Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080446

Association Rule Mining Considering Local Frequent Patterns with Temporal Intervals

Kuo-Cheng Yin[1], *Yu-Lung Hsieh*[1], *Don-Lin Yang*[1,*] *and Ming-Chuan Hung*[2]

[1] Department of Information Engineering and Computer Science, Feng Chia University, Taichung, 407 Taiwan
[2] Department of Industrial Engineering and Systems Management, Feng Chia University, Taichung, 407 Taiwan

Received: 15 Aug. 2013, Revised: 17 Nov. 2013, Accepted: 18 Nov. 2013
Published online: 1 Jul. 2014

Abstract: In traditional association rule mining algorithms, if the minimum support is set too high, many valuable rules will be lost. However, if the value is set too low, then numerous trivial rules will be generated. To overcome the difficulty of setting minimum support values, global and local patterns are mined herein. Owing to the temporal factor in association rule mining, an itemset may not occur frequently in the entire dataset (meaning that it is not a global pattern), but it may appear frequently over specific intervals (meaning that it is a local pattern). This paper proposed a temporal association rule mining algorithm for interval frequent-patterns, called GLFMiner, which automatically and efficiently generates all intervals without prior domain knowledge in an efficient manner. GLFMiner considers not only global frequent-patterns, but also local frequent-patterns. Using the same value of minimum support, it can locate many valuable temporal rules without losing the rules that traditional algorithms may find. Experimental results reveal that our novel algorithm mines more temporal frequent-patterns than traditional association rule mining algorithms and is effective in real-world applications such as market basket analysis and intrusion detection systems.

Keywords: association rule, local frequent-pattern, global frequent-pattern, temporal frequent-pattern

1 Introduction

In recent years, association rule mining has been extensively used in knowledge discovery in various fields. The process of association rule mining involves discovering all rules whose *support* and *confidence* are at least the user-defined minimum *support* and minimum *confidence* thresholds, respectively. For example, examination of the dataset of a supermarket in traditional association rule mining [1,2] for market basket analysis may yield an association rule such as "*Turkey* → *Pumpkin pie* $(support = 0.0001, confidence = 0.05)$", meaning that 0.01% of all transactions contain both turkey and pumpkin pie, and 5% of all transactions that contain turkey also contain pumpkin pie.

The association rule "*Turkey* → *Pumpkin pie*" raises three problems. First, the above rule cannot be regarded as a prominent association rule because its *support* and *confidence* are both too low. Therefore, a turkey and a pumpkin pie are seldom associated in the entire dataset of transactions and many transactions that contain turkey do not contain pumpkin pie. Second, since the association rule "*Turkey* → *Pumpkin pie*" is not prominent, the minimum *support* threshold must be set low enough for the above rule to be found. Consequently, too many association rules that are not very useful may be found, overwhelming data analysis. Third, if, in the summer, a shopkeeper launches a sale promotion that is based on the discovered rule, such as providing a 10% discount for buying a turkey and a pumpkin pie together, he is bound to fail because the right time to apply the "*Turkey* → *Pumpkin pie*" rule is well known to be during the weeks before Thanksgiving holiday.

If only the transactions in a specific time interval, such as the week before Thanksgiving, are considered, then most will be found to contain both turkey and pumpkin pie. Accordingly, the rule "*turkey* → *pumpkin pie*" has high *support* and high *confidence* in the week before Thanksgiving. From this example, an itemset may not be frequent in the entire dataset but it may be frequent in specific intervals. Therefore, effectively finding these intervals in which patterns are frequent is important.

In intrusion detection systems [3,4], association rule mining is used to identify malicious activity by analyzing

[*] Corresponding author e-mail: dlyang@fcu.edu.tw

network traffic data. In this application, the goal of applying association rules is to find relationships between the various attack signatures and IP addresses that constitute real attacks in the network environment. The generation of more association rules corresponds to more accurate attack detection. To prevent loss of any alarm, as many association rules as possible must be generated. A useful means of so doing is to set the *support* value relatively low, while enforcing a high *confidence* constraint on the result set. However, properly setting these thresholds is critical to successful detection and difficult to do. Most attackers leave tails behind them and their behaviors can be determined by analysis to be related to their malicious activities. One of the important features of the effective analysis is the capture of temporal data. For example, many intrusions occur late at night to reduce the likelihood that people will notice degraded system performance or malfunctions that might be associated with the unusual activity. Other potential temporal considerations include special dates or occasions, such as the release of new products or services. As in the market basket analysis, too many useless rules may waste processing time and too few rules may make detection ineffective. In this study, temporal factors are considered to find more meaningful rules with higher efficiency to improve the effectiveness of the intrusion detection system.

Li *et al*. [5] proposed the calendar-based script to find time intervals. However, devising this script requires domain knowledge. In this study, the intervals of association rules will be automatically generated without the need for any domain knowledge. The proposed GLFMiner (Global and Local Frequent-patterns Miner) algorithm finds frequent patterns in intervals in which the minimum *support* is higher and the likelihood of the application of discovered rules is greater. GLFMiner firstly transforms the transaction dataset into a bit-map representation, and then recursively joins *(k-1)* itemsets to form *k* itemsets in depth-first (DF) order. After joining the itemsets, GLFMiner generates the intervals of frequent itemsets.

The rest of this paper is organized as follows. Section 2 describes the temporal association rule mining problem and reviews related works on mining temporal association rules. Section 3 describes the three stages of GLFMiner algorithm, which are as follows. (1) Read the dataset and transform transactions into the bit-map representation. (2) Localize bit-map representations of all non-global frequent-items to construct an initial Interval-Tree. (3) Recursively join the *(k-1)* nodes of the Interval-Tree to generate *k* nodes and identify local and global frequent-itemsets. Section 4 presents experimental results obtained using real and synthetic datasets and compares the performance of GLFMiner with that of other algorithms. Section 5 discusses the correctness and computational complexity of our proposed algorithm. Finally, Section 6 draws conclusions.

2 Background and Related Works

2.1 Association rule mining

In recent years, an increasing number of researchers [6,7] have worked on association rule mining because it is an important part of data mining. Agrawal *et al*. were the first to consider the problem of association rule mining, which is formally defined as follows.

Let DB be a transaction dataset that contains a set of transactions $\{t_1, t_2, t_3, \cdots, t_n\}$. Let $I = \{i_1, i_2, \cdots, i_m\}$ be a set of items. Let $t = (Tid, t - itemset)$ be a transaction. Tid is a transaction number and t-itemset contains a set of items. Let X be a set of items. A transaction t contains X if and only if $X \subseteq t$. The length of a transaction that contains *k* items is denoted by *k*. The two important measurements in association rule mining are *support* and *confidence*. *Support* is the frequency with which patterns occur in DB and *confidence* is the strength of implication. The definitions are as follows.

(1) $Support(X) = \frac{|T(x)|}{|DB|}$,

(2) $Confidence(X \to Y) = \frac{Support(X \cup Y)}{Support(X)}$,

where DB represents the dataset; $|DB|$ is the number of transactions in DB, and $T(\alpha)$ represents the set of all transactions that contain α in DB.

In a *support-confidence* framework, if $X \to Y$ is an interesting relation, then X and Y must occur frequently. Two conditions define a frequent relation:

$Support(X \cup Y) \geq MinSup$

and

$Confidence(X \to Y) \geq MinConf$

where *MinSup* is the minimum *support* threshold and *MinConf* is the minimum *confidence* threshold.

2.2 Related works

In recent decades, various algorithms have been proposed to discover efficiently frequent itemsets in various applications. These methods include level-wise algorithms [1,8,9,10,11,12] and pattern-growth methods [2,13]. Temporal association rule mining is an extension of association rule generation. Several temporal association rule mining algorithms have been developed for mining more meaningful frequent patterns, temporal association rules, and up-to-date association rules than conventional association rule mining algorithms.

Hong *et al*. [14] studied up-to-date association rules. The main purpose of such a rule concerns an itemset that may not be frequent in an entire dataset, but which may be largely up-to-date because items that rarely occur earlier in the dataset may often occur later. An up-to-date pattern comprises an itemset and its up-to-date lifetime, which must satisfy the user-defined minimum *support* threshold.

Interesting associations with high *confidence*, albeit with small *support*, can be found. Ale and Rossi [15] limited the total transactions to those that belong to the lifetime of the items. Therefore, those associations would be discovered as they would have sufficient *support*. Those authors extended the notion of association rules by incorporating time into the frequent itemsets. Every item in the dataset has a period of lifetime or lifespan that explicitly represents the temporal duration of the information about the item which is the time during which the item is relevant to the user.

Lee *et al*. [16, 17] studied a new problem of mining general temporal association rules from publication datasets. A publication dataset is a set of transactions, where each transaction is a set of items of which each item contains an individual exhibition period. The authors claimed that the current model of association rule mining cannot handle the publication dataset because of the following fundamental problems: (1) lack of consideration of the exhibition period of each individual item and (2) lack of an equitable basis for determining the *support* of the items. The authors proposed the Progressive-Partition-Miner (PPM) algorithm. The basic concept of PPM is firstly to partition the publication dataset in light of the exhibition periods of items, and then progressively accumulates the occurrence count of each 2-itemset candidate based on intrinsic partitioning characteristics. Anjana *et al*. [18] proposed an innovative algorithm that combines the Progressive Partition approach with Counting Inference method (PPCI) to discover association rules in a temporal database.

Abdullah and Susan [19] examined the discovery of association rules in temporal data. They incorporated an enumeration operation into the relational algebra to prepare the data for the discovery of association rules and applied knowledge discovery techniques to a series of datasets over consecutive time intervals, rather than to the entire database. However, their algorithm requires that the interval size to be determined in advance. The contribution of this work is the generation of an interval without predefining the interval size.

Tsai [20] proposed a framework for mining data stream, called the weighted sliding window model. The model allows a user to specify the number of windows for mining, the size of each window, and the weight of each window. The weighted sliding window model emphasizes temporality and gives a greater weight to more recent events.

Gharib *et al*. [21] used the concept of temporal association rules to solve handling time series by incorporating time expressions into their algorithm. Their proposed algorithm is incremental and uses the results of earlier mining to obtain the final mining output. Their proposed algorithm reduces the time required to generate new candidates by storing 2-itemset candidates and includes a means of updating previously generated candidates rather than re-generating them from scratch.

Li *et al*. [5] studied the problem of mining association rules with reference to the time intervals in which an association rule holds. To identify meaningful time intervals, they used calendar schemas and calendar-based patterns. An example of a calendar schema is (year, month, day); a calendar-based pattern within the schema is (*, 3, 15), which represents the set of time intervals each of which is the 15th day in March.

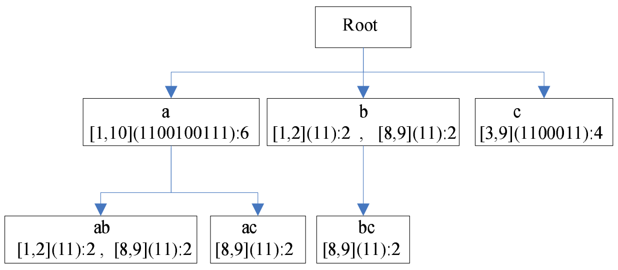

Fig. 1: Interval-Tree

Li *et al*. [5], Gharib *et al*. [21] and Verma *et al*. [22] assumed that transactions are time-stamped, enabling determination of whether a transaction occurs in a particular interval. Other researchers [14, 15, 17, 20] do not take this approach. GLFMiner does not assume that a transaction is time-stamped. Rather, it uses the sequence of transactions as the temporal basis for generating all of the intervals of the itemsets. The main contribution of GLFMiner is that it not only finds global frequent-patterns throughout the dataset but also identify local frequent-patterns in particular intervals.

3 Proposed GLFMiner Algorithm

3.1 Definitions

A frequent pattern is an itemset whose frequency in DB is larger than or equal to *MinSup*. This study proposes the concept of local and global frequent-patterns. GLFMiner performs a search for a frequent pattern set over the search space of a novel Interval-Tree as presented in Fig. 1. The following section will formally define the Interval-Tree. Each node in the Interval-Tree, represented by a datum of the form $< itemset, TidSegmentSet >$, is a prefix-based class. All children, with length *k*, of a given node are in the same class because they all share the same prefix. Therefore, all the children in the same class can be joined to form *(k+1)* frequent patterns. Next, some terms that are used in the proposed algorithm are defined.

Let $I = \{i_1, i_2, i_3, \cdots, i_m\}$ be a set of distinct items. An **itemset** is a subset of I. For example, in Table 1, $I = \{a, b, c, d, e, f\}$. Pattern {a, b} is an itemset that specifies coexistence of items “a” and “b” in the transaction. An **interval** is a set of consecutive transaction IDs and is denoted as [begin, end] where “begin” is the ID of the first transaction in the interval and “end” is the final one. For example, Interval [3, 5] is the range of transactions from Transaction ID 3 to Transaction ID 5.

Table 1: Sample dataset

Tid	Transaction Time	Items
1	2009/07/01	a,b,d,f
2	2009/07/16	a,b,d,f
3	2009/08/03	c,d,f
4	2009/08/09	c,f
5	2009/09/10	a,f
6	2009/09/30	f
7	2009/10/03	e
8	2009/10/18	a,b,c
9	2009/11/08	a,b,c,d,e
10	2009/11/27	a,f

Let α be an itemset. A **TidSegment** records α in a specific interval as [begin, end](bitmap):count where [begin, end] represents an interval. The bitmap of α is a bit string of 1s and 0s, where a 1-bit indicates the corresponding transaction contains α and a 0-bit indicates the corresponding transaction does not contain α. The count of α is the number of the transactions that contain α. For example, in Fig. 1, the TidSegment of the itemset "a", [1,10](1100100111):6, reveals that the itemset "a" appears in the interval [1,10]; the corresponding bitmap is (1100100111), where Transaction ID 1 represents the "begin" ID; Transaction ID 10 represents the "end" ID, and the count of occurrence is 6. The itemset "ab" comprises two TidSegments, [1,2](11):2 and [8,9](11):2, meaning that itemset "ab" appears in two intervals, [1,2] and [8,9], and that the corresponding bitmap is (11), and the frequency of occurrence is 2. A **TidSegmentSet** is the set of TidSegments that belongs to the same itemset. For example, the TidSegment of "ab" [1,2](11):2, and TidSegment of "ab" [8,9](11):2, have the same itemset "ab", and form a TidSegmentSet, which is denoted as {[1,2](11):2,[8,9](11):2}. A **node** comprises an itemset-TidSegmentSet pair, and is denoted as $< itemset, TidSegmentSet >$, where TidSegmentSet is the set of TidSegments that correspond to the itemset. For example, a node, $< ab, \{[1,2](11) : 2, [8,9](11) : 2\} >$, indicates that the itemset is "ab" and its corresponding TidSegmentSet is {[1,2](11):2, [8,9](11):2}.

Definition 1.*Interval-Tree of $\{Node_1, Node_2, \cdots, Node_n\}$ is a prefix-based tree. The root node of the Interval-Tree is null. The other nodes contain itemset-TidSegmentSet pairs. If $Node_n$ is the parent of $Node_m$, $|Node_m| = |Node_n| + 1$, and the itemset of $Node_n$ is the prefix of the itemset of $Node_m$, where $|Node_m|$ is the height of $Node_m$ in the Interval-Tree.*

Figure 1 presents an example of an Interval-Tree. Each node contains an itemset-TidSegmentSet pair. The itemset "a" of $Node_a$ is the prefix of the itemset "ab" of $Node_{ab}$. The itemset "b" of $Node_b$ is the prefix of the itemset "bc" of $Node_{bc}$. $Node_{ab}$ contains a set of two TidSegments.

Definition 2.*An itemset α is called a global frequent-pattern (GFP) in DB if*

(1) $Support(\alpha) = \frac{|T(\alpha)|}{|DB|}$, and

(2) $Support(\alpha) \geq MinSup$

The definition of global frequent-patterns is the same as that of frequent patterns in a conventional association rule mining algorithm. For example, based on $MinSup = 0.5$, item a[1,10] in Table 1 is a globally frequent 1-itemset for the entire dataset, where the first transaction that contains "a" is TID=1 and the final transaction that contains "a" is TID=10. Next, the definition of local frequent-patterns in specific intervals is presented.

Definition 3.*The minimum length of a TidSegment is denoted as MinLen.*

If an interval is less than *MinLen* $\times |DB|$, then this interval is not sufficiently prominent and is therefore deleted. For example, based on the DB of Table 1, $MinLen = 15\%$ is set. If the interval is 1, then the *support* of this interval is 100% and the *confidence* is also 100%. Since 1 is less than $MinLen \times |DB| = 1.5$, this interval can be regarded as noise and deleted.

Definition 4.*An item set α is called a local frequent-pattern (LFP) in a DB$_i$ if*

(1) $Support(\alpha) = \frac{|T_i(\alpha)|}{|DB_i|}$,

(2) $Support(\alpha) \geq MinSup$,

(3) $|T_i(\alpha)| \geq Minlen \times |DB|$ and

(4) α is not a global frequent-pattern, where $T_i(\alpha)$ is the set of all transactions that contains α in the dataset DB in a particular interval i.

Given a *MinSup*, local frequent-patterns are those whose number is larger than *MinSup* only in particular intervals. For example, if $MinSup = 0.5$, then the itemset "e" is a local frequent 1-itemset in the interval *i*=[7,9] because $|T_i(e)| = 2$, $|DB_i| = 3$ and $\frac{|T_i(e)|}{|DB_i|} = 0.67$. However, it is not a global frequent-itemset for $MinSup = 0.5$.

Definition 5.*Two consecutive TidSegments are "mergeable" if and only if the following condition is met; $\frac{TidSegment_n.count+TidSegment_{n-1}.count}{TidSegment_n.end-TidSegment_{n-1}.begin+1} \geq MinSup$, where $TidSegment_{n-1}[begin, end]$: count and $TidSegment_n[begin, end]$: count are two consecutive intervals of α.*

For example, for $TidSegment_{n-1}[4,4](1)$: 1 and $TidSegment_n[7,9](111)$: 3, $\frac{TidSegment_n.count+TidSegment_{n-1}.count}{TidSegment_n.end-TidSegment_{n-1}.begin+1} = \frac{3+1}{9-4+1} = 0.67 \geq MinSup$; therefore, they can be merged. The order of merging is right side first if both the current left interval and the current right interval can be merged.

Definition 6.*The merging of consecutive TidSegments is defined as follows.*

$$newTidSegment = Merge(TidSegment_{n-1}, TidSegment_n)$$

where

$$newTidSegment.begin = TidSegment_{n-1}.begin,$$
$$newTidSegment.end = TidSegment_n.end,$$
$$newTidSegment.count = TidSegment_{n-1}.count + TidSegment_n.count,$$

and the two bitmaps merged are defined as below:

$$newTidSegment.bitmap = TidSegment_{n-1}.bitmap + fill(0, TidSegment_n.begin - TidSegment_{n-1}.end - 1) + TidSegment_n.bitmap.$$

Here, the function "fill" generates a consecutive "0" bitmap, while the second parameter represents the number of "0"s to be generated.

For example, given two intervals, $TidSegment_{n-1}$ $[4,4](1):1$ and $TidSegment_n[7,9](111):3$, $Merge(TidSegment_{n-1}, TidSegment_n)$= $TidSegment_{n-1}[4,9](100111):4$.

Definition 7.*The nodes of itemsets α and β are called node-joinable if and only if both nodes share one parent node in the Interval-Tree.*

Definition 8.*The $TidSegment_n$ in the itemset α and the $TidSegment_m$ in the itemset β are called TidSegment-disjoinable if and only if*
(1) $TidSegment_n.begin > TidSegment_m.end$, or
(2) $TidSegment_m.begin > TidSegment_n.end$.

Definition 9.*The join of TidSegments, $TidSegment_n$ and $TidSegment_m$, is defined as*

$$newTidSegment = Join(TidSegment_n, TidSegment_m),$$

where

$$newTidSegment.begin = Max(T\,idSegment_n.begin, T\,idSegment_m.begin),$$

$$newTidSegment.end = Min(T\,idSegment_n.end, T\,idSegment_m.end),$$

$$newTid\ \ Segment.bitmap = AND(TidSegment_n[newTidSegment.begin, newTidSegment.end].bitmap, TidSegment_m[newTidSegment.begin, newTidSegment.end].bitmap)$$

For example, $newTidSegmentSet = Join\{[1,2](11), [1,5](11001)\} = [1,2](11)$

Definition 10.*The joining of nodes α and β is defined as follows:*

$$< \gamma, TidSegmentSet_\gamma >= Join(< \alpha, TidSegmentSet_\alpha >, < \beta, TidSegmentSet_\beta >)$$

where
$\gamma = \alpha \cup \beta$ as in the traditional definition of join.

$$TidSegmentsofTidSegmentSet_\gamma = Join(TidSegmentsofTidSegmentSet_\alpha, TidSegmentsofTidSegmentSet_\beta).$$

The nodes of α and β are node-joinable; therefore, they have the same parent node in the Interval-Tree, and α and β differ by only one item. If $\alpha = \{I_1, I_2, I_3, .., I_k, I_m\}$ and $\beta = \{I_1, I_2, I_3, .., I_k, I_n\}$, then $\gamma = \{I_1, I_2, I_3, .., I_k, I_m, I_n\}$.

For example,
$< \gamma, TidSegmentSet_\gamma >= Join(< ab, [1,2](11) >, < af, \{[1,5](11001), [8,9](11)\} >) = < abf, [1,2](11) >$, where

$TidSegment\gamma_1$=$Join([1,2](11), [1,5](11001))$ $= [1,2](11)$,

$TidSegment\gamma_2$=Join([1,2](11), [8,9](11)) $\rightarrow$ TidSegment-disjoinable.

Table 2 lists the possible results of joining two patterns. The joining of two GFPs yields three possible results, which are GFP, LFP, and IFP. Here GFP means the global frequent-pattern, LFP means the local frequent-pattern and IFP means the infrequent pattern.

Table 2: The possible results of joining two patterns

Pattern 1	Pattern 2	Join(Patttern1, Pattern2)
GFP	GFP	GFP, LFP, IFP
GFP	LFP	LFP, IFP
LFP	LFP	LFP, IFP

3.2 GLFMiner Algorithm

The proposed algorithm has three main steps. First, it scans the dataset to find the global frequency of each item and converts all items into a bit-map to speed up processing. Second, it localizes each item that is not globally frequent and constructs the initial Interval-Tree. Third, it recursively joins each node in DF order on the Interval-Tree. Then, it localizes the joined TidSegment to find the frequent intervals and adds them to the Interval-Tree. The frequent nodes are output as temporal frequent-itemsets. Figure 2 presents the pseudo code of

```
Main_Program of GLFMiner
Input: (1) DB (2) MinSup (3) MinLen
Output:(1) Global and Local Frequent-patterns
Begin
1)  BitMaps=ScanDataBase(DB)
2)  Interval_Tree=Construct_Initial_Interval_Tree(BitMaps, MinSup, MinLen)
3)  FreqSet = RecursiveJoin(Interval_Tree)
End
```

Fig. 2: Main program of GLFMiner

```
Function Construct_Initial_Interval_Tree
Input:  (1) BitMaps (2) MinSup (3) MinLen
Output: (1) Interval_Tree
Begin
1)   Interval_Tree ← Create Root Node
2)   Candidate_Items ← getItemFrequency (BitMaps)
3)   for (each item in Candidate_Items)
4)     Begin
5)        newNode = new Node(item)
6)        if newNode is not GlobalFrequent
7)           LocalizeNode(newNode, MinSup, MinLen)
8)           if newNode is LocalFrequent
9)             Interval_Tree.addChild(newNode)
10)       else
11)            Interval_Tree.addChild(newNode)
12)    End
13)  Return Interval_Tree
```

Fig. 3: Pseudo code for constructing initial Interval-Tree

```
a(1100100111), b(1100000110), c(0011000110), d(1110000010), e(0000001010),
f(1111110001)
```

Fig. 4: Bitmap representation of every item in Table 1

the GLFMiner algorithm. The following section will elucidate in detail the steps of the GLFMiner algorithm.

Step 1: GLFMiner scans the dataset and transforms all items into a bit-map representation. The dataset in Table 1 with $MinSup = 0.5$ is used to generate the bitmap representation of every item in Fig. 4.

Step 2: GLFMiner verifies whether each item is globally frequent. If it is not, then the Localize function is used to determine local frequent-intervals. GLFMiner adds global and local frequent-items into the Interval-Tree to construct an initial Interval-Tree.

In this step, GLFMiner localizes the nodes that contain TidSegments that are not globally frequent. Localizing a TidSegment comprises three steps. First, the bitmap of a TidSegment is partitioned to find the local frequent-intervals of the corresponding itemset. Second, the algorithm checks whether the two consecutive TidSegments can be merged to form a larger local frequent-interval. Third, GLFMiner prunes the TidSegments whose intervals are shorter than $MinLen \times |DB|$. For example, the bitmap representation of item "a" is (1100100111). The number of item "a" is 6, which is larger than $MinSup \times |DB| = 0.5 \times 10 = 5$; therefore, "a" is a global frequent-itemset. The bitmap representation of item "c" is (0011000110). The number of item "c" is 4, which is below the *MinSup* threshold.

```
Function RecursiveJoin
Input:  (1) ParentNode
Output: (1) FreqSet
Begin
1)for (each node$_1$ ∈ ParentNode ) {
2)  for (each node$_2$ ∈ Right Siblings of node$_1$ )
3)  {
4)    SonNode = Join(node$_1$, node$_2$)
5)    If SonNode is not GlobalFrequent
6)      LocalizeNode (SonNode, MinSup, MinLen)
7)    Else
8)      node$_1$.addChild (SonNode)
9)  }
10)  FreqSet=FreqSet ∪ node$_1$
11)  FreqSet=FreqSet ∪ RecursiveJoin (node$_1$)
12)  ParentNode.RemoveChild (node$_1$) }
End
```

Fig. 5: Pseudo code of RecursiveJoin

Therefore, "c" is not a global frequent-itemset and must be localized.

Step 2-1: Partition the bitmap representation of a TidSegment.

For example, itemset "c" is not a global frequent-itemset. Table 3 presents the procedure for partitioning itemset "c" with $MinSup = 0.5$.

Table 3: Procedure for partitioning the TidSegment of Itemset "c"

N	bit	Interval	*Support*	Action
10	0			Skip
9	1	[9,9]:1	$(1/0.5 = 2) >= (9 - 9 + 1)$	
8	1	[8,9]:2	$(2/0.5 = 4) >= (9 - 8 + 1)$	
7	0	[8,9]:2	$(2/0.5 = 4) >= (9 - 7 + 1)$	
6	0	[8,9]:2	$(2/0.5 = 4) >= (9 - 6 + 1)$	
5	0	[8,9]:2	$(2/0.5 = 4) < (9 - 5 + 1)$	
			Add A New TidSegment [8,9](11):2	
4	1	[4,4]:1	$(1/0.5 = 2) >= (4 - 4 + 1)$	
3	1	[3,4]:2	$(2/0.5 = 4) >= (4 - 3 + 1)$	
2	0	[3,4]:2	$(2/0.5 = 4) >= (4 - 2 + 1)$	
1	0	[3,4]:2	$(2/0.5 = 4) >= (4 - 1 + 1)$	
			Add A New TidSegment [3,4](11):2	

Step 2-2: Merge pairs of consecutive intervals to form a larger interval from the final interval to the first.

For example, consider c_1[3,4](11):2 and c_2[8,9](11):2; since $\frac{(c_1.count+c_2.count)}{(c_2.end-c_1.begin+1)} = \frac{(2+2)}{(9-3+1)} = \frac{4}{7} > MinSup$, c_1[3,4](11):2 and c_2[8,9](11):2 are merged into c_1[3,9](100111):4.

Step 2-3: Prune the TidSegment whose interval is shorter than $MinLen \times |DB|$ to eliminate trivial intervals.

For example, if $MinLen = 15\%$ and the length of TidSegment d_2 [9,9]:1 is 1, which is less than $MinLen \times |DB| = 15\% \times 10 = 1.5$, the TidSegment d_2 [9,9]:1 is removed.

Step 1	Bitmap representation of every item in Table 1	
	a(1100100111):6, b(1100000110):4, c(0011000110):3, d(1110000010):4, e(0000001010):2, f(1111110001):7	
Step 2-1	Partition	<a,[1,10](1100100111):6>, <b,{[1,2](11):2, [8,9](11):2}>,<c,{[3,4](11):2, [8,9](11):2}>, <d,{[1,3](111):3, [9,9](1):1}>, <e,[7,9](101):2>, <f,[1,10](1110110101):7>
Step 2-2	Merge	<a,[1,10](1100100111):6>, <b,{[1,2](11):2, [8,9](11):2}>, <c,[3,9](1100011):4>, <d,{[1,3](111):3, [9,9](1):1}>, <e,[7,9](101):2>, < f,[1,10](1111110001):7}>
Step 2-3	Prune	<a,[1,10](1100100111):6>, <b,{[1,2](11):2, [8,9](11):2}>, <c,[3,9](1100011):4>, <d,[1,3](111):3>, <e,[7,9](101):2>, <f,[1,10](1111110001):7>
Step 3-1	Join(<a,[1,10](1100100111),*)	<ab,{[1,2](11):2, [8,9](11):2}>, <ac,[8,9](11):2>, <ad,[1,2](11):2>, <af,[1,10](1100100001):4>
Step 3-2	Localize	<ab,{[1,2](11):2, [8,9](11):2}>, <ac,[8,9](11):2>, <ad,[1,2](11):2>, <af,[1,5](11001):3>
Step 3-3	Add node	<ab,{[1,2](11):2, [8,9](11):2}>, <ac,[8,9](11):2>, <ad,[1,2](11):2>, <af,[1,5](11001):3>
Step 3-4	Add to Frequent set	<ab,{[1,2](11):2, [8,9](11):2}>, <ac,[8,9](11):2>, <ad,[1,2](11):2>, <af,[1,5](11001):3>
Step 3-1	Join(<ab,{[1,2](11),[8,9](11)}>,*)	<abc,[8,9](11):2>, <abd,[1,2](11):2>, <abf,[1,2](11):2>
Step 3-2	Localize	<abc,[8,9](11):2>, <abd,[1,2](11):2>, <abf,[1,2](11):2>
Step 3-3	Add node	<abc,[8,9](11):2>, <abd,[1,2](11):2>, <abf,[1,2](11):2>
Step 3-4	Add to Frequent set	<abc,[8,9](11):2>, <abd,[1,2](11):2>, <abf,[1,2](11):2>
Step 3-1	Join(<abc,[8,9](11)>,*)	

Fig. 6: Deployment of GLFMiner

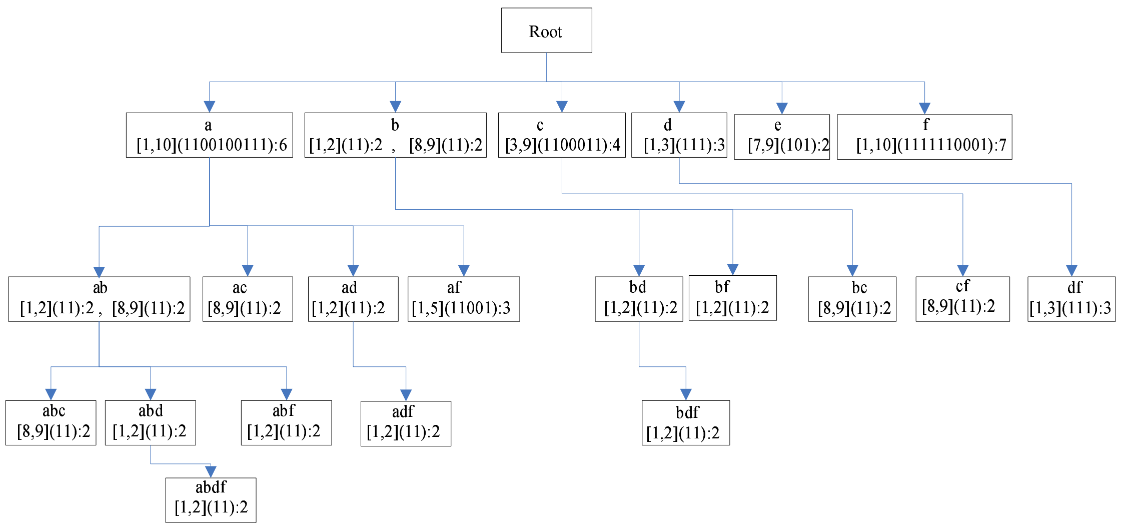

Fig. 7: Complete Interval-Tree based on Table 1

Step 3: Recursively join the itemsets in DF order from the root. Figure 5 presents the pseudo code. The node of length *(k-1)* joins all of the right siblings to generate a combined itemset of length *k*. Every joining procedure comprises the following steps.

Step 3-1: $Node_1$ joins $Node_2$ (right sibling).

For example, $SonNode = Join(< a, [1, 10](1100100111) : 6 > , < b, \{[1, 2](11) : 2, [8, 9](11) : 2\} >) = < ab, \{[1, 2](11) : 2, [8, 9](11) : 2\} >$

Step 3-2: Localize SonNode.

Step 3-3: If SonNode is frequent, add it to $Node_1$ as a child.

Step 3-4: Add $Node_1$ to the frequent itemset, FreqSet.

3.3 A complete Example

The dataset in Table 1 is used with $MinSup = 0.5$ and $MinLen = 15\%$ to demonstrate the procedure of the GLFMiner algorithm. The three steps of the GLFMiner algorithm are as follows.

```
Step 1: Transform each item in DB into a bitmap representation.
Step 2: Perform Localize function for non-global frequent-items to construct an initial Interval-Tree.
Repeat
    Step 3-1: SonNode = $Node_1$ joins $Node_2$
    Step 3-2: Localize SonNode
    Step 3-3: If SonNode is frequent, then add it to $Node_1$ as a child
    Step 3-4: $AddNode_1$ to the frequent itemset, FreqSet
```

Fig. 6 presents the procedures for deploying GLFMiner to mine the dataset in Table 1. The procedure for constructing the leftmost branch of the Interval-Tree is provided. Fig. 7 presents the complete Interval-Tree.

4 Experimental Results

This section describes in detail the experiments of the GLFMiner algorithm. The algorithm was implemented in JAVA on personal computer with an Intel Core Duo Processor at 1.99 GHz with 4 GB RAM. The synthetic dataset of T10I4D100K from the IBM dataset generator and the real dataset Chess were used to validate effectiveness of the proposed algorithm. In T10I4D100K, the size of a transaction is T=10, the size of a potential maximal frequent itemset is I=4, and the total number of transactions is D=100000. Multiple sets of such dataset are generated to perform the experiments. The resultant values are averaged to yield the final outcome.

Of the most popular association rule mining algorithms that have been used in recent decades [1,2], Apriori is still used extensively and has many extensions under study. The performance of GLFMiner is compared with that of Apriori although this classical method does not consider temporal intervals.

In the first experiment, the relation between execution time and *MinSup* settings is determined by varying the *MinSup* thresholds. Figure 8 presents the results obtained using the T10I4D100K dataset where $MinLen = 60\%$ and $MinSup = 0.09$ to 0.02. Figure 9 displays the results obtained using the Chess dataset with $MinLen = 60\%$ and $MinSup = 0.9$ to 0.7. In both experiments, the execution time increased as the *MinSup* decreased, as it did for Apriori. Lower *MinSup* thresholds yield more temporal interval patterns, especially when they are below 0.04 and 0.8 for T10I4D100K and Chess respectively. Therefore, *MinSup* is one of the main factors that influence the execution time. Another important factor is the total number of transactions.

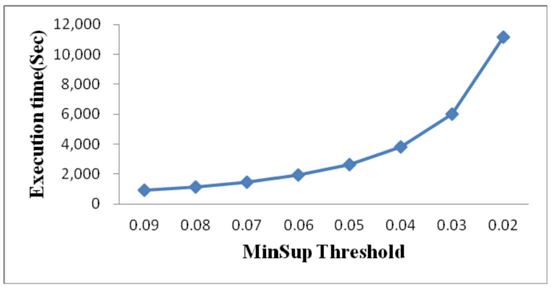

Fig. 8: Relationship between execution time and *MinSup* for T10I4D100K dataset

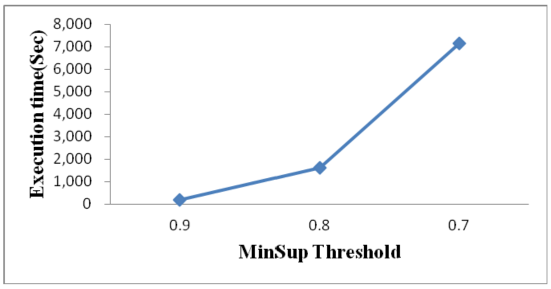

Fig. 9: Relationship between execution time and *MinSup* for Chess dataset

Although GLFMiner exhibits a similar relationship between execution time and *MinSup* to that of Apriori, it performs well in finding more frequent patterns when *MinSup* is low than Apriori. This effect is significant since most local frequent-patterns are found with a smaller *MinSup*. Figures 10 and 11 present the relationships between the number of frequent patterns and the *MinSup* thresholds. Figure 10 shows the result obtained for T10I4D100K with $MinLen = 1\%$ and $MinSup = 0.1$ to 0.9. Figure 11 shows the result obtained for Chess with $MinLen = 60\%$ and $MinSup = 0.7$ to 0.9. Clearly, for a given *MinSup*, GLFMiner discovers more frequent patterns than does the conventional method Apriori, for both datasets T10I4D100K and Chess. The experiments herein reveal that GLFMiner finds more potential association rules than Apriori. In particular, GLFMiner outperforms Apriori by a factor of seven to fifteen when *MinSup* is less than 0.1 and 0.7 in Fig. 10 (T10I4D100K) and Fig. 11 (Chess) respectively.

To confirm that GLFMiner performs well in finding local frequent-patterns, the following experiment elucidates the relationship between the number of local frequent-patterns and the *MinSup* threshold in Fig. 12. A smaller *MinSup* yields more local frequent-patterns. At this point, the consistency between the mined local frequent-patterns and the distribution of the generated transactions' pattern length must be checked. Figure 13 plots the number of local frequent-patterns of length one

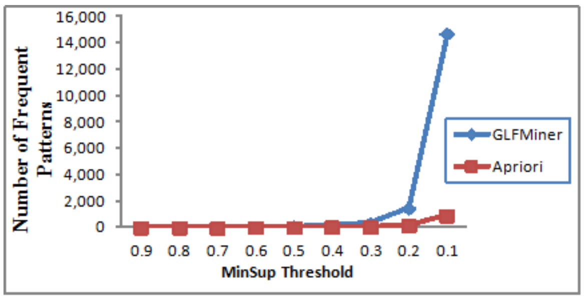

Fig. 10: Comparison of the number of frequent patterns discovered by GLFMiner and Apriori using T10I4D100K dataset for *MinSup* from 0.1 to 0.9

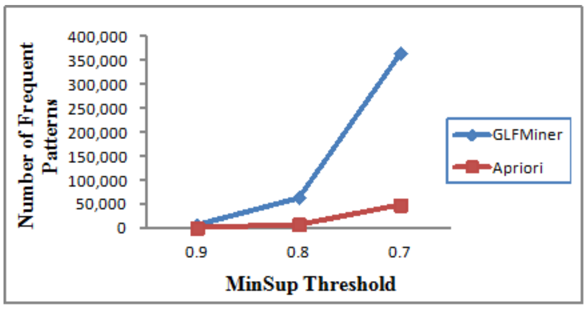

Fig. 11: Comparison of the number of frequent patterns discovered by GLFMiner and Apriori using Chess dataset for *MinSup* from 0.7 to 0.9

to seven for T10I4D100K with $MinSup = 0.6$ and $MinLen = 10\%$. The local frequent 4-patterns are found in the greatest number since the size of potential maximal frequent itemset was set to I=4.

When the length of a time interval is less than *MinLen* $\times |DB|$ number of frequent patterns that are found by GLFMiner is compared with that found by Apriori for T10I4D100K with $MinSup = 0.4$ and $MinLen = 0.1\%$ to 0.9%. As shown in Fig. 14, GLFMiner always finds more frequent patterns than does Apriori, especially when the *MinLen* threshold is small. When the GLFMiner algorithm is used, higher *MinSup* thresholds can be set, and a suitable *MinLen* can be used to mine valuable frequent patterns.

Lastly, we compare the GLFMiner algorithm with Up-To-Date [14] and Apriori algorithms by using the same T10I4D100K dataset. Using *MinLen*=0.1% (purple color) and *MinLen*=0.2% (green color) for GLFMiner, the relationships between the number of frequent 1-itemsets for different *MinSup* thresholds are shown in Figure 15. It is clear that the number of frequent-itemsets discovered by the GLFMiner algorithm for LFP and GFP was larger than that of Up-To-Date and Apriori algorithms. The relationships between the number of frequent 2-itemsets and 3-itmesets for different *MinSup* thresholds are shown

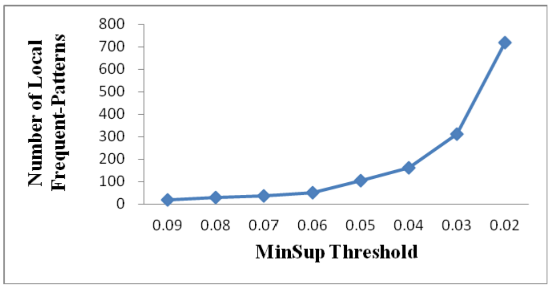

Fig. 12: Relationship between the number of local frequent-patterns and various *MinSup* thresholds for T10I4D100K dataset

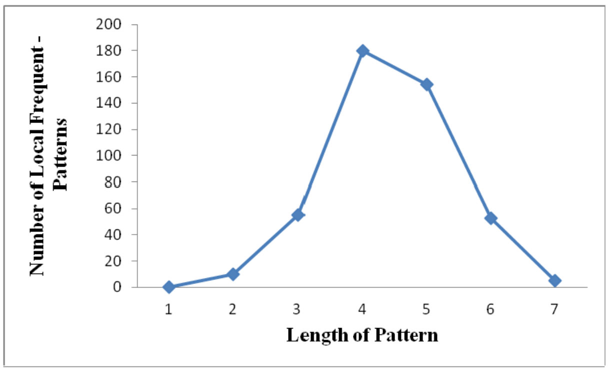

Fig. 13: Relationship between the number of local frequent-patterns and the length of patterns for T10I4D100K dataset

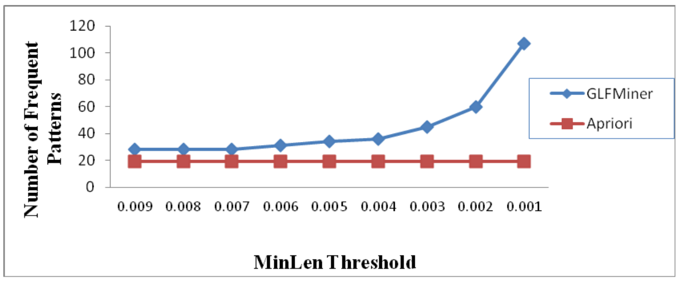

Fig. 14: Comparison of the number of frequent patterns discovered by GLFMiner and Apriori for various *MinLen* thresholds

in Figures 16 and 17, respectively. It can be observed that the number of the frequent 2-itemsets and 3-itemsets using Apriori algorithm are close to zero when the support thresholds were set at 1% and above. By the GLFMiner algorithm for GFP and LFP, the number of frequent-itemsets discovered are much more than that of the Up-To-Date algorithm (and the same for Apriori).

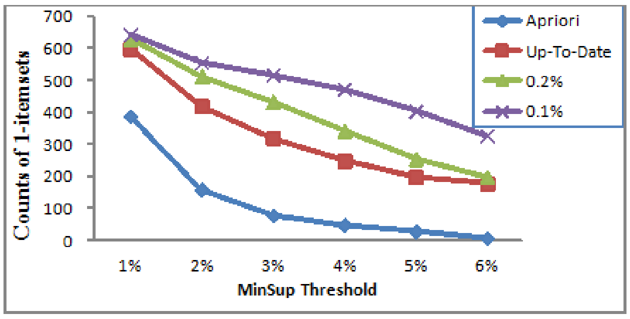

Fig. 15: Comparison of the number of frequent 1-itemsets discovered by Apriori, Up-To-Date and GLFMiner algorithms

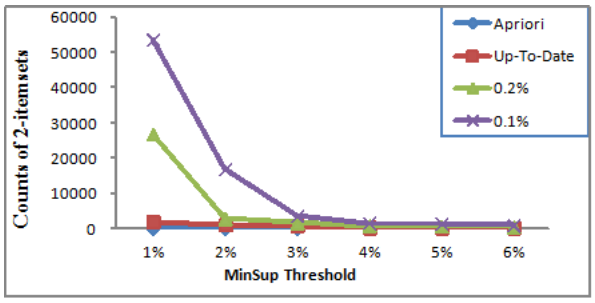

Fig. 16: Comparison of the number of frequent 2-itemsets discovered by Apriori, Up-To-Date and GLFMiner algorithms

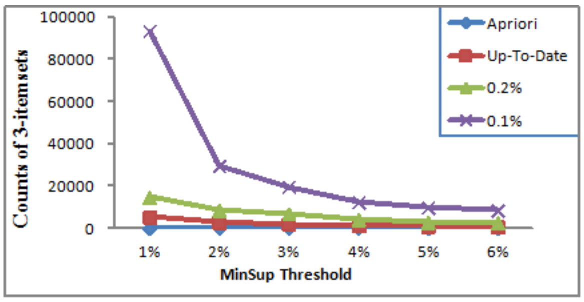

Fig. 17: Comparison of the number of frequent 3-itemsets discovered by Apriori, Up-To-Date and GLFMiner algorithms

5 Discussion

In this section, we prove the correctness of our GLFMiner and analyze its computational complexities.

5.1 Proof of correctness for GLFMiner

When the *MinLen* threshold is less than 100%, the GFPs are kept in every node to form TidSegmentsets with LFPs. By Definitions 9 and 10, the nodes are merged in the depth-first order. By using the result of Table 2, a GFP joins with a GFP may generate another GFP. The joined GFPs are the same as the GFPs generated from traditional association rule mining algorithms. In other cases of joining a GFP with a GFP or a GFP with a LFP or a LFP with a LFP, they all follow Definition 9 to generate LFPs. Therefore, GLFMiner can generate GFPs and LFPs correctly.

When the *MinLen* threshold is equal to 100%, by Definition 4, there will be no LFP. The frequent-patterns we get from GLFMiner are the same as the GFPs generated from tradition association rule mining algorithms. End of the proof.

5.2 Computational Complexity Analysis

Let *m* and *n* denote the size of a database and the number of items in the database respectively. According to the main program of GLFMiner in Fig. 2, we discuss the computational complexity for each of the three steps and then sum them up to get the total complexity.

After reading a record, GLFMiner has to map each item to *m* buckets. Therefore, the time complexity of first step is O($m \times n$). In the second step, GLFMiner processes the bitmaps of *n* items and every bitmap has to be further partitioned if it is not a globally frequent. Because the length of each bitmap is *m*, the time complexity of second step is O($m \times n$).

In the third step, GLFMiner joins the bitmaps of node with its sibling nodes in depth-first order. Because every node contains a TidSegmentSet, the join of two nodes consists of lots of joins of TidSegments. The time cost of the third step comes from the cost of joins. Every node has to join with its right siblings except leaf-nodes. The time complexity of the number of joins can be expressed as

$$O(\sum_{i=1}^{NL}(NTS(N_i) \times \sum_{j=1}^{NS_i} NTS(RS_j(N_i))))$$

NL represents the number of non-leaf nodes in an Interval-Tree. The non-leaf nodes are numbered from 1 to *NL* and N_i represents the *i*-th non-leaf node. *NTS* is the function to get the number of TidSegments in node N_i. RS_j is the function to get the *j*-th right sibling node. NS_i represents the number of the right siblings of N_i.

6 Conclusions

This work proposed the concept of global and local frequent-patterns and demonstrated the efficient implementation of the GLFMiner algorithm to discover patterns in temporal datasets. Market basket analysis and intrusion detection are considered as two illustrative examples of the use of the proposed algorithm. Experimental results show that the GLFMiner algorithm can mine more interesting patterns than conventional mining methods for given *MinSup* thresholds. The value of the proposed GLFMiner algorithm is as follows.

–The results mined using GLFMiner include all of the rules that can be mined by using conventional association rule algorithms.
–GLFMiner can discover association rules that are frequent in some intervals but not throughout the entire dataset.
–The mined association rules include time attributes that make them more useful.
–Valuable rules can still be mined by using larger *MinSup* thresholds.
–GLFMiner automatically generates the intervals of frequent patterns.

In the future, appropriate data structures may be used to further improve the execution time of GLFMiner and more real-world datasets will be used to confirm its effectiveness.

Acknowledgement

This research was supported by the National Science Council, Taiwan, under grant NSC 101-2221-E-035-082.

References

[1] R. Agrawal and R. Srikan, Fast Algorithms for Mining Association Rules, Proceedings of the 20th International Conference on Very Large Data Bases, 487-499 (1994).
[2] J. Han , J. Pei and Y. Yin, Mining Frequent Patterns without Candidate Generation, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 1-12 (2000).
[3] V. Katkar and R. Mathew, One pass incremental association rule detection algorithm for network intrusion detection system, International Journal of Engineering Science and Technology, **3**, 3055-3060 (2011).
[4] J. Treinen and R. Thurimella, A framework for the application of association rule mining in large intrusion detection infrastructures, Data and Knowledge Engineering, **44**, 1-18 (2006).
[5] Y. Li , N. Peng, X. Wang and J. Sushil, Discovering calendar-based temporal association rules, Data and Knowledge Engineering, **44**, 193-218 (2003).
[6] Y. Xiao, R. Zhang and I. Kaku, A New Framework of Mining Association Rules with Time-Windows on Real-Time Transaction Database, International Journal of Innovative Computing, Information and Control, **7**, 3239-3253 (2011).
[7] Z. Zhou, Efficiently Mining Positive Correlation Rules, Applied Mathematics & Information Sciences, **5**, 39-44 (2011).
[8] J. Dong and M. Han, BitTableFi: An efficient mining frequent itemsets algorithm, Knowledge-Based Systems, **20**, 329-335 (2007).
[9] M. Hung, S. Weng, J. Wu and D. Yang, Efficient Mining of Association Rules Using Merged Transactions Approach, WSEAS Transactions on Computers, **5**, 916-923 (2006).
[10] H. Lee, On-line association rules mining with dynamic support, Knowledge-Based Intelligent Information and Engineering Systems, Lecture Notes in Computer Science/LANI, **4252**, 896-901 (2006).
[11] W. Song, B. Yang and Z. Xu, Index-BitTableFi: An improved algorithm for mining frequent itemsets, Knowledge-Based Systems, **21**, 507-513 (2008).
[12] Y. Tsay and J. Chiang, CBAR: An efficient method for mining association rules, Knowledge-Based Systems, **18**, 99-105 (2005).
[13] M. Song and S. Rajasekaran, A Transaction Mapping Algorithm for Frequent Itemsets Mining, IEEE Transaction on Knowledge and Data Engineering, **18**, 472-481 (2006).
[14] T. Hong, Y. Wu and S. Wang, An effective mining approach for up-to-date patterns, Expert systems with applications, **36**, 9747-9752 (2009).
[15] J. Ale and G. Rossi, An approach to discovering temporal association rules, Proceedings of the 2000 ACM Symposium on Applied computing, **1**, 294-300 (2000).
[16] C. Lee, M. Chen and Cheng-Ru Lin, Progressive partition miner: an efficient algorithm for mining general temporal association rules, IEEE Transaction on Knowledge and Data Engineering, **15**, 1004-1017 (2003).
[17] C. Lee, C. Lin and M. Chen, On mining general temporal association rules in a publication dataset, Proceedings of the 2001 IEEE International Conference on Data Mining, 337-344 (2001).
[18] A. Pandey and K. Pardasani, PPCI algorithm for mining temporal association rules in large databases, In International Journal of Information & Knowledge Management (JIKM), 345-352 (2009).
[19] A. Tansel and S. Imberman, Discovery of Association Rules in Temporal Databases, In Proceedings of the 4th International Conference on Information Technology (ITNG'07), Las Vegas, Nevada, USA, 371-376 (2007).
[20] P. Tsai, Mining frequent itemsets in data streams using the weighted sliding window model, Expert Systems with Applications, **36**, 11617-11625 (2009).
[21] T. Gharib, H. Nassar, M. Taha and A. Abraham, An efficient algorithm for incremental mining of temporal association rules, Data and Knowledge Engineering, **69**, 800-815 (2010).
[22] K. Verma, O. Vyas, and R. Vyas, Temporal approach to association rule mining using T-tree and P-tree, Lecture Notes in Computer Science, **3587**, 651-659 (2005).

Kuo-Cheng Yin received the B.E. degree and M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan, in 1990 and 1992 respectively. He is now a Ph. D. candidate in the Department of Information Engineering and Computer Science at Feng Chia University. His research interests include data mining and software engineering.

Yu-Lung Hsieh received the M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan, in 2005. He is now a Ph. D. candidate in the Department of Information Engineering and Computer Science at Feng Chia University. His research interests include data mining and genetic algorithm.

Don-Lin Yang received the B.E. degree in Computer Science from Feng Chia University, Taiwan in 1973, the M.S. degree in Applied Science from the College of William and Mary in 1979, and the Ph.D. degree in Computer Science from the University of Virginia in 1985. He was a staff programmer at IBM Santa Teresa Laboratory from 1985 to 1987 and a member of the technical staff at AT & T Bell Laboratories from 1987 to 1991. Since then, he joined the faculty of Feng Chia University, where he was in charge of the University Computer Center from 1993 to 1997 and 2003 to 2006. Dr. Yang served as the Chairperson of the Department of Information Engineering and Computer Science from 2001 to 2003 and is currently a professor at Feng Chia University. His research interests include database system and data mining, image processing, distributed and parallel computing, Web service and mobile applications. He is a member of the IEEE computer society and the ACM.

Ming-Chuan Hung received the B.E. degree in Industrial Engineering and the M.S. degree in Automatic Control Engineering from Feng Chia University, Taiwan, in 1979 and 1985 respectively, and the Ph.D. degree from the Department of Information Engineering and Computer Science at Feng Chia University in 2006. From 1985 to 1987, he was an instructor in the Mechanics Engineering Department at National Chin-Yi Institute of Technology. Since 1987, he has been an instructor in the Industrial Engineering Department at Feng Chia University and served as a secretary in the College of Engineering from 1991 to 1996. Dr. Hung is currently an Associate Professor. His research interests include data mining, CIM, and e-commerce applications. He is a member of the CIIE.