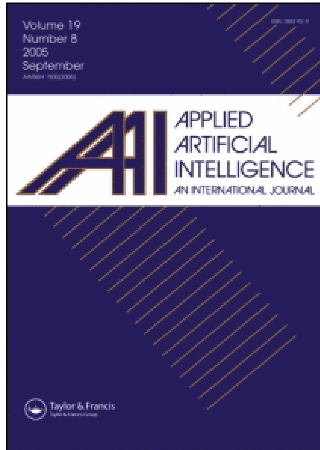
This article was downloaded by:[University of Technology Sydney]
[University of Technology Sydney]
On: 7 May 2007
Access Details: [subscription number 770853977]
Publisher: Taylor & Francis
Informa Ltd Registered in England and Wales Registered Number: 1072954
Registered office: Mortimer House, 37-41 Mortimer Street, London W1T 3JH, UK



Applied Artificial Intelligence
An International Journal
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713191765

ON DATA STRUCTURES FOR ASSOCIATION RULE DISCOVERY

To cite this Article: , 'ON DATA STRUCTURES FOR ASSOCIATION RULE DISCOVERY', Applied Artificial Intelligence, 21:2, 57 - 79
To link to this article: DOI: 10.1080/08839510601115544
URL: http://dx.doi.org/10.1080/08839510601115544

PLEASE SCROLL DOWN FOR ARTICLE

Full terms and conditions of use: http://www.informaworld.com/terms-and-conditions-of-access.pdf

This article maybe used for research, teaching and private study purposes. Any substantial or systematic reproduction, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The accuracy of any instructions, formulae and drug doses should be independently verified with primary sources. The publisher shall not be liable for any loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.

© Taylor and Francis 2007

Applied Artificial Intelligence, 21:57–79
Copyright © 2007 Taylor & Francis Group, LLC
ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839510601115544

Taylor & Francis
Taylor & Francis Group

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

ON DATA STRUCTURES FOR ASSOCIATION RULE DISCOVERY

Xiaowei Yan and Shichao Zhang □ *Computer Department, Guangxi Normal University, China*

Chengqi Zhang □ *Faculty of Information Technology, University of Technology, Sydney, Australia*

□ *Systematically we study data structures used to implement the algorithms of association rule mining, including hash tree, itemset tree, and FP-tree (frequent pattern tree). Further, we present a generalized FP-tree in an applied context. This assists in better understanding existing association-rule-mining strategies. In addition, we discuss and analyze experimentally the generalized k-FP-tree, and demonstrate that the generalized FP-tree reduces the computation costs significantly. This study will be useful to many association analysis tasks where one must provide really interesting rules and develop efficient algorithms for identifying association rules.*

A data structure is a way of organizing information. The design of an appropriate data structure can often be the foundation for an efficient algorithm (Aho et al. 1976). This paper is focused on the data structures for implementing the algorithms of association rule mining.

In most implementations of algorithms for association rule mining, a special kind of tree known as a prefix tree, is used to organize itemsets and store their support counters. This data structure also allows us to process transactions in database and generate association rules. In this data structure, each node represents an itemset. The root corresponds to the *null* itemset. Suppose that node A is the parent of node B. Then the itemset of node A is a subset of the itemset of node B. When an itemset is taken as a sequence instead of a set, the itemset of node A is a prefix of the itemset of node B. Therefore, this kind of data structure is commonly referred

This research is partially supported by Australian Large ARC Grants (DP0449535, DP0559536, and DP0667060), a China NSF Major Research Program (60496327), a China NSF Grant (60463003), an Overseas Outstanding Talent Research Program of the Chinese Academy of Sciences (06S3011S01), a High-level Studying-Abroad Talent Program of the China Human-Resource Ministry, and a Guangxi Natural Science Fund.

Address correspondence to S. Zhang, Faculty of Information Technology, University of Technology, Sydney, PO Box 123, Broadway NSW 2007, Sydney, Australia. E-mail: zhangsc@it.uts.edu.au

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

to as a *prefix tree*. For example, let $I = \{i_1, i_2, \ldots, i_k\}$ be an itemset, where $\{i_1, i_2, \ldots, i_k\}$ are listed in lexicographic order. Then the parent of node I is the itemset $\{i_1, \ldots, i_{k-1}\}$. Such a prefix tree is also called a *lexicographic tree*, because an itemset in the tree can be viewed as a lexicographically sorted sequence.

Many variations of the prefix tree have been developed for association rule mining since the prefix tree, namely the *hash tree*, was first used to represent itemsets in Agrawal and Srikant (1994). Brin et al. (1997), for instance, introduced an improved prefix tree, called the *itemset tree*. Agarwal et al. (2000; 2001) also used a similar prefix tree, namely the *lexicographic tree*. Han et al. (2000) designed an extended prefix-tree structure called the *Frequent Pattern Tree* (*FP*-tree). *FP*-tree was modified to the *threaded transaction forest* (*TTF*) by Liu et al. (2002), changed to the *Prefix-Path Tree* (*PP*-tree) by Xu et al. (2002), and extended to the *Pattern Tree* (*P*-tree) by Huang et al. (2002). This well-known tree was also enhanced by Pei et al. to the *Hyper-links Structure* (*H*-struc), another kind of tree for association rule mining in sparse data sets (Pei et al. 2001). Sucahyo and Gopalan (2003) integrated several ideas of data structures, including the itemset tree, *H*-struc, and *FP*-tree, into a new mining algorithm called *CT-ITL*. In *CT-ITL*, a compressed transaction tree (*CT*) is developed, and a linked-list data structure, called *Item-TransLink* (*ITL*), is used to track items. Another recent work is the *Co-Occurrence Frequent Item Tree* (*COFI*-tree) (El-Hajj and Zaane 2003), which is a small *FP*-tree-like tree built for each frequent 1-itemset. But unlike the *FP*-tree-based algorithm *FP*-growth, the *COFI*-tree-based mining algorithm is non-recursive, and requires less memory to generate frequent itemsets.

Theoretically, there are two strategies of generating a prefix tree: width-oriented strategy and the depth-oriented strategy. In general, depth-oriented strategy is more efficient, especially for dense databases and low support thresholds, while the width-oriented strategy is more scalable.

The candidate-generation-and-test approach proposed in *Apriori* (Agrawal and Srikant 1994), for example, and its many subsequent variations fall into the width-oriented strategy (Agarwal et al. 2001; Agrawal et al. 1996; Brin et al. 1997; Han and Fu 1995; Park et al. 1995; Savasere et al. 1995; Toivonen 1996; Zaki et al. 1997). These methods try to reduce the number of candidates generated, the number of transactions to be scanned, or the number of database scans. Because the anti-monotone property states that any subset of a frequent itemset is also a frequent itemset, these approaches generate a set of candidate itemsets of length $n+1$ from the set of itemsets of length n. Each candidate itemset is checked to see whether it is frequent by scanning the database and counting its support. The prefix tree is created level by level. As a result, the width-oriented approach needs to scan the database many times. Clearly, this approach is

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

inefficient especially when a dense database with long frequent itemsets is required for processing.

FP-growth (Han et al. 2000), *MAFIA* (Burdick et al. 2001), *H*-Mine (Pei et al. 2001), and *CT-ITL* (Sucahyo and Gopalan 2003), on the other hand, belong to the depth-oriented strategy. In this depth-oriented approach, generations of candidate itemsets and their support counting are processed simultaneously when scanning the database. Therefore, the depth-oriented approach reduces the number of database scans and has a better performance of the mining process. But its generation of candidate itemsets needs to recursively build as many conditional trees in order of magnitude as the frequent itemsets. This massive creation of conditional trees still limits its scalability.

Working with the data mining group of CMCRC (Australian Capital Markets on Cooperative Research Center), we use the prefix tree to store the itemsets in our mining algorithm developed for identifying false alerts in stock marketing (Yan et al. 2003; 2004). Therefore, in this article, we first investigate the construction, organization, and itemset encoding of the prefix tree, make comparisons among the existing prefix trees, and propose an expanded data structure for storing itemsets.

PRELIMINARIES

$I = \{i_1, i_2, \ldots, i_m\}$ is a set of literals, or *items*. X is an *itemset* if it is a subset of I.

$D = \{t_i, t_{i+1}, \ldots, t_n\}$ is a set of transactions, called the *transaction database*, where each transaction t has a *tid* and a t-itemset: $t = (tid, t\text{-itemset})$. A transaction, t, contains an itemset, X, if and only if, (iff) for all items, $i \in X$, i is in t-itemset.

There is a natural *lattice structure* on the itemsets, 2^I, namely, the subset/superset structure. Certain nodes in this lattice are natural grouping categories of interest (some with names).

An itemset, X, in a transaction database, D, has a *support*, denoted as $supp(X)$ (we also use $p(X)$ to stand for $supp(X)$), that is the ratio of transactions in D contain X. Or

$$supp(X) = |X(t)|/|D|,$$

where $X(t) = \{t \text{ in } D | t \text{ contains } X\}$.

An itemset, X, in a transaction database, D, is called a large (frequent) itemset if its support is equal to, or greater than, a threshold of minimal support (*minsupp*), which is given by users or experts.

An *association rule* is an implication $X \rightarrow Y$, where itemsets X and Y do not intersect.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

Each association rule has two quality measurements: *support* and *confidence*, defined as follows.

- The support of a rule $X \rightarrow Y$ is the support of $X \cup Y$, where $X \cup Y$ means both X and Y occur at the same time.
- The confidence of a rule $X \rightarrow Y$ is $con\,f(X \rightarrow Y)$ as the ratio: $|(X \cup Y)(t)|/|X(t)|$ or $supp(X \cup Y)/supp(X)$.

That is, support = frequencies of occurring patterns; confidence = strength of implication.

Support-confidence framework (Agrawal et al. 1993): Let I be the set of items in database $D, X, Y \subseteq I$ be itemsets, $X \cap Y = \emptyset$, $p(X) \neq 0$ and $p(Y) \neq 0$. Minimal support ($minsupp$) and minimal confidence ($mincon\,f$) are given by users or experts. Then $X \rightarrow Y$ is a valid rule if

1. $supp(X \cup Y) \geq minsupp$,
2. $conf(X \rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)} \geq mincon\,f$,

where "$con\,f(X \rightarrow Y)$" stands for the confidence of the rule $X \rightarrow Y$.

To demonstrate the use of the support-confidence framework, we illustrate the process of mining association rules by an example as follows.

Example 1. Assume that we have a transaction database shown in Table 1. There are 6 transactions in the database with 100 to 600 as their transaction identifiers (TIDs), respectively. The universal itemset $I = \{A, B, C, D, E\}$, where *A*, *B*, *C*, *D*, and *E* can be any items in the supermarket, for instance, A = "bread," B = "milk," C = "sugar," D = "coffee," and E = "biscuit."

There are totally $2^5 (= 32)$ itemsets. {*A*}, {*B*}, {*C*}, {*D*}, and {*E*} are all 1-itemsets, {*AC*} is a 2-itemset, and so on. $supp(BC) = 4/6 = 0.67$ because there are four transactions that contain both *A* and *B*. Let $minsupp = 50\%$ and $mincon\,f = 80\%$. Then, *A*, *B*, *C*, *E*, *AC*, *BC*, *BE*, and *BCE* are all frequent itemsets. The confidence of association rule $A \rightarrow C$ is $con\,f(A \rightarrow C) = supp(AC)/supp(A) = 3/3 = 1.0$. Hence, rule $A \rightarrow C$ is valid. Similarly, we have $con\,f(C \rightarrow A = 3/5 = 0.6$. Hence, rule $C \rightarrow A$ is not valid.

TABLE 1 An Example Transaction Database

TID	Items bought
100	*ABCD*
200	*BCE*
300	*ABCE*
400	*BE*
500	*ACD*
600	*BCE*

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

HASH TREE

Hash tree is the preexistence of the prefix tree for maintaining frequent itemsets in algorithms of association rule mining. In this section, we introduce the hash tree before turning to the prefix tree in the next section.

Overview

Hash tree is used in Agrawal and Srikant's (1994) *Apriori* algorithm to store the candidate k-itemsets. But the details of this data structure are not given in the paper. In this section, we are going to describe the hash tree according to our implementation.

As introduced in Agrawal et al. (1993), the *Apriori* algorithm can be stated as follows.

1. Determine the support for all 1-itemsets (single attributes).
2. Delete all the 1-itemsets that are not adequately supported, such that the remaining 1-itemsets are all frequent.
3. For all frequent 1-itemsets, construct candidate 2-itemsets (pairs of attributes).
4. If there is no 2-itemset, terminate; otherwise, determine the support for the constructed 2-itemsets and select those frequent 2-itemsets.
5. For all supported 2-itemsets, construct candidate 3-itemsets (triples).
6. If there is no triple, end; otherwise, determine the support for the constructed triples.
7. Continue until no more candidate itemsets are produced.

At each level k, the candidate k-itemsets C_k are stored in a hash tree. A hash tree is a dynamic multilevel hash table, where:

- The root is at depth 1 ($d = 1$), the second-level nodes are at depth 2 ($d = 2$), and so forth.
- Leaf nodes are itemsets and internal nodes are hash tables.
- The hash table interprets the hashing function.
- An element of a hash table at depth d can be taken as a pointer to another child node at depth $d + 1$.

Figure 1 is an example of a hash tree for candidate 3-itemsets. The hash table, i.e., the hashing function for a given itemset I at depth d, is defined as

$$\mathrm{h}(I, d) = \begin{cases} \textit{left} & \text{if } I(d) = 1, 4, \text{or } 7, \\ \textit{middle} & \text{if } I(d) = 2, 5, \text{or } 8, \\ \textit{right} & \text{if } I(d) = 3, 6, \text{or } 9, \end{cases}$$

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

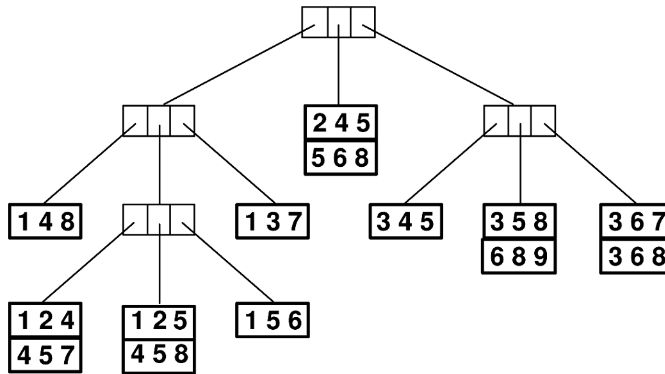


FIGURE 1 An example of candidate hash tree.

where $I(d)$ is the dth element of I. The values of function $\mathrm{h}(I, d)$ denote the branches where we shall go with the itemset I. Namely, *left*, *middle*, and *right* denote the left-hand, middle, and right-hand branches, respectively.

Generation of Hash Tree

Given a transaction database, hash trees are generated in a procedure according to the *Apriori* algorithm. We assume that the maximum number of itemsets that can be stored at a leaf node is m, and items in each transaction are sorted in lexicographic order. m is a specified threshold, representing the size of a hash bucket.

First, the itemsets with level $k = 1$ are considered. That is, a hash tree is constructed for the set C_1 of candidate 1-itemsets, which only contains all items in the transaction database. We commence with creating the root. Whenever a node is generated, it is initialized as a leaf node. Itemsets can be added to this leaf node. When the number of itemsets in a leaf node exceeds the given maximum number m of itemsets, and the depth d where the leaf node stays is less than the level $k + 1$, the node is converted to an internal node. Some new descendent nodes of it are created. Those itemsets, which originally belong to the new internal node, now move down to these new leaf nodes according to the hash table of the internal node. In order to save space, this conversion from leaf node to interior node does not take place until the number of itemsets on the leaf node accumulates up to the threshold m. Unlike the hash tree given by Agrawal and Srikant (1994), we additionally require that the depth d of an interior node should not exceed the length k of an itemset. This is because we can't decide which branch to follow at an interior node at depth d by applying a hash function to the dth item of the itemset, if $d > k$. We can apply other techniques to process this overflow problem of itemsets on leaf nodes at depth $k + 1$.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

After we finish the construction of the candidate 1-itemset hash tree, we count the support of all itemsets in the hash tree for all transactions in the database, by using the traversal strategy discussed later. Only frequent 1-itemsets are chosen to compose the set L_1.

Then, itemsets with level $k = 2$ are considered. Namely, we are going to construct a hash tree for the set C_2 of 2-itemsets. In Agrawal and Srikant's *Apriori* algorithm, a function, called $apriori-gen(L_{k-1})$, is used to generate the set of candidate k-itemsets, C_k, from the set of frequent $(k-1)$-itemsets, L_{k-1}. The function first joins L_{k-1} with L_{k-1}, and obtains an initial C_k. Then it prunes those itemsets c in C_k, which contain such a $(k-1)$-subset of c that is not in L_{k-1}. More about this function can be reached in Agrawal and Srikant (1994) and Agrawal et al. (1993).

Once C_k is determined, the same process as that for C_1 can be applied for the hash tree construction of C_k, and supports of all itemsets in C_k are counted through traveling the tree for each transaction in the database. This procedure is iterated until L_{k-1} is empty.

An Example

We now give an example to demonstrate the generation of a hash tree. Suppose that we have a transaction database as shown in Table 2.

Given a lexicographically-ordered itemset I and a depth d, the hashing function is simply defined as

$$\mathrm{h}(I, d) = \begin{cases} \textit{left} & \text{if } I(d) = 1 \text{ or } 4, \\ \textit{middle} & \text{if } I(d) = 2 \text{ or } 5, \\ \textit{right} & \text{if } I(d) = 3 \text{ or } 6, \end{cases}$$

which is also intuitively represented by Figure 2.

The function will select one of the three possible values: *left*, *middle*, and *right*. Consequently, interior nodes in the hash tree will have three branches. We also assume that the maximum number of itemsets that can be stored at a leaf node is two. That is, $m = 2$.

TABLE 2 A Simplified Example of Transaction Database

TID	Transaction
10	*1 2*
20	*1 3 6*
30	*1 2 3*
40	*2 4*
50	*2 3 6*
60	*5 6*

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

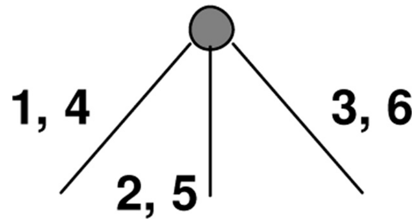
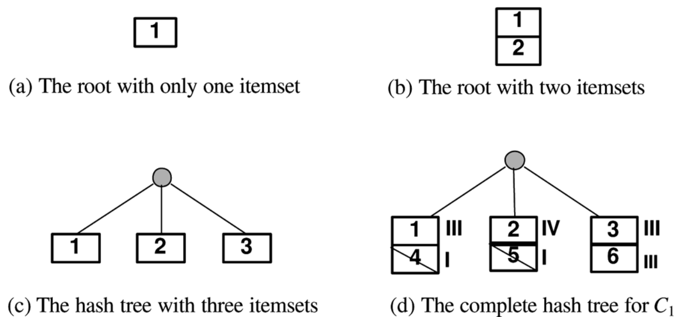


FIGURE 2 The hash function.

We begin with considering the 1-itemsets $C_1 = \{1, 2, 3, 4, 5, 6\}$. First, the itemset {1} is added to the root, as shown in Figure 3 (a). Figure 3 (b) shows the result after itemset {2} is added. When the third itemset {3} is added, the number of itemsets goes over the threshold *m*. Therefore, we convert the root to an interior node and the itemsets of {1}, {2}, and {3} are moved down to the three new generated leaf nodes, respectively, as shown in Figure 3 (c). Similarly we can insert the itemsets of {4}, {5}, and {6} to their corresponding leaf nodes by applying the hashing function to their first element, respectively. The finally resulted hash tree for C_1 can be seen in Figure 3 (d). We then pass through the database, transaction by transaction, updating supports of the itemsets in the hash tree by the tree traversal algorithm presented in the next subsection. In Figure 3 (d), the roman numerals beside the itemset boxes represent the supports for the itemsets.

From Figure 3 (d), we can see that $L_1 = \{1, 2, 3, 6\}$. By using function of *apriori-gen*(L_1) in the Agrawal and Srikant's Apriori algorithm, we can obtain $C_2 = \{12, 13, 16, 23, 26, 36\}$. Now we generate the 2-itemset hash tree, as shown in Figure 4, having $L_2 = \{12, 13, 23, 36\}$.

Again, we have $C_3 = \textit{apriori-gen}(L_2) = \{123\}$. Therefore, the 3-itemset tree is as simple as that shown in Figure 5. Because the support of {123} is 1, we have $L_3 = \emptyset$ and the algorithm terminates.



(a) The root with only one itemset

(b) The root with two itemsets

(c) The hash tree with three itemsets

(d) The complete hash tree for C_1

FIGURE 3 Candidate 1-itemset hash tree.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

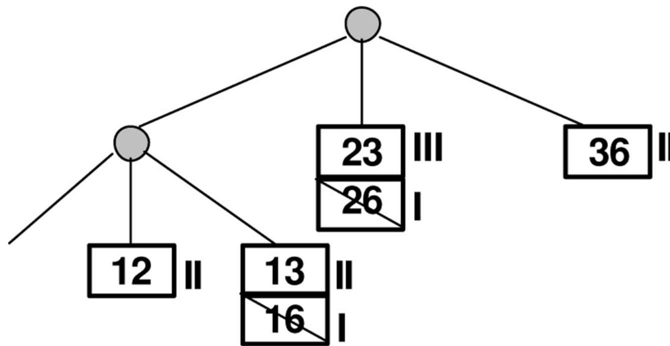


FIGURE 4 Candidate 2-itemset hash tree (support counts included).

Ergodic Algorithm

Given a transaction database and a hash tree for C_k, we need to calculate the supports for all the itemsets in C_k by ergoding the corresponding hash tree for each transaction in the database. During the ergodic process for a transaction, those candidate itemsets which are subsets of the transaction are temporarily stored in a buffer, called the answer set. On completion of this travel, the supports of these itemsets in the answer set are updated by an increment of one. The ergodic algorithm is explained next.

To ergode the tree, we start at the root node with a transaction t, and t is taken as an initial itemset. We are going to check which itemsets in the hash tree are contained in t. We apply the hashing function to each item in t in turn, by which we can decide which branches we should go along. When we reach a node by hashing on item i, we remove any items before item i in t and obtain a new itemset. If the reached node is an interior node, we continue to go down by hashing on every item that is behind i in the new itemset recursively, until we reach a leaf or the number of items after i in

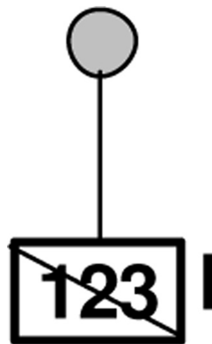


FIGURE 5 Candidate 3-itemset hash tree.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

t is less than $k - d$, where d is the depth of the reached node. If there are less than $k - d$ items left after i in the new itemset, it is unnecessary to keep going down by hashing on items of the new itemset again because itemsets in C_k are all k-itemset. Hence, we do not need to process this new itemset anymore. If the reached node is a leaf node, we examine whether an itemset at the leaf node is a subset of transaction t. We add those itemsets of the leaf node, which are contained in t, into the answer set.

Example 2. Assume that we have a hash tree as presented in Figure 4, except for the support counters. Given a transaction {1, 3, 6}, we proceed as follows.

1. At the root node, apply the hashing function to each item in itemset {1, 3, 6}.
2. Travel down the left-hand branch with itemset {1, 3, 6} by hashing on item 1, as shown in Figure 6.
   i. At the next internal node, we apply the hashing function to each item starting from the second item, i.e., 3 or 6.
   ii. Hashing on item 3 will lead to the right-hand branch. We are now at a leaf node. Because the itemsets of this leaf node, i.e., {1, 3} and {1, 6}, are subsets of {1, 3, 6}, we add them to the answer set. The result is shown in Figure 7.
   iii. Hashing on item 6 leads again to the right-hand branch, that is, the same leaf node. We do not need to check any itemsets this time.
3. Return to the root node and apply the hashing function to the second item, 3, going down to the right-hand branch with the itemset {3, 6}. The branch contains only a leaf node. We add its itemset {3, 6}, which is contained in {3, 6}, to the answer set.
4. Return again to the root node and be ready to hash on the last item 6. Because $k = 2$ and $d = 1$, we have that the number of items behind item 6 in {1, 3, 6} is 0, which is less than $k - d = 1$. Hence, we do not need to hash on item 6 any more.

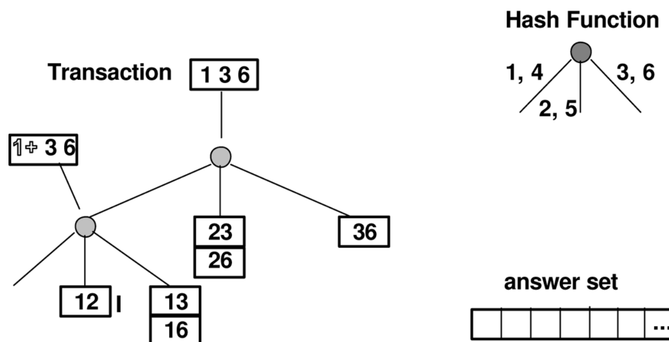


FIGURE 6 Result after hashing on item 1.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

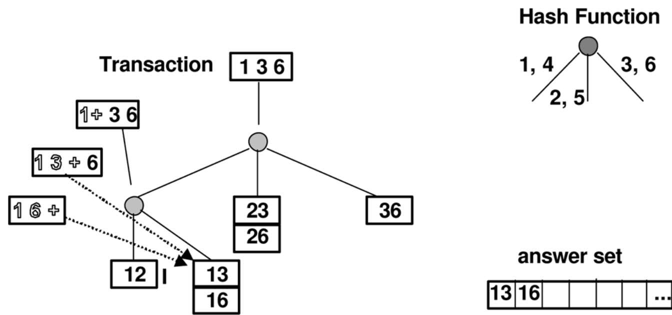


FIGURE 7 Result after finishing step 2.

5. Update the hash tree by incrementing the supports of the itemsets which are listed in the answer set. Figure 8 shows the final result after we have completed processing the transaction of {1, 3, 6}. In practice, the answer set contains references to the corresponding itemsets, instead of these itemsets themselves.

Summary

It is generally alleged that the *Apriori* algorithm based on the hash tree works well in terms of reducing the candidate set. However, if we are processing databases with many patterns, long patterns, or low support thresholds, we might have to generate many candidate itemsets and repeatedly scan the databases. Also, the hash tree can be improved in some ways, as shown in the following sections.

ITEMSET TREE

In a hash tree, every candidate itemset is stored in a leaf, and the leaf might be overflowed with too many itemsets. To improve the performance

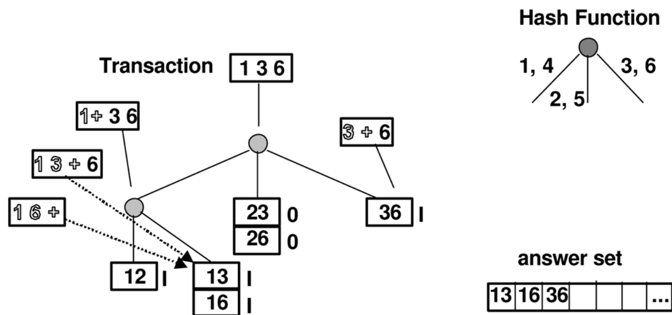


FIGURE 8 Result after completion of processing the transaction.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

of the hash tree, Brin et al. (1997) design a modified hash tree, that is the itemset tree, for keeping track of itemsets. To be more specific, an itemset tree is a prefix tree, where each edge is labeled with an item and each vertex represents an itemset composed of all the items which label the edges of the path from the root to the node. The root node represents the empty itemset. For the itemset tree, we require that all items in an itemset should be in a certain order, such as frequency-ascending, frequency-descending, or lexicographic orders. An itemset E is an extension of itemset I, if E has one more item than I and every item in E is the same as in I except for the last item in E. Let $I = \{i_1, \ldots, i_k\}$ be an itemset corresponding to a vertex v. Then, its extension E must be the form of $\{i_1, \ldots, i_k, i_{k+1}\}$, where $i_1, \ldots, i_k, i_{k+1}$ are increasingly sorted by their frequencies. For each extension E of itemset I, there must be an edge from I to E, labeled with i_{k+1}, and vice versa. What are included in a vertex depends on implementation of the *Apriori-like* algorithm. For example, in the *DIC* (dynamic itemset counting) algorithm proposed by Brin et al. (1997), itemsets are sorted in a frequency-ascending order, and each vertex in the itemset tree includes the fields as shown in Table 3.

In our implementation of the *Apriori* algorithm, we use a different structure for vertex fields as shown.

```
class CItemNode{                 // item node
public:
int            iid;             // item identifier
int            size;            // number of counters in the list
int            offset;          // offset of the items in the
                                   counter list
int            counters[ ]      // counter list
CItemsetNode   *parent;         // point to parent node
CItemsetNode   *next;           // point to next node on the same
level
......
}
```

TABLE 3 Fields Included in a Vertex for DIC Algorithm

Field name	Description
IID	Identifier of last item in the itemset the vertex represents
counter	Support counter for the represented itemset
Marker	Indicate start point of counting in the transaction database
fullyCountedFlag	Set to “1” when the itemset has been fully counted
SupportedFlag	Set to “1” when counter exceeds the minimum support
branches	Pointers to extensions of the vertex

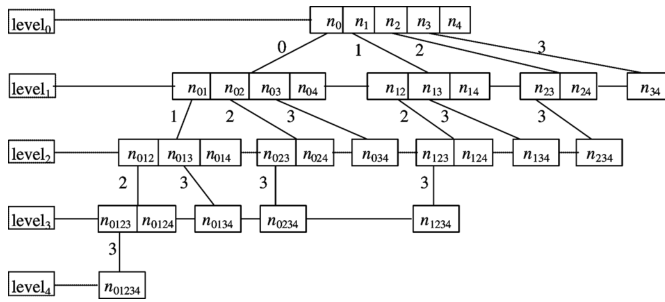


FIGURE 9 A complete itemset tree of the transaction database.

The counter of a vertex in Table 3 is moved up to its parent. Thus, each vertex in the itemset tree contains a list of counters, each of which records the support for an extension of the vertex, respectively. The itemset tree is created level by level. First, we calculate the set of frequent 1-itemsets, L_1, by scanning the transaction database, sort L_1 decreasingly by frequencies, and map items in L_1 into sequential integers, respectively, starting with 0. All items are then viewed as a sequence of non-negative integers. Level 0 can be generated therefore without difficulty. Level 1 is generated by the second scan of the database for all 2-itemsets. Processes for other levels are similar. Additionally, all nodes at the same level are linked together, as shown in Figure 9, to improve the performance.

FREQUENT PATTERN TREE

Construction of a prefix tree given in the previous sections requires repeated database scans. It is costly to process a huge database, especially when the database has a large amount of candidate itemsets. Han et al. (2000) proposed an extended prefix-tree structure, called the *frequent pattern tree* or *FP*-tree for short, in order to handle this problem. An *FP*-tree has two components. One is a prefix tree and the other is a frequent-item header table. Every node in the prefix tree is composed of a frequent-item identifier and a counter, except the root, which is labeled with "*null*" and needs no counter. Hence, each node is labeled with a frequent item identified by the frequent-item identifier. In other words, each node represents an itemset, which contains all items in the path from the root to the node. The counter holds the support for the itemset the node represents. All nodes labeled with the same identifier in the prefix tree are linked together as a linked item list. The second component of an *FP*-tree, i.e., the frequent-item header table, can actually be taken as an index for all frequent items, sorted by their frequencies. An entry for item i in the header table points to the corresponding linked item list for item i in the prefix tree.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

TABLE 4 The Transaction Database in Example 1

TID	Items bought	Ordered frequent itemset
100	ABCD	BCA
200	BCE	BCE
300	ABCE	BCEA
400	BE	BE
500	ACD	CA
600	BCE	BCE

Table 4 is the same transaction database used to demonstrate the association rule mining in Example 1. Figure 10 shows the *FP*-tree corresponding to this database, when the minimum support is specified as 50%.

To construct an *FP*-tree for a given transaction database *DB* and a specified minimum support threshold *minupp*, the first step is the same as that in the *Apriori* algorithm, that is, calculate the set of all frequent 1-itemsets L_1 by scanning the transaction database. Suppose that operator γ sorts a set of itemsets in a descending order by frequencies, and operator π is similar to the project operation in relational algebra. Each transaction in *DB* can also be viewed as a pattern of candidate frequent itemset, and *DB* can be called consequently a pattern base. Let $F = \gamma(L_1)$ and $PB = \gamma(\pi_F DB)$, where $\pi_F DB)$ is the projection of *DB* over *F*. Then *F* contains all frequent items, which are sorted in a frequency-descending order. *PB* is a sorted and projected pattern base reduced from *DB*. For example, given the transaction database *DB*, as shown in Table 4, and the minimum support $supp = 50\%$, we have that $L_1 = \{A, B, C, E\}, F = \gamma(L_1) = \{B, C, E, A\}$, and *PB* is shown in the right-hand column of Table 4.

After obtaining the ordered frequent-item list *F*, we begin to establish the frequent-item header table. For each item *A* in *F*, we create a linked item list, denoted as $H(A)$, which is initialized as an empty list.

Now with the header table and the new pattern-base *PB*, we begin to create the root, *T*, of the *FP*-tree, labeling *T* with "*null*." For each pattern

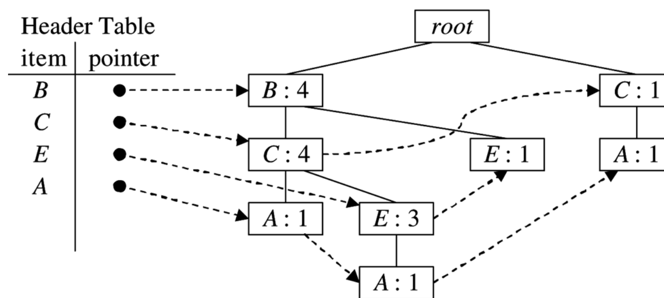


FIGURE 10 The *FP*-tree corresponding to Table 4.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

tuple $t \in PB$, assume that $t = \{A_1, A_2, \ldots, A_k\}$ and $insert(t, T)$ is the operation of inserting tuple t into a tree which has the root T. The insertion is processed as follows. If T already has a child node labeled with A_1, then we increase the counter of the child by 1; otherwise, we create a new child labeled with A_1, initialize its counter by 1, and append this new node to the end of the linked item list $H(A_1)$, which starts from the header table.

To complete the construction of the *FP*-tree, we call $insert\,(A_2, \ldots, A_k, A_1)$ recursively until no item is left in the pattern. In actual implementation, operation $\pi_F DB$ can be deferred and merged into the construction of *FP*-tree for *DB*. In other words, for each transaction $t \in DB$, we first project it over F, obtaining a pattern $\pi_F(t) \in PB$. Assume that $\pi_F(t) = A_1, A_2, \ldots, A_k$. Then the operation of $insert(\pi_F(t), T)$ is the same as $insert(t, T)$.

It can be seen easily that the database has been compressed into a highly condensed much smaller data structure, which can avoid costly, repeated database scans.

For any item A_i in F, there is a linked list $H\,(A_i)$, indexed at the header table. By traveling along this linked list, we can reach all the nodes labeled with A_i in the *FP*-tree. For each node labeled with A_i in the linked list, there is a path from the root to this node. All items in the path compose an itemset, i.e., a pattern, for which the support is equal to the number recorded in the counter of the node labeled with A_i. All these patterns compose a pattern base, called A_i's conditional pattern-base. In the *FP*-tree shown in Figure 10, for instance, there are two nodes labeled with *E*. Then, we have two patterns concerned with *E*. One is *{B, C}* with support 3. The other is $\{B\}$ with support 1. Thus, E's conditional pattern base is generated as shown in the middle column of Table 5. For item *A*, we can also generate *A's* conditional pattern base shown in the middle column of Table 6.

An *FP*-tree constructed for A_i's conditional pattern base is called A_i's conditional *FP*-tree. For example, we can construct *E*'s conditional *FP*-tree corresponding to *E*'s conditional pattern base shown in the middle column of Table 5. First, we should calculate the sorted frequent items set F in a same way. Apparently, we have $F = \{B, C\}$, when the minimum support threshold is still set as 50%. *E*'s conditional *FP*-tree is shown in Figure 11

TABLE 5 The Resulted Pattern-Base for Item E

TID	Pattern	Ordered frequent itemset
100	∅	∅
200	BC	BC
300	BC	BC
400	B	B
500	∅	∅
600	BC	BC

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

TABLE 6 The Resulted Pattern-Base for Item A

TID	Pattern	Ordered frequent itemset
100	BC	C
200	∅	∅
300	BCE	C
400	∅	∅
500	C	C
600	∅	∅

(a). Figure 11 (b) is *A*'s conditional *FP*-tree constructed for *A*'s conditional pattern base, as shown in the middle column of Table 6.

Now, the following algorithm, *FP*-growth, is used to mine all frequent itemsets based on the constructed *FP*-tree for the given transaction database *DB* and the minimum support threshold *minsupp* (Han et al. 2000).

Algorithm 1 *FP-growth*
Input: *FP-tree* T.
Output: all frequent itemsets.
Method: **call** *FP-growth*(T, ∅).
Procedure 1 *FP-growth*(T, α)
begin
  if (T is only a single path) **then**
    for each combination β of items in the path T **do**
    begin
    let *supp* = the minimum support of items in β;
    output *itemset* $\beta \cup \alpha$ with **support** of *supp*;
  end
  else
  for each item A_i in the header table of T **do**
  begin
    let $\beta \longleftarrow \{A_i\} \cup \alpha$;
    output β with support $A_i.support$;
    let *PB* $\longleftarrow$ β's conditional pattern-base generated from T;

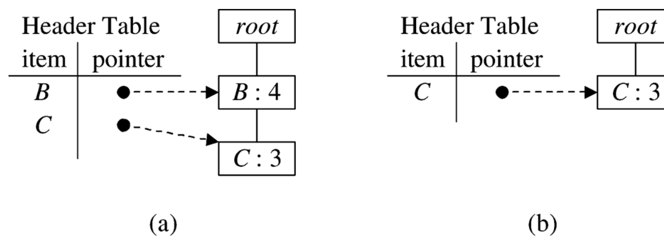


FIGURE 11 The *FP*-trees corresponding to Table 5 and Table 6.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

let $T_\beta \longleftarrow \beta$'s conditional FP-tree constructed from PB;
if $T_\beta \neq \emptyset$ **then call** FP - *growth*(T_β, β);
end
end.

In our implementation, we need to calculate the support for a given itemset. Let I be any itemset of transaction database DB. Without losing generality, we assume that $\gamma(I) = \{A_1, A_2, \ldots, A_k\}$. The following algorithm is designed to calculate the support for I based on *FP*-tree.

Algorithm 2 *FP-support*
Input: Itemset $I = \{A_1, \ldots, A_k\}$, in descending order by frequencies;
FP–tree T, constructed on DB and $minsupp$.
Output: the support $supp$ for I.
Method:
begin
let $supp \leftarrow 0$;
let $H \leftarrow$ the linked items list of A_k
starting from the header table in *FP*–tree T;
for each node $n \in H$ **do**
begin
let $P \leftarrow$ the path from the root to the node n;
let $J \leftarrow$ the set of all nodes in P;
if $I \subseteq J$ **then** $supp \leftarrow supp + n.counter$;
end
output *supp;*
end.

With the properties of *FP*-tree, we can show that this algorithm calculates correctly the support for the given itemset I. As claimed in Han et al. (2000), *FP-tree* T of DB contains the complete information of DB concerned with frequent patterns for the minimum support *supp*, and we can reach every node labeled with A_k in the *FP*-tree by traveling along the linked list H. Consequently, we have considered all patterns which contain item A_k in DB. Therefore, the algorithm is complete. Because A_k is the last item in itemset I, we can neglect those items after A_k in F. This is also why we only consider the path P. The soundness of algorithm *FP*-support is proved.

Let l be the maximum length of itemset and s the number of leaves in Tree T. Then the time complexity of Algorithm *FP*-support is $O(l^2 + ls)$.

GENERALIZED *FP*-TREE

As shown previously, the construction of an *FP*-tree starts with finding the set of all frequent 1-itemsets from a given transaction database. When

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

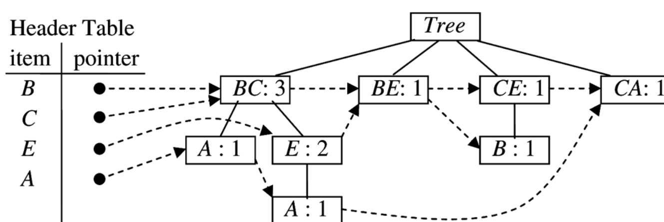


FIGURE 12 The 2-*FP*-trees based on Table 4.

the database is very dense, almost all items may be frequent and it is unnecessary to find the set of frequent 1-itemsets. On the other hand, when the database is large and sparse, an *FP*-tree might be inefficient. Sometimes we are only concerned with those itemsets with a specific length k. In this section, we expand the *FP*-tree to a general k-*FP*-tree, where k is a certain non-negative integer. The construction of a k-*FP*-tree is demonstrated next.

3. Example 1

Suppose that the transaction database is given in Table 4, and the minimum support minsupp = 0.5. The 2-FP-tree is shown in Figure 12.

After selecting an integer k for a given transaction database, $k \geq 0$, the collection of all frequent k-itemsets F and their supports are first generated by scanning the database once. Then F is considered as a conditional pattern base, and the patterns in F are sorted in support-descending order. In Example 2, we have $k = 2$ and $F = \{BC : 4,\ BE : 4,\ CE : 3,\ CA : 3\}$. Let K be the union of all k-itemsets in F. Thus, for each item e in itemset K, there exists at least an itemset I in F, such that e is in I. From the anti-monotonic *Apriori* property (Agrawal and Srikant 1994), each item e in K is frequent. We create and sort the items in K by using the following procedure. In Example 2, we obtain, $K = \{B,\ C,\ E,\ A\}$.

Procedure 2 *FIMerge*(F,K)
//Merge all items in F into K according to a certain order
begin
  let $n \leftarrow$ the number of itemsets in F;
  let $K \leftarrow \emptyset$;
  for $m \leftarrow 1$ **to** n **do**
    for (any item e in the mth itemset of F) **do**
      if $e \notin K$ **then** append e to the end of K;
end.

We then create the root of a k-*FP*-tree, *Tree,* and label it as "*null.*" For each itemset $I \in F$, a child of the root is created, its counter is initialized

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

with 0, and the node is labeled BY the itemset I. For each transaction T in the database, find out the first itemset J in F, such that $J \subseteq T$. Let $S = T - J$, project S over K, sort all the resulted items by the same order as that in K, and obtain R at last. Suppose that $R = lL$, where l is the first item of R and L is the list of remaining items in R. Insert R to the sub-tree J of *k-FP*-tree *Tree*, by calling a procedure *FP Insert*(lL,J). We also adjust the related counters in increments and the relevant item lists by a procedure *append*(), if necessary.

In Example 2, the first transaction is $T = \{ABCD\}$. We can easily see that $J = \{BC\}$ is the first itemset in F, which is contained in T. Hence, $S = \{AD\}$. Project S over K and obtain $R = \{A\}$, where $K = \{BCEA\}$. Other transactions are similarly processed recursively. A 2-*FP*-tree is constructed finally, as shown in Figure 12.

Algorithm Design

A formal description of construction algorithm for a *k-FP*-tree is given next.

Algorithm 3 *k-FP-tree_Constructor*
input: *DB*: a transaction database;
    minsupp: the minimum support;
    k: a non-negative integer;
output: *Tree:* a corresponding k-FP-tree;
begin
(1) **if** $k > 0$ **then**
  begin
  let $F \leftarrow$ the collection of all frequent k-itemsets
      and their supports;
  sort F in order of descending support;
  end
  else
    let $F \leftarrow \emptyset$;
(2) **call** *FIMerge(F, K)*;
(3) **create the root**, *Tree*, of a *k-FP-tree*;
(4) **for** $\forall$ itemset $I \in F$ **do**
  begin
    create a child of *Tree*, labelled as I;
    let $I.count \leftarrow 0$;
  end
(5) **for** $\forall$ transaction $T \in DB$ **do**
  begin
    let $J \leftarrow$ the first itemset $I \in F$ such that $I \subseteq T$;
    if $J \neq \emptyset$ **then**

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

```
        begin
          J.count++;
          for ∀ e ∈ J do
            if H(e) = NULL then
              append(H(e), J);
          let S ← T − J;
          sort S in the same order as that of K;
          let l ← the first item in S;
          let L ← S − {l};
          call FPInsert(lL, J);
        end
      end.
```

```
Procedure 3 FPInsert(lL, J)
//Insert itemset lL into tree J
begin
  if l is a child of J then
    l.count++;
  else
  begin
    create node l as a child of J;
    let l.count ← 1;
    append(H(l), l);
  end
  call FPInsert(L, l);
end
```

In this algorithm, $H(e)$ is the linked item list of e, which starts from the header table, and procedure $append(H(A), B)$ adds node B to the end of the linked item list $H(A)$.

From Algorithm 3, we can see that a *k-FP-tree* contains only the information of those frequent itemsets with their length equal to or larger than k. When $k = 0$, we do not need to scan the database for F. Instead, we directly have $K = F = \emptyset$. When $k = 1$, *k-FP*-tree is a normal *FP*-tree. Clearly, the information contained in a *k-FP*-tree is complete in the sense that all frequent itemsets can be obtained from the *k-FP*-tree, if and only if, $k < 2$.

Experimental Study and Analysis

The Connect-4 data set from the Irvine Machine Learning Database Repository is used to demonstrate the performance of our data structure (Blake and Merz 1998). Connect-4 contains 67557 records of legal positions and corresponding optimal results in the game of connect-4, with 129 items totally in the data set and 43 items on average in each transaction. Table 7

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

TABLE 7 Number Nodes at the Second Level for Connect-4

support	0.1	1	10	20	30	40	50	60	70	80	90
0-*FP*-tree	129	129	129	129	129	129	129	129	129	129	129
1-*FP*-tree	125	109	73	59	46	43	38	36	31	28	21
2-*FP*-tree	5793	4294	1986	1358	894	744	633	539	420	319	108

shows the number of direct children of the root for different minimum supports generated by Algorithm 3.

Based on Table 7, we can see that we would rather use 0-*FP*-tree than use 1-*FP*-tree if the minimum support is set to 1 or 0.1. For data set Connect-4 and most other support thresholds, 1-*FP*-tree is better than 0-*FP*-tree. Because Connect-4 is a rather dense data set, k-*FP*-tree is not considered here for $k > 1$.

Suppose that the total number of items is m and the total number of transactions is n for a given transaction database DB. Let the expected probability that an item appears in a transaction of DB be p. Then, the expected number of items that appear in a transaction is $m \cdot p$. The expected number of item-pairs that appear in a transaction is

$$\binom{m}{2} \cdot p^2 = \frac{m(m-1)}{2} \cdot p^2.$$

If

$$p < \frac{2}{m-1} \quad \text{or} \quad m - 1 < \frac{2}{p}, \tag{1}$$

Then,

$$\frac{m-1}{2} \cdot p \cdot mp < mp, \text{ that is, } \binom{m}{2} \cdot p^2 < mp.$$

This implies that when the database DB is so sparse that inequality 1 holds, the number of frequent 1-itemsets is probabilistically more than the number of frequent 2-itemsets. Consequently, 2-*FP*-tree is smaller than 1-*FP*-tree.

Generally, when

$$p < \frac{k}{k - m + 1}, \tag{2}$$

we have,

$$\binom{m}{k} \cdot p^k < \binom{m}{k-1} \cdot p^{k-1}.$$

Therefore, $(k-1)$-*FP*-tree is larger than k-*FP*-tree if condition (2) holds.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

CONCLUSIONS

We have studied systematically data structures used to implement the algorithms of association rule mining, including hash tree, itemset tree, and *FP*-tree (frequent pattern tree). This assists in better understanding existing association-rule-mining strategies.

In particular, a generalized *FP*-tree has been proposed in an applied context, designed specifically to identify false alerts in the stock market. We have discussed and analyzed experimentally the generalized *k-FP*-tree, and demonstrated that the generalized *FP*-tree reduces significantly the computation costs. The study in this paper will be useful to many association analysis tasks where one must provide really interesting rules and develop efficient algorithms for identifying association rules.

REFERENCES

Agarwal, R., C. Aggarwal, and V. Prasad. 2000. Depth first generation of long patterns. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 108–118, Boston, Massachusetts, USA.

Agarwal, R., C. Aggarwal, and V. Prasad. 2000. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing, Special Issue on High Performance Data Mining* 61(3):350–371.

Agrawal, R., T. Imielinski, and A. Swami. 1993. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (SIGMOD'93), pages 207–216, Washington, D.C. USA.

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. 1996. Fast discovery of association rules. In: *Advances in Knowledge Discovery and Data Mining*, eds. U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, 307–328. Menlo Park, CA: AAAI/MIT Press.

Aho, A., J. Hopcroft, and J. Ullman. 1976. *The Design and Analysis of Computer Algorithms*. Massachusetts: Addison-Wesley.

Blake, C. and C. Merz. 1998. *UCI Repository of Machine Learning Databases* [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

Brin, S., R. Motwani, and C. Silverstein. 1997. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 265–276, Tucson, Arizona, USA.

Brin, S., R. Motwani, J. Ullman, and S. Tsur. 1997. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 255–264, Tucson, Arizona, USA.

Burdick, D., M. Calimlim, and J. Gehrke. 2001. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Heidelberg, Germany.

El-Hajj, M. and O. Zaane. 2003. Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003)*, Washington D.C., USA.

Han, J. and Y. Fu. 1995. Discovery of multiple level association rules from large databases. In *Proceedings of 21st International Conference on Very Large Databases (VLDB'95)*, pages 420–431, Zurich, Switzerland.

Han, J., J. Pei, and Y. Yin. 2000. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12, Dallas, Texas, USA.

Downloaded By: [University of Technology Sydney] At: 08:03 7 May 2007

Huang, H., X. Wu, and R. Relue. 2002. Association analysis with one scan of databases. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 629–631, Maebashi City, Japan.

Liu, J., Y. Pan, K. Wang, and J. Han. 2002. Mining frequent item sets by opportunistic projection. In *Proceedings of the Eighth ACM SIGKDD*, pages 229–238, Edmonton, Alberta, Canada.

Park, J., M. Chen, and P. Yu. 1995. An effective hash based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 175–186, San Jose, California, USA.

Pei, J., J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. 2001. H-Mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01)*, pages 441–448, San Jose, California, USA.

Savasere, A., E. Omiecinski, and S. Navathe. 1995. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Databases (VLDB'95)*, pages 432–444, Zurich, Switzerland.

Sucahyo, Y. and R. Gopalan. 2003. CT-ITL: Efficient frequent item set mining using a compressed prefix tree with pattern growth. In *Proceedings of the 14th Australasion Database Conference (ADC2003)*, pages 95–104, Adelaide, Australia.

Tan, P., V. Kumar, and J. Srivastava. 2002. Selecting the right interestingness measure for association patterns. In *Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining*, pages 32–41, Edmonton, Alberta, Canada.

Toivonen, H. 1996. Sampling large databases for association rules. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB'96)*, pages 134–145, Mumbay, India.

Xu, Y., J. Yu, G. Liu, and H. Lu. 2002. From path tree to frequent patterns: A framework for Mining Frequent Patterns. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 514–521, Maebashi City, Japan.

Yan, X., C. Zhang, and S. Zhang. 2003. A database-independent approach of mining association rules with genetic algorithm. In *Proceedings of the Fourth International Conference on Intelligent Data Engineering and Automeated Learning*(IDEAL), pages 882–886, Hong Kong.

Yan, X., C. Zhang, and S. Zhang. 2004. Identifying software component association with genetic algorithm. *International Journal of Software Engineering and Knowledge Engineering: Special Issue on Data Mining for Software Engineering and Knowledge Engineering*, (accepted, forthcoming).

Zaki, M., S. Parthasarathy, M. Ogihara, and W. Li. 1997. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd International Conference on Knowledge Discovery in Databases (KDD'97)*, pages 283–286, Newport Beach, California, USA.