Journal of ICT, I (I), pp: 19-31

COMPACT STRUCTURE REPRESENTATXON IN DISCOVERING FREQUENT PATTERNS FOR ASSOCIATION RULES

N. Mustapha, M. N. Sulaiman, M. Othman and M. H. Selamat

Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Selangor {norwati, nasir, mothman, hasan}@fsktm.upm.edu.my

ABSTRACT

Frequent pattern mining is a key problem in important data mining applications, such as the discovery of association rules, strong rules and episodes. Structure used in typical algorithms for solving this problem operate in several database scans and a large number of candidate generation. This paper presents a compact structure representation called Flex-tree in discovering frequent patterns for association rules. Flex-tree structure is a lexicographic tree which finds frequent patterns by using depth first search strategy. Efficiency of mining is achieved with one scan of database instead of repeated database passes done in other methods and avoid the costly generation of large numbers of candidate sets, which dramatically reduces the search space.

Key words: Frequent patterns, Candidate sets, Association rules, Lexicographic tree, Itemsets.

1.0 INTRODUCTION

The explosive growth of many business, government and scientific databases has far outpaced human ability to interpret and digest this data. Data mining therefore appears as a tool to address the need for sifting useful information such as hidden patterns from databases.

Frequent pattern mining is one of the active research themes in data mining. It covers a broad spectrum of data mining tasks including mining various kinds of association rules, strong rules, and episodes. Most algorithms for solving problem in mining frequent pattern require multiple database scans and produce a large number of candidates (Agrawal *et al.*, 1993; Agrawal, R. and Srikant, R. 1994; Park *et al.,* 1997; Brin *et al.*, 1997a; 1997b). A structure representation called *Flex-tree* is proposed in order to generate candidates free and improve the I/O costs by reducing the number of passes over the transaction database. It is a lexicographic tree which finds frequent patterns by using depth first search strategy. The basic concepts of frequent patterns and association rules are briefly reviewed in Section 2. Motivation of this work is present in Section 3. Section 4 will present the proposed algorithm for mining frequent patterns using *Flex-tree.* Experimental results and conclusions are presented in Section 5 and 6 respectively.

2.0 BASIC CONCEPTS

The task of association rules mining, first introduced in Agrawal *et al.,* (1993) can be stated as follows:

Let $\tau = \{i_1, i_2 ..., i_m\}$ be a set of items. Let D be a set of transactions where each transaction T has a unique identifier (*tid*) and contains a set of items such that $T \subseteq \tau$. A transaction containing X is set to some items in τ and $X \subseteq T$. *Patterns* are essentially a set of items and are also referred to as *itemsets*. In our later discussion, we may use the two terms – "itemsets" and "patterns" alternatively. An itemset that contains k items is a k-itemset. The *support* of an itemset X, denoted $\sigma(X)$, is the number of transactions in which it occurs as a subset. In other words, support is the measure used to evaluate the level of presence of an itemset in the database. An itemset is *frequent* if its support is more than a user-specified *minimum support* (*min_sup*) value.

An *association rule* is an implication of the form $X \Rightarrow Y$, where X and Y are itemsets and $X \cap Y = \phi$. The support of the rule is given as $\sigma(X \cup Y)$ (i.e. the joint probability that a transaction containing both X and Y) and the *confidence* as $\sigma(X \cup Y) / \sigma(X)$ (i.e., the conditional probability that a transaction contains Y, given that it contains X). The rule is strong if its confidence is more than a user-specified *minimum confidence* (*min_conf*). The idea of an association rule is to develop a systematic method by which a user can figure out how to infer the presence of some items, given the presence of other items in a transaction. Such information can extend the application of association rules to finding useful patterns in consumer behavior, target marketing, and electronic commerce.

The problem of mining association rules is to generate all association rules in the database that have certain user-specified *min_sup* and *min_conf.* This can be decomposed into two steps (Agrawal and Srikant, 1994):

(i) find all frequent pattern.
(ii) generate strong association rules from frequent patterns.

Consider an example customer database shown in Fig. 1. There are five different items (name of the items the supermarket carries), i.e., $\tau = \{a, b, c, d, e\}$, and the database consists of six customers who bought items from the supermarket. Fig. 2 shows all the frequent patterns that are contained in at least three customer transactions, i.e., *min_sup* = 50%. Fig. 3 shows the set of all association rules with *min_conf* = 100%.

ITEMS

Bread	a
Butter	b
Milk	c
Cheese	d
Coke	e

DATABASE

Transaction ID (*tid*)	Items Bought
1	a b d e
2	b c e
3	a b d e
4	a b c e
5	a b c d e
6	b c d

Fig. 1: A Customer Database

FREQUENT PATTERNS (min_sup=50%)	
Itemsets	Support
B	100% (6)
e be	83% (5)
a c d ab ae bc bd abe	64% (4)
ad ce de abd ade bce bde abde	50% (3)

Fig. 2: The Frequent Patterns With *min_sup* = 50%

ASSOCIATION RULES (min_conf=100%)	
a $\Rightarrow$ b (4/4)	ae $\Rightarrow$ b (4/4)
a $\Rightarrow$ e (4/4)	ce $\Rightarrow$ b (3/3)
a $\Rightarrow$ be (4/4)	de $\Rightarrow$ a (3/3)
c $\Rightarrow$ b (4/4)	de $\Rightarrow$ b (3/3)
d $\Rightarrow$ b (4/4)	ad $\Rightarrow$ be (3/3)
e $\Rightarrow$ b (5/5)	de $\Rightarrow$ ab 3/3)
ab $\Rightarrow$ e (4/4)	abd $\Rightarrow$ e (3/3)
ad $\Rightarrow$ b (3/3)	ade $\Rightarrow$ c (3/3)
ad e (3/3)	bde a (3/3)

Fig. 3: The Association Rules with *min_conf* = 100%

3.0 MOTIVATION

Most of the previous studies such as Agrawal and Srikant (1994), Klemettinen *et al.* (1994), Park *et al.* (1995), Savasere *et al.* (1995) adopt the approach called Apriori-like approach. This approach is based on an anti-monotone heuristic which is any length k -patterns is not frequent in the database, its length (k+1) super-patterns can never be frequent (Agrawal and Srikant, 1994). This heuristic achieves good performance gain by (possible significantly) reducing the size of candidate sets but it may still suffer from the following nontrivial costs:

(a) It is costly to handle a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 length-2 candidates and accumulate and test their occurrence frequencies. Moreover, to discover a frequent pattern of size 100, such as $\{a_1, \ldots, a_{100}\}$, it must generate more than $2^{100} \approx 10^{30}$ candidates in total. This is the inherent cost of candidate generation, no matter what implementation technique is applied.

(b) It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching to count their support, which is especially true for mining long patterns.

The bottleneck of the Apriori-like method is at the candidate set generation and test. If one can avoid generating a huge set of candidates to test, the mining performance can substantially improve.

Is there any other way that one may reduce these costs in frequent patterns mining? Could some novel data structure or algorithm help? *Flex-tree* is introduced in the next section to explore these posibilities.

4.0 MINING FREQUENT PATTERNS USING FLEX-TREE

A compact data structure for efficient frequent pattern mining can be designed based on the following observations:

i. Since only the frequent items will play a role in frequent pattern mining, it is necessary to perform a single scan of database to identify the list of all transactions containing the item.

ii. If we store the list of items together with the list of its transaction identifier, we may avoid repeated scanning of database, with frequency count obtained by intersecting the list of *tids*. In addition, support of potential branch of particular node can be checked before it is completely generated if its support is greater than minimum support.

iii. The nodes generated are certainly frequent by *restricted test-and-generation* operation, therefore any pruning procedure to remove infrequent branches are not necessary.

4.1 Structure of Flex-tree

Flex-tree is the abbreviation for frequent lexicographic tree which is lexicographic tree-based. Assume that a lexicographic ordering exist among the items in the database. In order to indicate that an item *i* occurs lexicographically earlier than *j*, the notation $i < j$ will be used. Flex-tree is an abstract representation of the frequent itemsets with respect to this ordering. The Flex-tree is defined in the following way:

(1) A node exists in the tree corresponding to each frequent itemset. The root of the tree corresponds to the *null* itemset.

(2) Let $\tau = \{i_1, i_2 ..., i_m\}$ be a frequent itemset, where $i_1, i_2 ..., i_m$ are listed in lexicographic order. The parent of the node τ is the itemset $\{i_1, i_2 ..., i_{m-1}\}$.

Flex-tree is rooted at the *null* node. An example of the Flex-tree is illustrated in Fig. 4 which is based on customer database shown in Fig.1. A frequent 1-extension of an itemset such that the last item is the contributor to the extension will be called a frequent lexicographic tree extension. Thus, each edge in the Flex-tree corresponds to an item which is the frequent lexicographic tree extension to a node. The set of frequent lexicographic tree extensions of a node *N* is denoted by $\varepsilon(N)$. In the example illustrated in Fig. 4, the frequent lexicographic extensions of node *a* are *b*, *c*, *d*, and *e*.

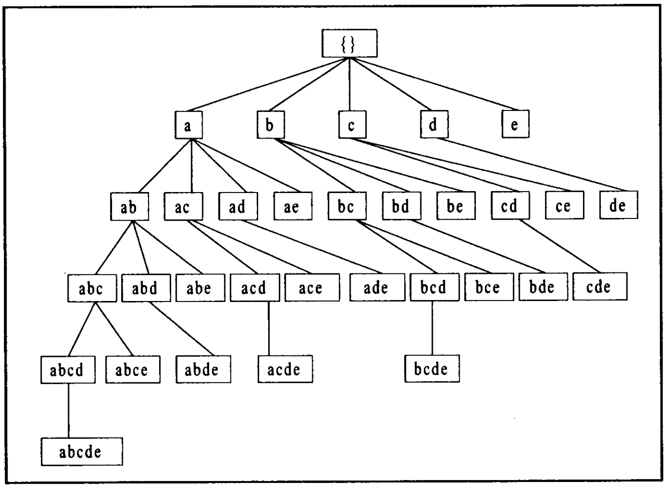

Fig. 4: The Lexicographic Tree

Let *M* be the immediate ancestor of the itemset *N* in the lexicographic tree. The set of *prospective branches* of a node *N* is defined to be those items in $\varepsilon(M)$ which occur lexicographically after the node *N*. These are the possible frequent lexicographic extensions of *N* and denoted as $\rho(N)$. Thus, the following relationship is constructed: $\varepsilon(N) \subseteq \rho(N) \subset \varepsilon(M)$. The value of $\varepsilon(N)$ in Fig. 4, when $N = ab$ is $\{c, d\}$. The value of $\rho(N)$ for $N = ab$ is $\{c, d, e\}$, and for $N = ae$ $\rho(N)$ is empty.

The node is said to be *generated*, the first time its existence is discovered by virtue of the extension of its immediate parent. A node is said to have been *tested*, when its frequency has been determined. Thus the process of testing a node *N* results in its generation if its support exceeds *min_sup*, unless the set $\varepsilon(N)$ for that node is empty. Obviously a node can be generated only after it has been tested. This can ensure that the set of branches which occur in Flex-tree are certainly frequent; therefore it does not need any pruning procedure to prune the tree.

4.2 The Algorithm

The algorithm shown in Fig. 5 constructs the Flex-tree by starting at the node null and successively generating nodes until all nodes have been tested and subsequently generated. Flex-tree shown in Fig. 4 will be used as a running example in the mining process of frequent patterns.

At the first level of Flex-tree, each node will be examined with all the transactions in database using level-wise manner. Consider the node *a*, all transactions projected at node *a* would be 1, 3, 4 and 5 are stored as a list of transaction identifiers together with the node.

At subsequent levels, all the nodes will be examined in depth-first fashion. The examination of a node is the process of counting the support of the node's potential candidates. In other words, the support of all descendant patterns of a node is determined before determining the extensions of other nodes of the Flex-tree. Potential candidates are tested in the counting phase to ensure that they are the ones whose support is the same or greater than *min_sup* before inserting a corresponding node in the Flex-tree. Computing support of itemsets in each node using depth-first manner will be discussed later. Depth-first strategy would count the extension of nodes in Fig. 4 starting with *ab*, *abc*, *abcd*, *abcde*, *abce*, *abd*, *abde*, *abe*, *ac*, *acd*, *acde*, *ace*, *ad*, *ade* and so on.

The first step of the Flex-tree algorithm is to create the first level of the tree by generating all the frequent 1-itemsets using breadth-first strategy to get their support. This is accomplished by calling the procedure *getFreqItems (transactions, min_sup).*

```
// 1st level of Flex-tree using BFS together with counting
// support

N = getFreqItems(transactions, min_sup);

// level-n of Flex-tree using DFS together with tid-list
// intersection

FLex-tree(N: ItemsetNode; min_sup)
{
  C = PotentialBranch(N);
  E = CountExtensions(N, C);
  {Let E = {$i_1$, …, $i_{|E|}$}, when expressed in lexicographic order}

  InsertFLex-tree(N, C, min_sup);

  for (r=1; r$\leq$ |E|; r++) {
   FLex-Tree(C, min_sup);
  }
}

InsertFLex-tree(N: ItemsetNode; C:CandidateNode; min_sup)
{
  $\sigma(C) = \sigma(N) \cap \sigma(N+1)$;
  if ($\sigma(C) \geq$ min_sup)
  C = N $\cup$ {$i_r$} for r $\varepsilon$ {1, …, |E|};
}
```

```
PotentialBranch(Itemset Node: N)
{
    if (N == ∅)
      return all items;
    else
      return frequent extensions of N which are lexicographically
      larger than any item in N;

}


CountExtensions(N: ItemsetNode; C: CandidateNode)

{
   count all the possible frequent extensions of node N
   denoted by candidate set C.

}
```

Fig. 5: Algorithm Flex-tree

Depth-first creation of the Flex-tree begins after the set of frequent 1-itemset has been found. Search for potential candidates *C* must be accomplished by the procedure call *PotentialBranch (N).*

Before we insert nodes for the next level, support of each candidate must be counted to ensure its support is exceeded or greater than *min_sup* by calling *InsertFlex-tree(N: ItemsetNode; C:CandidateNode; min_sup).* In this procedure, intersections of *tids* are performed to compute support of itemsets.

4.3 Computing Support of Itemsets

There are two possible layouts of the database for mining frequent patterns. The *horizontal* layout (Agrawal and Srikant, 1994) consists of a list of transactions. Each transaction has an identifier followed by a list of items as shown in Fig 1. The *vertical* layout (Holsheimer *et al.*, 1995) consists of a list of items. Each item *X* has a *tid-list* (the list of all transactions containing the item) denoted as $\ell(X)$. For example shown in Fig. 6, $\ell(a) = \{1, 3, 4, 5\}$ and $\ell(b) = \{1, 2, 3, 4, 5, 6\}$.

Transformation from horizontal to vertical format is done in the Flex-tree at the first level. At subsequent levels of the Flex-tree, the vertical format is used because it seems more suitable for mining frequent patterns since the support of a candidate *k*-itemset can be computed by simple *tid-list* intersections. The *tid-lists* cluster relevant transactions; this avoids scanning the whole database to compute support, and the larger the itemset, the shorter the *tid-lists*, resulting in faster intersections. Each item is associated with its *tid-list*,

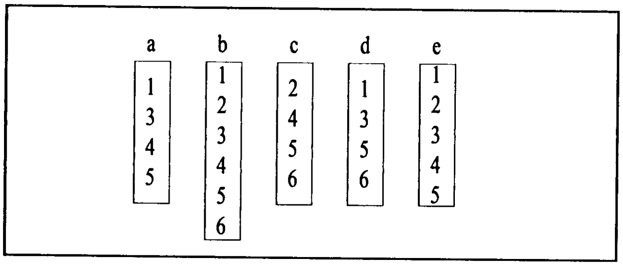

Fig. 6: The Vertical Database

Since the frequent itemsets and its support will play a role in frequent pattern mining, designing a compact structure is important in order to find support count without scanning database repeatedly. One of the key features of our structure is that it requires only a single database scan which can minimize the I/O cost. The nodes generated are certainly frequent because our structure is *restricted test-and generation* instead of Apriori-like *restricted generation-and-test*. Consider a node *k*-itemset, at which it is potentially a candidate to be generated if its support count is no less than a predefined *min_sup* threshold. By using this structure, the support of any node (*k*-itemset) can be determined by simply intersecting the *tid-lists* of any two of its (*k*-1) length subsets.

A simple check on the cardinality of the resulting *tid-list* tells us whether the new itemset is frequent or not. Fig. 7 shows this process pictorially. It shows the initial *tid-list* for each item. The intermediate *tid-list* for *ab* is obtained by intersecting the lists of *a* and *b* i.e. $\ell(ab) = \ell(a) \cap \ell(b)$. Similarly, $\ell(abc) = \ell(ab) \cap \ell(ac)$, and so on. Thus, only the lexicographic first two subsets at the previous level are required to compute the support of itemsets at any level. A practical and important consequence of this situation is that the cardinality of intermediate *tid-lists* shrink as we move down the tree. No pruning technique is needed on the tree by using *restricted test-and generated* approach.

Table 1: Database Characteristics

Database	#Items	Record Length	# Records
chess	75	37	3,196
connect-4	129	43	67,557
mushroom	119	23	8,124

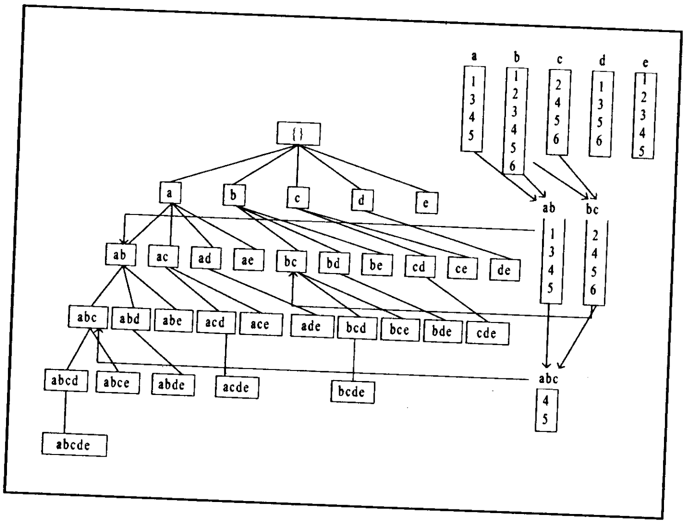

Fig. 7: Computing Support of Itemsets

5.0 EXPERIMENTAL RESULTS

All experiments below were performed on a 450MHz Pentium PC with 256MB of memory. Table 1 shows the characteristics of the real datasets taken from UCI Irvine Machine Learning Database Repository (Murphy, P.M.).

The mushroom database contains characteristics of various species of mushrooms. The connect-4 and chess datasets are derived from their respective game steps. Typically, these real datasets are very dense, i.e., they produce many long frequent patterns even for very high values of support.

Table 2 shows the total number of frequent patterns and the total time obtained by setting minimum support as shown in column 2 using Flex-tree structure.

Table 2: Number of Patterns and Running Time
(Sup=minimum support; Len=longest frequent pattern)

Database	Sup	Len	#Freq patterns	Running time (sec)
Chess	90%	7	622	1.92
chess	80%	10	8225	43.34
chess	70%	13	48443	392.33
connect-4	97%	6	487	17.19
connect-4	90%	10	27127	4062.13
connect-4	85%	13	142127	20519.18
mushroom	40%	7	565	1.54
mushroom	30%	9	2735	8.62
mushroom	20%	15	53583	3688.3

No candidate sets were presented in the table because the Flex-tree always make sure all patterns must be frequent before they can be generated.

This algorithm scans database only once for each of those three databases that were used in experiments. Database scan is done once during the first level of building the Flex-tree for support counting purposes. In subsequent levels, it does not need to refer to original database anymore to get support of each pattern by intersecting *tid-lists* which are stored together with corresponding patterns.

6.0 CONCLUSION

The experimental result shows that by storing the database in Flex-tree structure and mining it in depth-first fashion, efficiency of mining is achieved with one scan of database instead of repeated database passes done in other methods. It also avoids the costly generation of large number of candidate sets which dramatically reduces the search space. This structure is proved that no matter how big the database is and how long the frequent pattern will be, Flex-tree always need only one database scan to generate the complete set of frequent patterns.

Comparison of this structure with others will be implemented in the next version for a more improved method. Experiments should be done up on more datasets for testing this structure representation to get more reliable results.

REFERENCES

Agrawal, R. I. T., & Swami, A. (1993). Mining Association Rules between Sets of Items in very Large Databases. *Proceedings of the ACM SIGMOD Conference on Management of Data*, 207-216.

Agrawal, R., & Srikant, R. (1994). Fast Algorithms for Mining Association Rules. *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, 487-499.

Brin, S. M. R., & Silverstein, C. (1997a). Beyond Market Baskets: Generalizing Association Rules to Correlation. *Proceedings of ACM SIGMOD*, 265-276.

Brin, S. M. R., Ullman, J.D., & Tsur, S. (1997b). Dynamic Itemset Counting and Implication Rules for Market Basket Data. *Proceedings of ACM SIGMOD*, 255-264.

Holsheimer, M., Kersten, M., Mannila, H., & Toivonen, H. (1995). A Perspective on Databases and Data Mining. 1st *KDD Conference.*

Klemettinen, M., Manilla, H., Ronkainen, P., Toivonen, H., & Verkamo, A.I. (1994). Finding Interesting Rules from Large Sets of Discovered Association Rules. *Proceedings of CIKM'94*, 401-408.

Murphy, P.M. Repository of Machine Learning and Domain Theories. http://www.ics.uci.edu/~mlearn/MLRepository.html.

Park J.S., Chen, M.S & Yu, P.S. (1995). Using a Hash-based Methods with Transaction Trimming for Mining Association Rules. *IEEE Transaction of Knowledge and Data Engineering*, 9(5):813-825.

Savasere, A., Omiecinski, E., & Navathe, S. (1995). An Efficient Algorithm for Mining Association Rules in Large Databases. *Proceedings of the International Conference on Very Large Databases (VLDB'95)*, 432-443.