length-1 *regular* items. The objective of this scan is to collect the support count and the *regularity* of each item in a database. Consequently, for further processing we can ignore all *irregular* items from each transaction. Let R be the set of all items that are found *regular* at this stage. An RP-tree is constructed in such a way that, it only contains nodes for items in R. To facilitate the construction of a highly compact tree structure, the items in the tree are arranged in support-descending item order (the support information for each item is obtained during the first database scan). It has been shown and proven in [2], [14] that a support-descending tree can provide not only a highly compact tree structure (as FP-tree in [2] and CP-tree in [14]), but also a platform for efficient mining process using pattern growth mining technique (as FP-growth in [2]). We refer interested readers to [2] and [14] for further reading on the influence of sort order on tree structure and pattern growth mining. Inspired by this observation, we follow a mining technique similar to FP-growth mining on a highly compact RP-tree. A detailed discussion on mining on the RP-tree is given in the remaining part of this section.

4.1 The Regular Table

To facilitate the tree traversal and to store all length-1 items, an item header table, called a *regular* table (R-table in short), is built. This table also contains the support information of each item and is created with the first scan of *DB*. The structure of the table is as follows. Each entry consists of four fields in sequence (i, s, t_l, r); item name (i), support (s), *tid* of the last transaction where i occurred (t_l), and the *regularity* of i (r). The item name is just a symbol to identify each item; the total support of an item is reported in s; the value of t_l is used to calculate the interval length of the item's most recent occurrence period; and the maximum period, which

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

Fig. 1 R-table maintenance.

R-table	R-table	R-table	R-table
i: s; t_l; r	i: s; t_l; r	i: s; t_l; r	i: s; t_l; r
a:1;1;1	a:2;2;1	a:5;5;1	a:5;5;4
b:	b:1;2;2	b:6;9;3	b:6;9;3
c:	c:1;2;2	c:5;9;2	c:5;9;2
d:1;1;1	d:1;1;1	d:5;9;5	d:5;9;5
e:	e:1;2;2	e:6;8;2	e:6;8;2
f:	f:	f:3;8;3	f:3;8;3

(a) After *tid* = 1 (b) After *tid* = 2 (c) After *tid* = 9 (d) After refreshing

Fig. 2 R-table population for the *DB* in Table 1.

is the *regularity* of i, is dynamically maintained in r. Let t_{cur} and p_{cur} denote the *tid* of the current transaction and the most recent period for an item X, respectively. The R-table is, therefore, maintained according to the process given in Fig. 1. Figure 2 shows how the R-table is populated with a single scan of the database from Table 1. The first transaction, $\{ad\}$, initializes all R-table entries for items 'a' and 'd', as shown in Fig. 2 (a). According to the procedure given in Fig. 1, since t_{cur} is the first occurrence of both items, the values (for both items) for fields t_l and r are set to t_{cur} ($t_{cur} = 1$) and s to 1. The next transaction ($t_{cur} = 2$) $\{abce\}$ initializes R-table entries for items 'b', 'c' and 'e' with the values $\{s; t_l; r\} = \{1; 2; 2\}$. Item '$a$' also appears in $t_{cur} = 2$, which is reflected in the entries for 'a' by updating its s and t_l values. However, the value of p_{cur} for 'a' ($p_{cur} = 2 - 1$) is not found to be greater than the current value of r (1). Therefore, the updated R-table entries for item 'a' will be $\{2; 2; 1\}$. Figure 2 (b) shows the status of the R-table after processing the second transaction. The R-table after scanning all transactions (i.e., up to $tid = 9$) is given in Fig. 2 (c). To reflect the correct *regularity* for each item in the table, the whole table is refreshed (as mentioned in line 7 of Fig. 1) by updating the r values of each item at the end of database. During this update, the *tid* of the last transaction (i.e., t_{last}) is considered as t_{cur} to calculate the p_{cur} for each item in the R-table. The values of the r fields of all items are, therefore, updated by satisfying the condition in line 5. Figure 2 (d) represents the final contents of the R-table after the first database scan. Note that, due to the refreshing of the R-table, we obtain the correct *regularity* ($r = 4$) for item 'a', which is different from the *regularity* ($r = 1$) it carried up to the end of the database. Values of r for other items are unchanged, since p_{cur} in each case is less than the current r.

Once the R-table is built, we generate R by removing items that are not *regular*. The support values of items in R

are then used for sorting the R-table in support-descending order to facilitate the RP-tree construction. In the next sections, we describe the structure and construction of the RP-tree.

4.2 Structure of the RP-Tree

Before discussing the construction process of an RP-tree in detail, we give a brief description of the RP-tree structure. The structure of an RP-tree consists of one *root* node referred to as the "*null*", a set of item-prefix sub-trees (children of the *root*) and an R-table, consisting of each distinct item with relative *regularity* and a pointer pointing to the first node in the RP-tree that carries the item. The RP-tree contains nodes representing an itemset in the path from the *root* up to that node. With the help of a transaction-id list, called *tid*-list, which is kept only at every last node for every transaction, the tree explicitly maintains the occurrence information for the same transactions in the tree. Hence, there are two types of nodes maintained in an RP-tree; *ordinary* nodes and *tail*-nodes. The former one is the type of node used in an FP-tree [2] whereas the latter one can be defined as follows:

Definition 3 (*tail*_node): Let $t = \{i_1, i_2, \ldots, i_n\}$ be a transaction that is sorted according to the R-table order. If t is inserted into the RP-tree in this order, the node that represents the last item i.e., i_n is defined as the *tail*-node for t. Each *tail*-node maintains a list to explicitly store the *tid*s of transactions when it is the last node.

Irrespective of the node type, no node in the RP-tree needs to maintain a support count value. Like the FP-tree, each node in the RP-tree maintains parent, children, and node traversal pointers. In addition, each *tail*-node maintains a *tid*-list. Therefore, the structures of an *ordinary* node and a *tail*-node are as follows:

For *ordinary* node: N, where N is the item name of the node.

For *tail*-node: $N[t_1, t_2, \ldots, t_n]$, where N is the item name of the node and t_i, $i \in [1, n]$, (with n being the total number of transactions ending at node N) is a transaction-id in the *tid*-list, indicating that N is the *tail*-node for transaction t_i.

Therefore, from the above definitions we can deduce the following lemma.

Lemma 1: A tail-node in an RP-tree inherits an ordinary node; but not the vise versa.

Proof: According to the structure of an *ordinary* node, it exactly and explicitly maintains three types of pointers: a parent pointer, a list of children pointers, and a node traversal pointer. Like an *ordinary* node, a *tail*-node explicitly maintains all such information. Moreover, it maintains another *tid*-list, which is additional information. Therefore, there is an *ordinary* node in every *tail*-node and, in contrast, there is no *tail*-node in an *ordinary* node, since the *tid*-list is not maintained in an *ordinary* node. □

In the next subsection we show the construction of the RP-tree in detail.

4.3 Construction of the RP-Tree

As mentioned before, the construction of an RP-tree requires exactly two database scans: one to collect the R in the R-table with the respective *regularities* of each item and the other to construct the RP-tree based on the R-table sorted in support-descending order. We use an example (in Fig. 3) to illustrate the step-by-step construction process of the RP-tree for the transactional database of Table 1 for $\lambda = 3$.

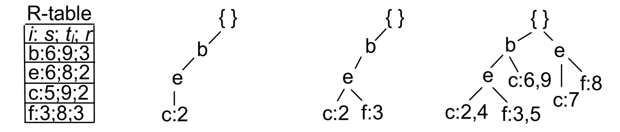
Figure 3 (a) shows the support-descending R-table for $\lambda = 3$, which is constructed through the first database scan by removing the items with *regularity* $> \lambda$ (e.g., items '*a*' and '*d*') from the R-table of Fig. 2 (d). Only the items in the R-table of Fig. 3 (a), therefore, are involved in RP-tree construction. Initially the RP-tree is empty (i.e., starts with a "*null*" root node). It follows the FP-tree construction technique to insert any sorted transaction into the tree. To simplify the figures, we do not show the node traversal pointers in the trees, although they are maintained as in an FP-tree. The first inserted transaction is $\{abce\}$ (i.e., $tid = 2$), since all the items in $tid = 1$ are *irregular*. After removing *irregular* items from $tid = 2$ (i.e., '*a*') and sorting the remaining *regular* items according to the sort order of the R-table in Fig. 3 (a), we insert $\{abce\}$ in the form and an order of $\{bec\}$ in the tree with node "c : 2" being the *tail*-node that carries the *tid* information for the transaction, as shown in Fig. 3 (b). For the next transaction (i.e., $tid = 3$), as in Fig. 3 (c), since its (ordered) *regular* item list (b, e, f) shares a common prefix (b, e) with the existing path (b, e, c), one new node ("f : 3") is created as a *tail*-node with value 3 in its *tid*-list and linked as a child of node ("e"). After scanning all the transactions and inserting them in a similar fashion, the final RP-tree for the database of Table 1 and on $\lambda = 3$ is shown in Fig. 3 (d).

Based on the R-table building technique discussed in Sect. 4.1 and the above example, we have the following property and lemma of an RP-tree.

Let R be the set of all *regular* items for a *regularity* threshold *max_reg* for a given DB. For each transaction t in a DB, $reg(t)$ is the set of all *regular* items in t, i.e., $reg(t) = item(t) \bigcap R$, and is called the *regular* item projection of t.

Property 1: An RP-tree maintains a complete set of *regular* item projections for each transaction in a *DB* only once.

Lemma 2: Given a transactional database *DB* and a *reg-*



(a) *Regular* items ($\lambda = 3$) (b) After inserting *tid* = 2 (c) After inserting *tid* = 3 (d) After inserting *tid* = 9

Fig. 3 Construction of the RP-tree.

ularity threshold *max_reg*, the complete set of all *regular* item projections of all transactions in a *DB* can be derived from the RP-tree for the *max_reg*.

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

The RP-tree constructed in the previous example can achieve a highly compact tree structure while keeping information about *regular* items in a *DB*. One can assume that the structure of the RP-tree may not be memory efficient, since it explicitly maintains *tid*s for each of the transactions in the tree structure. But we argue that the RP-tree achieves memory efficiency by keeping such transaction information only at the *tail*-nodes and avoiding the support count field at each *ordinary* node in the tree. The number of *tail*-nodes in an RP-tree depends on the similarity of the transactions in a *DB*. In the worst case, if all the transactions are different, then this number is equal to the number of transactions in a *DB*. On the other hand, the best case is when every transaction is the same, with the number of *tail*-nodes being one. Moreover, keeping the *tid* information in a tree structure has also been found in literature discussing the efficient mining of frequent patterns [5]–[7]. To a certain extent, some of those approaches additionally maintain a support count and/or the *tid* information [6], [7] in each tree node. Therefore, based on the above discussion we can deduce the following lemma on the size of an RP-tree.

Lemma 3: The size (without the *root* node) of an RP-tree for a transactional database *DB* with a *regularity* threshold *max_reg* is bounded by $\sum_{t \in DB} |reg(t)|$.

Proof: Based on the RP-tree construction process, Property 1 and Lemma 2, each transaction t contributes at best one path of the size $|reg(t)|$ to an RP-tree. Therefore, the total size contribution of all transactions is $\sum_{t \in DB} |reg(t)|$ at best. However, since there are usually a lot of common prefix patterns among the transactions, the size of an RP-tree is normally much smaller than $\sum_{t \in DB} |reg(t)|$. □

In the next subsection, we discuss the mining of *regular* patterns from an RP-tree.

4.4 Mining with an RP-Tree

Construction of a highly compact RP-tree enables the subsequent mining of *regular* patterns to be performed with a rather compact data structure. In this subsection, we study how *regular* patterns can be generated from an already constructed RP-tree.

As mentioned before, *regular* pattern mining is similar to pattern growth mining [2]; the basic operations in the mining approach are (*i*) counting *regular* items, (*ii*) constructing a conditional pattern-base for each *regular* item, and (*iii*) constructing a new conditional tree from each conditional pattern-base. The *regular* pattern mining technique proceeds to recursively mine an RP-tree of decreasing size to generate *regular* patterns without candidate generation or an additional database scan. It does so by examining all the conditional trees of the RP-tree, which consists of the set of *regular* patterns occurring with a suffix pattern. Before discussing these operations in detail, we explore the following important property and lemma of an RP-tree related to mining phase.

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

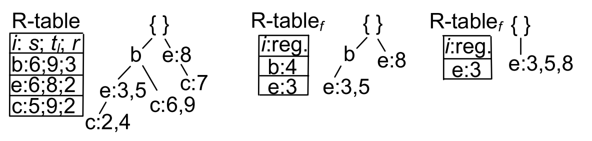
Lemma 4: Let $Z = \{a_1, a_2, \ldots, a_n\}$ be a path in an RP-tree where node a_n, being the *tail*-node, carries the *tid*-list of the path. If the *tid*-list is pushed-up to node a_{n-1}, then node a_{n-1} maintains the information on the occurrence of path $Z' = \{a_1, a_2, \ldots, a_{n-1}\}$ for the same set of transactions in the *tid*-list without any loss.

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

Using the features revealed by the above property and lemma, we give, in detail, how conditional pattern-bases and corresponding conditional trees can be constructed during the mining of *regular* patterns.

Counting *regular* items: Counting *regular* items is facilitated with the help of the R-table containing each distinct item along with its respective *regularity*. For example, the R-table in Fig. 3 (a) represents the set of *regular* items for the RP-tree in Fig. 3 (d) for $\lambda = 3$.

Conditional pattern-base construction: The conditional pattern-base is constructed starting from the item at the bottom of the R-table. While creating the conditional pattern-base, a small R-table for that item is also created. The conditional pattern-base for an item i, PB_i and the corresponding R-table$_i$ are created as follows. Since i is the bottom-most item in the R-table, each node in the RP-tree labeled i must be a *tail*-node. To facilitate the construction of a conditional pattern-base for the next item in the R-table, based on Lemma 4, the *tid*-lists of all such *tail*-nodes are pushed-up to their respective parent nodes in the original RP-tree and in PB_i. Therefore, each parent node is converted to a *tail*-node if it was an *ordinary* node, otherwise, the *tid*-list is merged with its previous *tid*-list. Only the prefix sub-paths of nodes labeled i in the RP-tree are then accumulated as PB_i. All nodes labeled i and the entry for an item i are, thereafter, deleted from the original RP-tree and corresponding R-table. Figure 4 (a) shows the structure of the RP-tree and the corresponding R-table of Fig. 3 (d) after creating the conditional pattern-base for item '*f*'. Each parent of node "*f*" in Fig. 3 (d) (nodes "*e*" in this example) becomes a *tail*-node by receiving the *tid*-list of its child "*f*".



(a) RP-tree after creating conditional pattern-base for '*f*' (b) Conditional pattern-base for '*f*' (c) Conditional tree for '*f*'

Fig. 4 Conditional pattern-base and conditional tree construction with the RP-tree.

All nodes of '*f*' in the RP-tree and its entry in the R-table are deleted thereafter.

The structure of the R-table$_i$ is different from that of the original R-table and consists only of the items present in PB_i with their corresponding *regularities*. To compute the *regularity* of each item *j* in the R-table$_i$, based on Property 2, we map the *tid*-list of every node of *i* to all items in the respective path explicitly in a temporary array (one for each item), while constructing PB_i. Upon completion of the construction of PB_i, the contents of a temporary array for *j* in the R-table$_i$ represent the T^{ij} (i.e., set of all *tid*s where items *i* and *j* occur together) in PB_i. Therefore, it is a rather simple calculation to compute $reg(j)$ from T^{ij} by generating P^{ij}. Figure 4 (b) shows the conditional pattern-base for '*f*', PB_f and the corresponding R-table$_f$ constructed from the original RP-tree of Fig. 3 (d).

Conditional tree construction: The conditional tree for *i* CT_i is, then, constructed from its conditional pattern-base PB_i by removing all *irregular* items and their respective nodes from the R-table$_i$ and PB_i. If the deleted node is a *tail*-node, its *tid*-list is pushed-up to its parent node. For example, the conditional tree of Fig. 4 (c) can be generated for '*f*' by removing *irregular* item '*b*' from the R-table$_f$ and PB_f.

Let *j* be the bottom-most item in R-table$_i$ of CT_i. Then the pattern "ij" is generated as a *regular* pattern with the *regularity* of *j* in the R-table$_i$. The same process of creating a conditional pattern-base and its corresponding conditional tree is repeated for further extensions of pattern "ij". The whole process is repeated if R-table $\neq \emptyset$.

With the above mining process, one can see that, from an RP-tree constructed on a *DB*, the complete set of *regular* patterns for a given *max_reg* can be generated with the pattern growth technique. The technique is efficient due to the support-descending item order in the RP-tree structure. Furthermore, performing the mining operation from the bottom to the top dramatically shrinks the search space during the mining process. In the next section, we present the experimental results of finding *regular* patterns.

5. Experimental Results

In this section, we present a comprehensive experimental analysis and performance results on mining *regular* patterns with our RP-tree. All programs are written in Microsoft Visual C++ 6.0 and run in a time-sharing environment with Windows XP operating system on a 2.66 GHz dual core machine with 1 GB of main memory. The runtime specifies the total execution time, i.e., CPU and I/Os. The runtime reported in different figures in this section are the average of multiple runs in each case.

To perform the experimental study, we use the datasets frequently used in frequent pattern mining experiments, since such datasets maintain the characteristics of a transactional database. Several synthetic datasets (e.g., *T10I4D100K*, *T5I2*, *T20I6*), developed at the IBM Almaden Quest research group and obtained from *http://cvs.buu.ac.th/mining/Datasets/synthesis_data/*, and real datasets (e.g., *chess*, *mushroom*, *connect-4*, *Kosarak*) from UCI Machine Learning Repository (*University of California - Irvine, CA*) were considered. We obtained consistent results for all of the above datasets. However, in the remaining part of this section, due to space constraints, we report only the experimental results on a subset of theses datasets (*T5I2*, *T10I4D100K*, *chess*, *mushroom* and *Kosarak*). Among these datasets *chess* and *mushroom* are dense, but the others are sparse; and *T10I4D100*, *T5I2* (with around $100K$ transactions) and *Kosarak* (with around $1M$ transactions) are reasonably large datasets. We divide the experimental analysis into four parts. First, we show the strengths of the RP-tree structure over a traditional tree structure such as FP-tree; second, we study the compactness of the RP-tree; third, we show its performance with mining the set of *regular* patterns; and finally, we provide the results to prove the scalability of mining *regular* patterns with an RP-tree.

5.1 Tree Structure Comparison

As discussed previously, to achieve a highly compact tree structure with as much prefix sharing among the patterns as possible, the RP-tree captures the *regular* items from the transactions with a support-descending order of items. A support metric-based FP-tree also has an extremely compact tree structure [2], [14] to capture the database content with a minimum time requirement. In this subsection, we compare both of the above tree structures in mining *regular* patterns. However, one limitation of such a comparison is that the FP-tree is not designed to capture *regular* items, but rather, it stores frequent items with a given support threshold. Therefore, for an appropriate comparison with a common parameter, we use the following mapping mechanism between the *regularity* threshold used in RP-tree and the support threshold used in FP-tree in order to ensure that the FP-tree contains the complete set of *regular* patterns (as the RP-tree does) with the given *max_reg* value. The support count of all *regular* patterns in a *DB* for $max\_reg = \lambda$ must be at least $^{|DB|}/_{\lambda}$. For example, the support count of all *regular* patterns in the *T10I4D100K* dataset (with 100,000 transactions) for max_reg = 0.3% of $|DB|$ (i.e., λ = 300) must be at least 334 (i.e., $^{100,000}/_{300}$). Thus, we can say that the set of all frequent patterns (F_{DB}) satisfying the support threshold $^{|DB|}/_{\lambda}$ must contain R_{DB} for $max\_reg = \lambda$. Therefore, an alternate approach for discovering the R_{DB} for λ can be to mine F_{DB} for

Table 2 Required memory (MB).

Dataset	*max_reg*(%)	FP-tree	RP-tree
T10I4D100K	0.3	13.70	1.96
	0.4	13.96	5.12
	0.5	14.07	7.08
	0.6	14.14	9.06
chess	0.1	0.54	0.01
	0.2	0.70	0.04
	0.3	0.73	0.07
	0.4	0.74	0.18

Table 3 Tree construction time (Sec.).

Dataset	*max_reg*(%)	FP-tree	RP-tree
T10I4D100K	0.3	93.25	67.86
	0.4	94.75	72.25
	0.5	95.80	75.41
	0.6	97.49	79.66
chess	0.1	5.98	5.91
	0.2	6.03	5.88
	0.3	6.11	5.89
	0.4	6.06	5.94

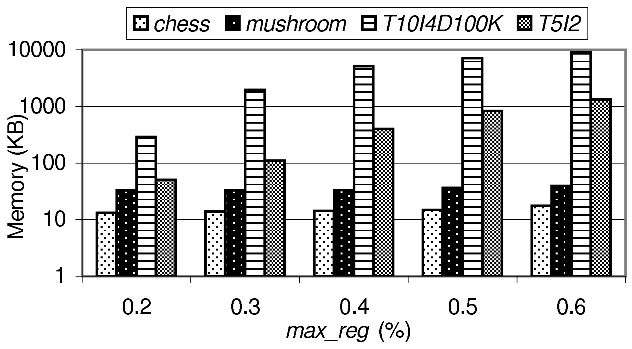


Fig. 5 Compactness of the RP-tree.

Table 4 Pattern count on dataset and *max_reg*.

Dataset	*max_reg* (%)	# of pattern	Dataset	*max_reg* (%)	# of pattern
mushroom	1.0	977	*T10I4D100K*	0.2	19
	2.5	8829		0.6	309
chess	0.1	5	*T5I2*	0.5	78
	0.6	4839		3.0	1522

$|DB|/\lambda$ threshold from the FP-tree using the FP-growth mining technique and then to find R_{DB} with an additional scan of *DB*. It has been observed in our experiment that mining *regular* patterns from an FP-tree is multiple orders of magnitude slower compared to our proposed method, because of the extra processing cost for the additional database scan and finding R_{DB} from F_{DB}. For instance, the mining time required for an FP-tree to find R_{DB} for the *regularity* threshold of 0.1% of |*T10I4D100K*| (with appropriately mapping the threshold) is about 2,000 seconds. RP-tree, on the other hand, requires less than 1 second for mining the same set of *regular* patterns. Therefore, we only show the comparison of tree size and construction time between an RP-tree and an FP-tree for a fixed *max_reg* over different datasets.

We report the results for a sparse (*T10I4D100K*) and a dense (*chess*) dataset in Table 2 and Table 3 for the comparison of required memory and tree construction time, respectively, between two tree structures. It is noteworthy that RP-tree requires significantly less memory for different *max_reg* values for both datasets. Although the tree construction costs are almost similar for *chess*, the FP-tree requires a higher time for *T10I4D100K*. The reasons for the advantage of the RP-tree over the FP-tree are two-fold. First, even though all the frequent items for the support threshold mapped for the *max_reg* may not be *regular*, the FP-tree maintains a complete set of such items. RP-tree, on the other hand, captures only the length-1 *regular* items for the same *max_reg*, which is normally a subset of length-1 frequent items. Second, RP-tree avoids maintaining the support count field at each node in the tree structure.

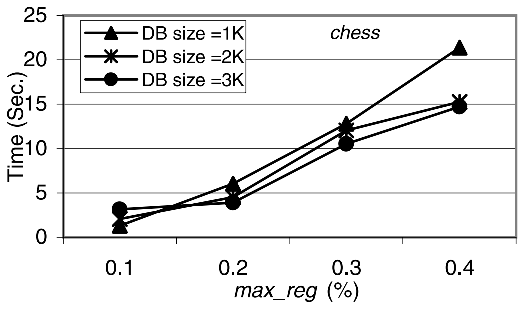
Therefore, with no additional overhead compared to FP-tree, the RP-tree can capture database contents for mining *regular* patterns in both memory and time efficient manner.

5.2 Compactness of the RP-Tree

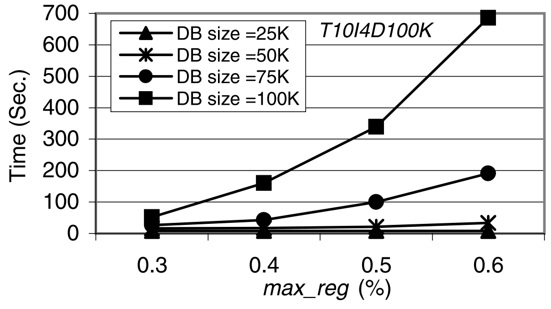
The memory consumptions of RP-tree for different values of *max_reg* over several dense and sparse datasets are reported in Fig. 5. The *x*-axis in the graph indicates the change in *max_reg* values as a percentage of the data point. The figure demonstrates that the higher the *max_reg* value, the more memory required by an RP-tree. However, the rate of memory consumption is almost steady in dense datasets compared to in sparse datasets. This steady rate is due to less variation in pattern *regularities* with the change in the *max_reg* value for dense datasets compared to sparse datasets. The results for *chess* and *mushroom* reflect this scenario, while memory requirements for *T10I4D100K* and *T5I2* greatly vary with *max_reg* as shown in the figure. However, it is clear from the figure that, the structure of the RP-tree can easily be handled in a memory efficient manner irrespective of the dataset type (dense or sparse) or size (large or small). In the next experiment, we focus on the execution time requirement for mining *regular* patterns with the RP-tree.

5.3 Execution Time Performance of the RP-Tree

In this subsection, we report the execution time that the RP-tree requires for mining *regular* patterns over datasets of different types on changes in the *max_reg*. The execution time encompasses all the phases of R-table construction, the RP-tree building and the corresponding mining. Due to space limitations, we report only the results on one sparse (*T10I4D100K*) and one dense (*chess*) dataset. However, the total numbers of *regular* patterns in R_{DB} generated by our experiments for different *max_reg* values across several datasets are provided in Table 4. We can observe from the table that the higher the value of *max_reg*, the greater the numbers of *regular* patterns in all datasets. This is due to the fact that as the *max_reg* increases, there is a greater possibility of getting more *regular* patterns compared to low *max_reg* values. This is why we need a longer execution time for higher *max_reg* values, which are shown in Fig. 6 for the *chess* and *T10I4D100K* datasets. To grasp the effect of mining on the variation in size of such datasets, we performed *regular* pattern mining while increasing the size of both of the datasets from $1K$ to $3K$ (full dataset) for *chess* and $25K$ to $100K$ (full dataset) for *T10I4D100K*. Thus, the plots for $3K$ and $100K$ represent the results for the full size of both datasets, respectively. The graphs show that both of the datasets require more execution time when mining larger datasets. As the database size and *max_reg* value increase, the tree structure size increases. Therefore, a comparatively longer time is needed to generate *regular* patterns from large trees. However, when the whole database and reasonably high *max_reg* values are concerned, we see that mining *regular* patterns from the corresponding RP-tree is rather time efficient for both sparse and dense datasets. The scalability studies we perform in the next subsection also reflect this scenario.



(a) Execution time over *chess*



(b) Execution time over *T10I4D100K*

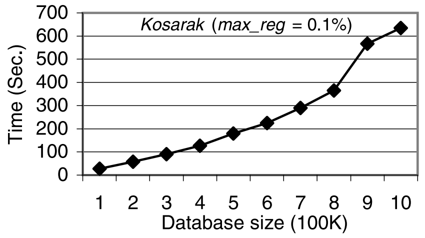
Fig. 6 Execution time on the RP-tree.

5.4 Scalability of the RP-Tree

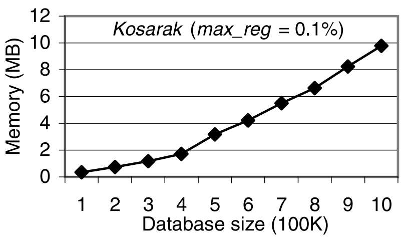
We study the scalability of the RP-tree by varying the number of transactions in the database and the value of the *regularity* threshold. We use *Kosarak* dataset for the scalability experiment, since it is a huge sparse dataset with a large number of distinct items (41,270) and transactions. To test the scalability on execution time and memory consumption, we divided this dataset into ten portions with 0.1 million transactions per portion. Then we investigated the performance of the RP-tree after combining each portion with previous parts by performing *regular* pattern mining each time. We fix the *max_reg* value to 0.1% of the database size. The experimental results are shown in Fig. 7 (a) and Fig. 7 (b). The time in the y-axis of Fig. 7 (a) specifies the total execution time with the increase in database size. The tree-building time includes the tree construction time from the first transaction up to the number of transactions for the data point on the x-axis. Clearly, as the database size increases, the overall tree construction and mining time, and required memory (Fig. 7 (b)) increase. However, the RP-tree shows a stable performance with a linear increase in runtime and memory consumption as the database size increases. The execution times of RP-tree on *Kosarak* with the *max_reg* increasing from 0.02% to 0.1% of the database size are shown in Fig. 7 (c), which demonstrates that the RP-tree can mine R_{DB} on this dataset for a reasonably large value of *max_reg* with a considerable amount of execution time. Therefore, the scalability results in the graphs show that the RP-tree is highly scalable for database size, memory, and *max_reg* values.

6. Discussions

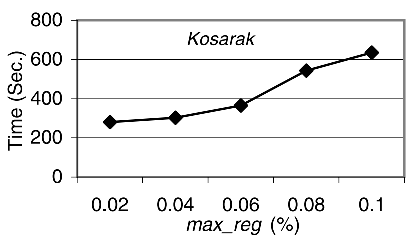
In Sect. 4.4, we have shown that during mining *regular* patterns from an RP-tree the conditional pattern-bases explicitly provide the *tid*-lists of all patterns along with respective suffixing patterns. In this paper, the maximum occurrence interval (i.e., *max_period*) of a pattern (calculated from the *tid*-list) in a database is considered to be its *regularity*. How-



(a) On number of transactions



(b) On memory



(c) On *max_reg*

Fig. 7 Scalability of the RP-tree.

ever, other parameters such as the arithmetic mean or variance of occurrence intervals can also be considered as the measures of *regularity* for finding interesting patterns from a database. Since the RP-tree maintains the occurrence information for each transaction in the tree structure and the mining phase provides the complete *tid*-list for each pattern, computing such parameters can also be simple similar to computing the maximum occurrence interval for a pattern. However, the technique to find all length-1 *regular* items with the first scan of a database needs to be revised to compute the *regularity* accordingly to new measures. It can be noted that the sets of patterns discovered with different *regularity* measures might be different, since they are dependent on the definition of the *regularity* measure. Although in this paper we focus on the patterns generated by considering *max_period* as the *regularity* measure with a minor computation tuning the proposed tree structure and the mining technique can easily be adapted to above new measure of *regularity*.

One limitation of using the *max_period* measure for the *regularity* calculation might be its susceptibility to erroneous or noisy data. For example, the set of all transactions where pattern "*b*" occurs in the database of Table 1 is $T^b = \{2, 3, 4, 5, 6, 9\}$. For a value of $max\_reg = 3$, pattern "*b*" is a *regular* pattern, since $reg(b) = 3$. If the interference of noise or error causes any transaction between $tid = 2$ to $tid = 5$ to be deleted or modified to remove item '*b*', the pattern "*b*" still remains as a *regular* pattern. However, if item '*b*' from $tid = 9$ is deleted anyhow (due to noise or error), the pattern becomes *irregular* for the given *max_reg*. Therefore, temporal location of noisy data in database sometimes might be an issue as per as the degree of robustness of the proposed method is concerned.

7. Conclusions

In this paper, we introduce the concept of *regular* pattern mining on a transactional database, and we provide a highly compact tree structure, the RP-tree, to capture the database contents and a pattern growth-based mining technique to discover the complete set of *regular* patterns with user-defined maximum *regularity* over a database. To make our RP-tree more memory efficient, we used a novel concept of maintaining transaction information only at the *tail*-nodes, without explicitly maintaining the support information at the other nodes. We have shown that the technique of pushing-up the transaction information from the *tail*-nodes upward during the mining phase enables the RP-tree to efficiently and completely mine *regular* patterns. We discussed our experimental results in detail and demonstrated that our RP-tree can provide time and memory efficiency during *regular* pattern mining. Moreover, our method is highly scalable for time, memory, and *regularity* threshold.

Acknowledgements

We would like to express our deep gratitude to the reviewers of this paper. Their useful and constructive comments have played a significant role in improving the quality of this work.

This research was supported by a grant of the Korea Health 21 R&D Project, Ministry for Health Welfare and Family Affairs, Republic of Korea (A020602).

References

[1] R. Agrawal, T. Imielinski, and A.N. Swami, "Mining association rules between sets of items in large databases," Proc. ACM SIGMOD Conf. on Management of Data, pp.207–216, 1993.
[2] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," Proc. 2000 ACM SIGMOD International Conf. on Management of Data, pp.1–12, 2000.
[3] B. Ozden, S. Ramaswamy, and A. Silberschatz, "Cyclic association rules," Proc. 14th International Conf. on Data Engineering, pp.412–421, 1998.
[4] J. Han, G. Dong, and Y. Yin, "Efficient mining of partial periodic patterns in time series database," Proc. 15th International Conf. on Data Engineering, pp.106–115, 1999.
[5] Y. Chi, H. Wang, P.S. Yu, and R.R. Muntz, "Catch the moment: Maintaining closed frequent itemsets over a data stream sliding window," Knowledge and Information System, vol.10, no.3, pp.265–294, 2006.
[6] M.J. Zaki and C.-J. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," IEEE Trans. Knowl. Data Eng., vol.17, no.4, pp.462–478, April 2005.
[7] X. Zhi-Jun, C. Hong, and C. Li, "An efficient algorithm for frequent itemset mining on data streams," Proc. International Conf. on Management of Data, ed. P. Perner, pp.474–491, 2006.
[8] M.G. Elfeky, W.G. Aref, and A.K. Elmagarmid, "Periodicity detection in time series databases," IEEE Trans. Knowl. Data Eng., vol.17, no.7, pp.875–887, 2005.
[9] K.-Y. Huang and C.-H. Chang, "Mining periodic patterns in sequence data," Proc. DaWaK, eds. Y. Kambayashi, M.K. Mohania, and W. Wöß, pp.401–410, 2004.
[10] G. Lee, W. Yang, and J.-M. Lee, "A parallel algorithm for mining multiple partial periodic patterns," Inf. Sci., vol.176, pp.3591–3609, 2006.
[11] I.H. Toroslu and M. Kantarcioglu, "Mining cyclically repeated patterns," Proc. DaWaK, eds. Y. Kambayashi, W. Winiwarter, and M. Arikawa, pp.83–92, 2001.
[12] M. Zhang, B. Kao, D.W. Cheung, and K.Y. Yip, "Mining periodic patterns with gap requirement from sequences," ACM Trans. Knowledge Discovery from Data, vol.1, no.2, article 7, 2007.
[13] F. Maqbool, S. Bashir, and A.R. Baig, "E-MAP: Efficiently mining asynchronous periodic patterns," International Journal of Computer Science and Network Security, vol.6, no.8A, pp.174–179, Aug. 2006.
[14] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, and Y.-K. Lee, "CP-tree: A tree structure for single-pass frequent pattern mining," Proc. PAKDD, eds. T. Washio, E. Suzuki, K.M. Ting, and A. Inokuchi, pp.1022–1027, 2008.

Syed Khairuzzaman Tanbeer received his B.S. degree in Applied Physics and Electronics and M.S. degree in Computer Science from University of Dhaka, Bangladesh in 1996 and 1998 respectively. Since 1999, he has been working as a faculty member in Department of Computer Science and Information Technology, Islamic University of Technology, Dhaka, Bangladesh. Currently he is pursuing his Ph.D. degree in Department of Computer Engineering, Kyung Hee University, South Korea. His research interests include data mining and knowledge engineering.

Chowdhury Farhan Ahmed received his B.S. and M.S. degrees in Computer Science from University of Dhaka, Bangladesh in 2000 and 2002 respectively. During 2003–2004 he worked as a faculty member in Institute of Information Technology, University of Dhaka, Bangladesh. In 2004, he joined as a faculty member in Department of Computer Science and Engineering, University of Dhaka, Bangladesh. Currently he is pursuing his Ph.D. degree in Department of Computer Engineering, Kyung Hee University, South Korea. His research interests are in the area of data mining and knowledge discovery.

Byeong-Soo Jeong received his B.S. degree in Computer Engineering from Seoul National University, Korea in 1983, M.S. degree in Computer Science from the Korea Advanced Institute of Science and Technology, Korea in 1985 and Ph.D. degree in Computer Science from Georgia Institute of Technology, Atlanta, USA in 1995. In 1996, he joined Kyung Hee University, Korea. He is now an Associate Professor at the College of Electronics & Information at Kyung Hee University. From 1985 to 1989, he was a research staff at the Data Communications Corp., Korea. From 2003 to 2004, he was a visiting scholar at Georgia Institute of Technology, Atlanta. His research interests include database systems, data mining, and mobile computing.

Young-Koo Lee received his B.S., M.S. and Ph.D. in Computer Science from Korea Advanced Institute of Science and Technology, Korea. He is a professor in the Department of Computer Engineering at Kyung Hee University, Korea. His research interests include ubiquitous data management, data mining, and databases. He is a member of the IEEE, the IEEE Computer Society, and the ACM.