DISCOVERY TEMPORAL FREQUENT PATTERNS USING TFP-TREE

Long Jin, Yongmi Lee, Sungbo Seo, Keun Ho Ryu

Dept. of Computer Science, Chungbuk National University
12, Gaesin-dong, Heungdeok-gu, Chungbuk 361-763, Korea
{kimlyong, ymlee, sbseo, khryu}@dblab.chungbuk.ac.kr

ABSTRACT:

Mining frequent patterns in transaction databases, time-series databases, and many other kinds of databases has been studied popularly in data mining research. Most of the previous studies adopt an Apriori-like candidate set generation-and-test approach. However, candidate set generation is still costly, especially when there exist prolific patterns and/or long patterns. And calendar based on temporal association rules proposes the discovery of association rules along with their temporal patterns in terms of calendar schemas, but this approach is also adopt an Apriori-like candidate set generation. In this paper, we propose an efficient temporal frequent pattern mining using TFP-tree (Temporal Frequent Pattern tree). This approach has three advantages: (1) this method separates many partitions by according to maximum size domain and only scans the transaction once for reducing the I/O cost. (2) This method maintains all of transactions using FP-trees. (3) We only have the FP-trees of 1-star pattern and other star pattern nodes only link them step by step for efficient mining and the saving memory. Our performance study shows that the TFP-tree is efficient and scalable for mining, and is about an order of magnitude faster than the Apriori algorithm and also faster than calendar based on temporal frequent pattern mining methods.

KEY WORDS: Knowledge discovery, Temporal data Mining, Association rule

1. INTRODUCTION

Data mining technique has already been an important research area in computer science. There are many organizations developing in these areas. One of the most popular techniques is the association rule mining (ARM), which is the automatic discovery of pairs of element sets that tend to appear together in a common context.

Time is an important aspect of all real world phenomena. Any systems, approaches or techniques that are concerned with information need to take into account the temporal aspect of data. Temporal data are now being viewed as invaluable assets from which hidden knowledge can be derived, so as to help understand the past and plan for the future [1, 2]. Calendar association rules has been defined by combining the association rules with temporal features which represented by calendar time expression [3]. In this paper, for solving out the mining problem of temporal association rules, a framework for temporal data mining has been proposed. Also, this paper suggested a SQL-like mining language and shows architecture. However, the proposed method is just one kind of temporal pattern. According to [4] logic the technique first search the association ship than it is used to incorporate temporal semantics. It can be used in point based & interval based model of time simultaneously [4]. In calendar based on temporal association rules [5], proposed the discovery of association rules along with their temporal patterns in terms of calendar schemas and identified two classes of rules, precise match and fuzzy match, to represent regular association rules along with their temporal patterns. An important feature of this paper representation mechanism is that the corresponding data mining problem requires less prior knowledge than the prior methods. A Frequent pattern approach for mining the time sensitive data was introduced in [6]. Here the pattern frequency history under a tilted-time window framework in order to answer time-sensitive queries. A collection of item patterns along with their frequency histories are compressed and stored using a tree structure similar to FP-tree and updated incrementally with incoming transactions [6]. This approach is also adopt an Apriori-like candidate set generation.

In this paper, we propose an efficient temporal frequent pattern mining using TFP-tree (Temporal Frequent Pattern tree). This approach has three advantages. First, this method separates many partitions by according to maximum size domain and only scans the transaction once for reducing the I/O cost. Second, this method maintains the all transactions using FP-trees. Third, we only have the FP-trees of 1-star pattern and other star pattern nodes only link them step by step for efficient mining and the saving memory.

The remaining of the paper is organized as follows. Section 2 introduces the TFP-tree structure and its construction method. Section 3 develops a TFP-tree-based frequent pattern mining algorithm, TFP-tree Mining. Section 4 presents our performance study. Section 5 summarizes our study.

2. FREQUENT PATTERN TREE: DESIGN AND CONSTRUCTION

Let $I=\{a_1,a_2,\ldots,a_m\}$ be a set of items, and a transaction database $DB=<T_1,T_2,\ldots,T_n>$, where $T_i(i \in [1..n])$ is a transaction which contains a set of items in I. The support (or occurrence frequency) of a pattern A, which is a set of items, is the number of transactions containing A in DB. A, is a frequent pattern if A's support is no less than a predefined minimum support threshold, ε.

Given a transaction database DB and a minimum support threshold, ε, the problem of finding the complete set of frequent patterns is called the frequent pattern mining problem.

A *calendar schema* is a relational schema (in the sense of relational databases) $R=(f_n:D_n,f_{n-1}:D_{n-1},\ldots,f_1:D_1)$ together with a *valid* constraint (explained below). Each attribute f_i is a calendar unit name like year, month, and week etc. Each domain D_i is a finite subset of the positive integers. The constraint *valid* is a Boolean function on $D_n x D_{n-1} x \cdots x D_1$ specifying which combinations of the values in $D_n x \cdots x D_1$ are "valid".

2.1 Frequent Pattern Tree

To design a compact data structure for efficient temporal frequent pattern mining, we define TFP-tree as follows.

Definition 1 (TFP-tree) A temporal frequent pattern tree (or TFP-tree in short) is a tree structure defined below.

1. It consists of one root labelled as "null", an array of inner node link point *link_list[n]*, and a all start node link point *n-link*.
2. It has an all star node that consists of one *label*, an inner node link point *child*, an inner frequent pattern link point *list_link*, and a frequent pattern list *freqnt_list*.
3. The fields of inner node are one *label*, a *child*, and inner pattern link list *list_link*. And inner pattern is a set of a pattern name *pattern_name*, an inner_pattern link list *pattern_list*, and a *freqnt_list*.
4. The leaf node consists of one *label* and transaction pattern link list *list_link*. And transaction pattern is a set of a *pattern_name*, a transaction list *tran_list*, a FP-tree *FP_T*, and a *freqnt_list*.

2.2 Construction of Frequent Pattern Tree

Based on definition 1, we have the following TFP-tree construction algorithm.

Algorithm 1 (TFP-tree construction)

Input: A transaction database *DB* and calendar schema *R*.
Output: Its frequent pattern tree, *TFP-tree*
Method: The *FP-tree* is constructed in the following steps.

1. Select maximum size domain *d* from the calendar schema *R*.
2. Sort and scan the transaction database *DB* according to *d*.
3. Create the root of a TFP tree *T*, and label it as "null." *T* has n of link points *link_list[n]* and a n-link point *n-link*.
4. Create a parameter *max_domain* and initialize 1;

For each transaction *Trans* in *DB* **do** {
  Filter the maximum domain schema *tem_max* from the basic time e_0 of *Trans*.
  If *max_domain* is equal to *tem_max* **then**
    insert_tree(Trans,T);
  Else
    refresh_tree(max_domain,T);
    max_domain = tem_max;
    insert_tree(Trans,T);
}

Scanning the transaction, we sort the transaction according of maximum size domain. Because of this reason is that firstly we can separate many partitions and secondly save memory.

The procedure of *insert_tree(Trans,T)* is a function that insert each transactions into the transaction list of the containing the basic time in leaf node.

insert_tree(*Trans, T*) {
1. **If** *T* is not any child **Then** {
    initial_tree(T);
  }
2. Filter basic time e_0 from *Trans*.
3. Insert *Trans* into *T* as follows.
  For each 1-star pattern node *N* in *T* **do** {
    If the star pattern list *lis_likt* in *N* does not contain the e_0 **then**
      Create a trans_pattern *temp_list* that contains the e_0 and label *temp_list.pattern_name*.
      Insert the *Trans* into *temp_list.tran_list*.
      list_link link *temp_list*.
      If 2-star pattern of this parent node does not already contain *temp_list*, **then**
        temp_list be linked to *pattern_list* of the contained 2-star pattern.
      Else
        Create the 2-star pattern list and link the *temp_list*.
      Like this, until all star pattern list contain pattern list step by step.
    Else
      Select the trans_pattern *temp_list* that contains e_0. .
      Insert the *Trans* into *temp_list.tran_list*.
  }
}

The procedure of *refresh_tree(max_domain,T)* is a function of creating FP-tree that the transaction pattern lists contain the containing *max_domain* at leaf nodes.

```
refresh_tree(max_domain,T) {
  //Create the FP-tree.
  For each 1-star pattern node N in T do {
    If the star pattern N.list_link.pattern_name contains the
    maximum domain schema max_domain then
      Create the root of an FP-tree, FP_T, and label it as
      "null."
      Scan the transaction N.list_link.tran_list once.
      Collect the set of frequent items F and their supports.
      Sort F in support descending order as L, the list of
      frequent items.
      For each transaction Trans in N.list_link.tran_list do
        Select and sort the frequent items in Trans according
        to the order of L.
        Let the sorted frequent item list in Trans be [p|P],
        where p is the first element and P is the remaining
        list.
        Call FP_insert tree([p|P]; FP_T);
    N.list_link.FP_T link FP_T;
    Delete N.list_link.tran_list.
  }
}
```

The procedure of FP-insert_tree([p|P]; T) is the creating FP-tree function every transaction list at transaction pattern.

```
FP_insert tree([p|P]; T) {
  If T has a child N such that N.item-name= p.item-name, then
    increment N's count by 1;
  Else
    Create a new node N, and let its count be 1, its parent link
    be linked to T, and its node-link be linked to the nodes
    with the same item-name via the node-link structure.
  If P is nonempty then
    Call FP_insert tree(P,N)
}
```

The procedure of *initial_tree(T)* is a function that creates all of nodes (all star node, inner node, and leaf node) and link them.

```
initial_tree(T) {
  Create a all_star_node node a, and label it as "<0,0,....0,0>.
  T.n-link link a.
  Create a parameter of inner_node point temp_point;
  For each i in n do {
    Create n of inner_node nodes tmp_node, and label them.
    T.link_list[i]=tmp_node;
    temp_point=tmp_node;
    For each j in n-2 do {
      Create n of inner_node nodes tmp_node, and label them.
      temp_point.child = tmp_node.
      temp_point = tmp_node;
    }
    Create n of leaf_node nodes tmp_node, and label them.
    temp_point.child = tmp_node;
  }
}
```

3. MINING FREQUENT PATTERNS USING TFP-TREE

Construction of a compact TFP-tree can be performed with a rather compact data structure. However, this does not generate the frequent patterns. In this section, we will study how to explore the compact information stored in a TFP-tree and develop an efficient mining method for mining the complete set of frequent patterns.

Based on the above section, we have the following algorithm for mining frequent patterns using TFP-tree.

Algorithm 2 (TFP-tree Mining)

Input: TFP-tree *T*, a minimum support threshold ε.
Output: The complete set of frequent patterns.
Method: Call FP-growth (FP-tree, *null*).

```
For each (n-1)-star pattern node N in T.link_list do {
    generate_pattern(N,ε);
}
```

The procedure of *generate_pattern*(N,ε) explore the TFP-tree until reach to the leaf node. From leaf node, this algorithm is mining the frequent patterns, call *FP-growth*(*Tree*,α), and next mining the frequent patterns, call *generate_star_pattern*(*list*,ε), at inner node step by step.

```
generate_pattern(N,ε) {
  If N is not the leaf node then
    Create a parameter of inner node m and m=N.child;
    generate_pattern(m,ε);
    For each temp_list in N.list_link do {
      generate_star_pattern(temp_list,ε);
    }
  Else
    For each trans_pattern list list in N.list_link do {
      list.freqnt_list = FP-growth(list.FP_T,null)
    }
}

generate_star_pattern(list,ε) {
  Create a parameter of frequent pattern list
  FP_list;//{freqnt_ptn,count}
  For each temp_FTP in list.pattern_list do {
    For each temp_pattern in temp_FTP do {
      If FP_list contains temp_pattern then
        FP_list.count += temp_pattern.count;
      Else
        Insert temp_pattern into FP_list;
    }
  }
  For each temp in FP_list do {
    If the temp.count is smaller than ε then
      Delete temp from FP_list;
  }
  list.freqnt_list = FP_list;
  list.pattern_list is null.
}
```

FP-growth(*Tree*,α)
{
  If *Tree* contains a single path *P* **then**
    For each combination (denoted as β) of the nodes in the path *P* **do**
      Generate pattern $\beta \cup \alpha$ with *support* = *minimum support of nodes in* β;
  Else
    For each *ai* in the header of *Tree* **do** {
      Generate pattern $\beta = a_i \cup \alpha$ with *support* = a_i.*support*;
      Construct β 's conditional pattern base and then β's conditional FP-tree $Tree_\beta$;
      If $Tree_\beta \neq \emptyset$ **then**
        call *FP-growth*($Tree_\beta$,β);
    }
}

4. EXPERIMENTAL RESULT AND EVALUATION

We conduct all the experiments on a Windows 2000 Server desktop with Pentium PC 2.8GHz and 512 Mbytes of main memory. Also, we use JDK 1.4, MS-SQL 2000 database and JDBC driver for connecting MS-SQL 2000. We choose the clicks data file in the KDD Cup 2000 data sets to perform our experiments. The clicks data file consists of homepage request records, each of which contains attribute values describing the request and the person who sent the request. We consider each request record as a transaction. The requests recorded in the clicks data file are from January 30, 2000 to March 31, 2000, which cover 8 weeks plus 6 days.

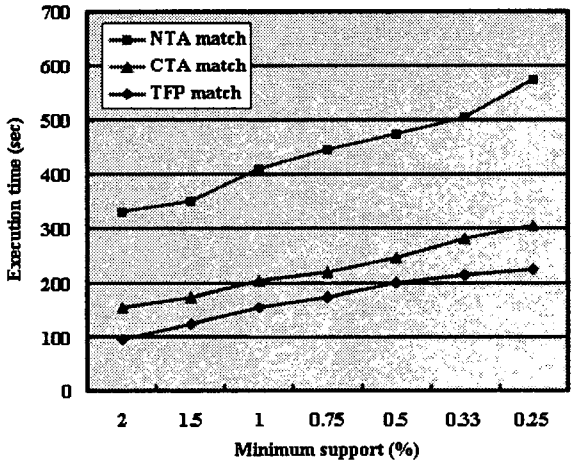

Figure 1 Execution time of three algorithms versus minimum support.

In our experiments, we compare three algorithms: nontemporal association match (NTA match), calendar temporal association match (CTA match), and temporal frequent pattern match (TFP match).

Figure 1 shows the execution time of three algorithms versus minimum support. It also shows our proposed TFP match method performances more efficient than other two algorithms.

5. CONCLUSIONS

Association rules mining is a hot-spot in data mining area. Recent years, there are many research works about association rules mining.

We have proposed a novel data structure, temporal frequent pattern tree (TFP-tree), for storing compressed, crucial information about temporal frequent patterns, and developed a pattern growth method, TFP-tree mining, for efficient mining of frequent patterns in large databases. There are several advantages of TFP-tree over other approaches: (1) this method separates many partitions by according to maximum size domain and only scans the transaction once for reducing the I/O cost, (2) this method maintains the all transactions using FP-trees, (3) we only have the FP-trees of 1-star pattern and other star pattern nodes only link them step by step for efficient mining and the saving memory.

Our performance study shows that the TFP-tree is efficient and scalable for mining, and is about an order of magnitude faster than the Apriori algorithm and also faster than calendar based on temporal frequent pattern mining methods.

Acknowledgement. This work was partially supported by ETRI (Telematics & USN Research Division) in Korea.

References:

[1] J.F. Roddick and M. Spiliopoulou.: Temporal data mining : survey and issues, Research Report ACRC-99-007, University of South Australia, 1999.

[2] Y. J. Lee, S. B. Seo and K. H. Ryu.: Discovering Temporal Relation Rules form Temporal Interval Data, Korea Information Science Society (KISS), 2001.

[3] X. Chen and I. Petrounias.: A Framework for Temporal Data Mining, In Proc. of the 9th International Conference on Database and Expert Systems Applications, 1998.

[4] Chris P. Rainsford, John F. Roddick R: Adding Temporal semantics to association rule, 3rd International conference KSS Springer 1999.

[5] Y. Li and P. Ning.: Discovering Calendar-based Temporal Association Rules, In Proc. of the 8th International Symposium on Temporal Representation and Reasoning, 2001.

[6] Chris Giannella_, Jiawei Hany, Jian Peiz, Xifeng Yany, Philip S. Yu R: Mining Frequent Patterns in Data Streams at Multiple TimeGranularities, H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha (eds.), Next Generation Data Mining, 2003.

[7] Sungbo Seo, Long Jin, Jun Wook Lee, Keun Ho Ryu, Similarity Pattern Discovery using Calendar Concept Hierarchy in Time Series Data, Asia Pacific Web Conference (APWeb), 2004.