Information Sciences 179 (2009) 1724–1737

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

FIUT: A new method for mining frequent itemsets

Yuh-Jiuan Tsay [a,*], Tain-Jung Hsu [a], Jing-Rung Yu [b]

[a] *Department of Management Information Systems, National Ping-Tung University of Science and Technology, Ping-Tung, 912, Taiwan*
[b] *Department of Information Management, National Chi Nan University, Nantou 545, Taiwan*

A R T I C L E   I N F O

Article history:
Received 13 September 2005
Received in revised form 7 January 2009
Accepted 11 January 2009

Keywords:
Ultrametric trees
Frequent items ultrametric trees
Frequent itemsets
FP-growth

A B S T R A C T

This paper proposes an efficient method, the frequent items ultrametric trees (FIUT), for mining frequent itemsets in a database. FIUT uses a special frequent items ultrametric tree (FIU-tree) structure to enhance its efficiency in obtaining frequent itemsets. Compared to related work, FIUT has four major advantages. First, it minimizes I/O overhead by scanning the database only twice. Second, the FIU-tree is an improved way to partition a database, which results from clustering transactions, and significantly reduces the search space. Third, only frequent items in each transaction are inserted as nodes into the FIU-tree for compressed storage. Finally, all frequent itemsets are generated by checking the leaves of each FIU-tree, without traversing the tree recursively, which significantly reduces computing time. FIUT was compared with FP-growth, a well-known and widely used algorithm, and the simulation results showed that the FIUT outperforms the FP-growth. In addition, further extensions of this approach and their implications are discussed.

1. Introduction

Data mining is a process of discovering previously unknown and useful information from large databases. The most widely used data mining technologies include association rules discovery, clustering, classification, and sequential pattern mining [10,13]. Among them, the most popular technology is association rules discovery, which is mining the possibility of simultaneous occurrence of items, and then building relationships among them in databases. Association rules mining can be divided into two parts: find all frequent itemsets, and generate reliable association rules straightforward from all frequent itemsets. Because frequent itemsets mining is the most time-consuming procedure, it plays an essential role in mining association rules. The algorithms developed for mining frequent itemsets can be classified into two types: the first is the candidate itemsets generation approach, such as Apriori algorithm [2], called Apriori-like; another aspect is a method without candidate itemsets-generation approach, such as FP-growth algorithm [9], called FP-growth-like.

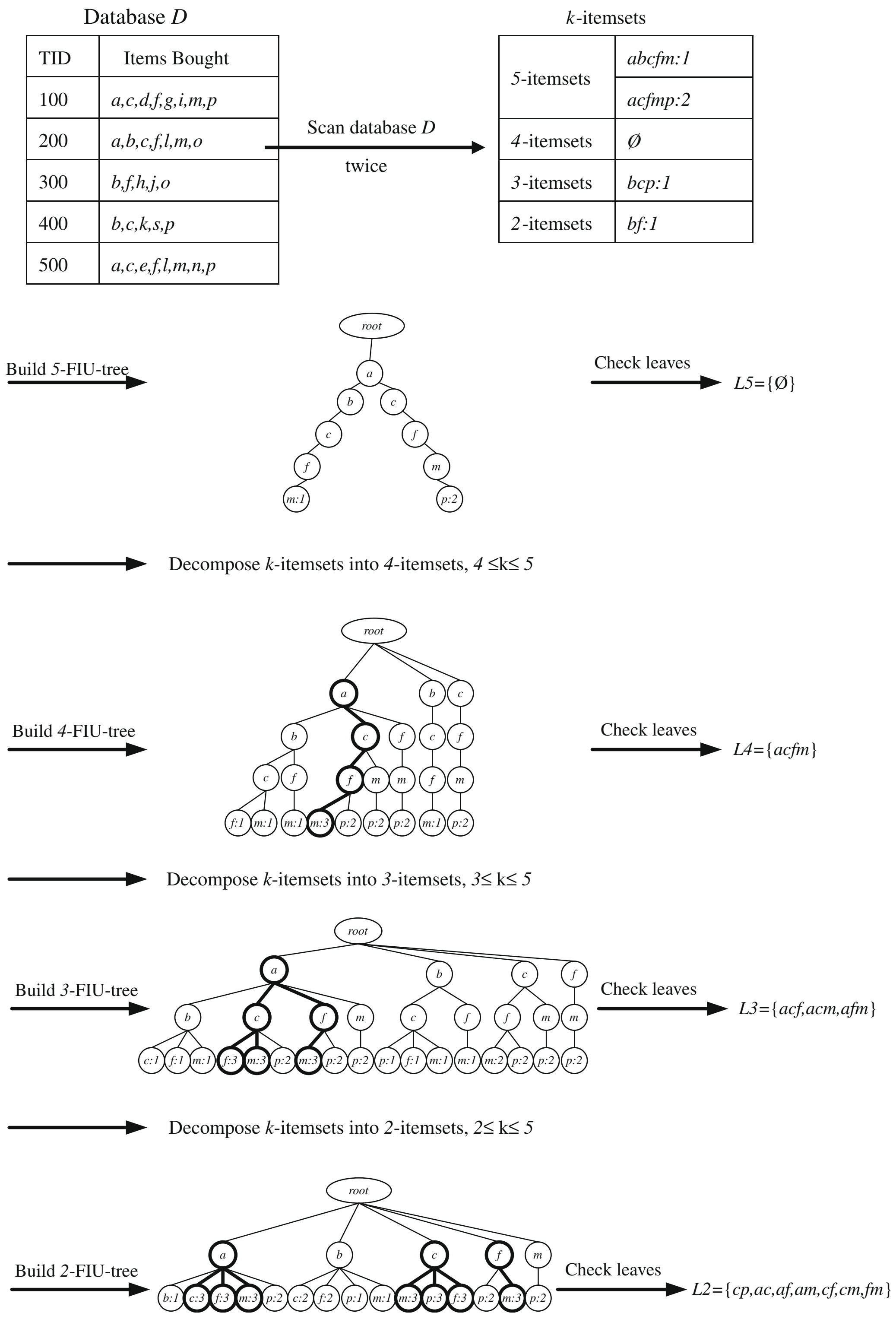
1.1. Apriori-like methods

The well-known Apriori algorithm [2] is the first proposed method to find frequent itemsets in a database. Some algorithms adopt an Apriori-like method, and are focused on reducing candidate itemsets, which in turn reduce the time required for scanning databases, are briefly described as follows. Park et al. [17] proposed an efficient Direct Hashing and Pruning (DHP) algorithm to control the number of candidate 2-itemsets and prune the size of the database by utilizing a hash technique. The inverted hashing and pruning (IHP) algorithm [11] is similar to the Direct Hashing and Pruning

* Corresponding author. Tel.: +886 8 7703202x7913; fax: +886 8 7740306.
E-mail address: yjtsay@mail.npust.edu.tw (Y.-J. Tsay).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.01.010

(DHP) algorithm [17] in the sense that both use hash tables to prune candidate itemsets. In DHP, every *k*-itemset within each transaction is hashed into a hash table. In IHP, the transaction identifiers of each item of the transactions that contain the item are hashed into a hash table associated with the item. The Tree-Based Association Rule (TBAR) algorithm [4]



Database *D*

TID	Items Bought
100	*a,c,d,f,g,i,m,p*
200	*a,b,c,f,l,m,o*
300	*b,f,h,j,o*
400	*b,c,k,s,p*
500	*a,c,e,f,l,m,n,p*

k-itemsets

5-itemsets	*abcfm:1*
	acfmp:2
4-itemsets	*Ø*
3-itemsets	*bcp:1*
2-itemsets	*bf:1*

Scan database *D*
twice
Build *5*-FIU-tree
Check leaves
L5={Ø}
Decompose *k*-itemsets into *4*-itemsets, *4* ≤k≤ *5*
Build *4*-FIU-tree
Check leaves
L4={*acfm*}
Decompose *k*-itemsets into *3*-itemsets, *3*≤ k≤ *5*
Build *3*-FIU-tree
Check leaves
L3={*acf,acm,afm*}
Decompose *k*-itemsets into *2*-itemsets, *2*≤ k≤ *5*
Build *2*-FIU-tree
Check leaves
L2={*cp,ac,af,am,cf,cm,fm*}
root

Fig. 1. An example.

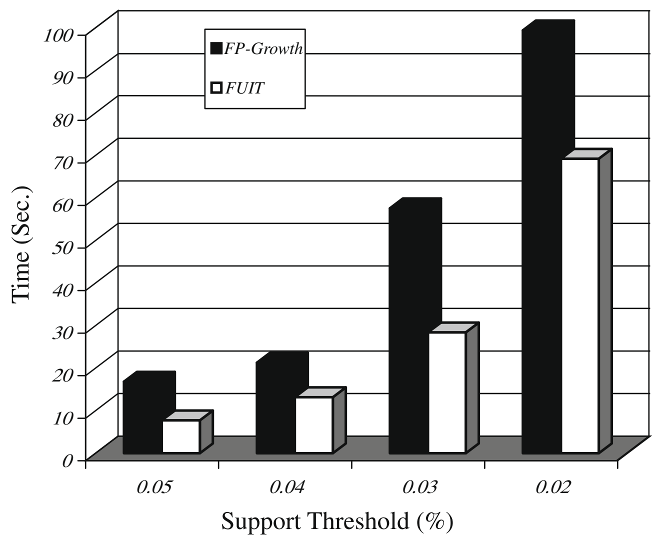


Fig. 2. Runtime on 10,000 transactions from FoodMart 2000.

employs an effective data-tree structure to store all itemsets to reduce the time required for database scans. It can also use other techniques, such as DHP. Cheung et al. [6] proposed Fast Distributed Mining (FDM) of association rules to efficiently discover frequent itemsets, which is a parallelization of the Apriori algorithm. At each level, a database scan is independently performed. Brin et al. [5] proposed the Dynamic Itemset Count (DIC) algorithm to locate frequent item sets, which uses fewer passes over the database than classic algorithms, and fewer candidate itemsets than methods based on sampling. Agarwal et al. [3] presented the TreeProjection method using the hierarchical structure of a lexicographic tree to project transactions at each node of the tree, and matrix counting on this reduced set of transactions for mining frequent itemsets. Another efficient method [19] uses the techniques of clustering transactions and decomposing larger candidate itemsets for mining frequent itemsets. Tsay et al. [20] proposed the Cluster-Based Association Rule (CBAR) method, which uses cluster tables to load the database into a main memory that requires only a single scan of the database. Its support count is performed on cluster tables, and thus, does not need to rescan the whole database. The Efficient Dynamic Database Updating Algorithm (EDUA) [21] is designed for mining dynamic databases when some data are deleted. A special database structure BitTableFI [7] is used horizontally and vertically to compress the database for quickly generating candidate itemsets and counting support.

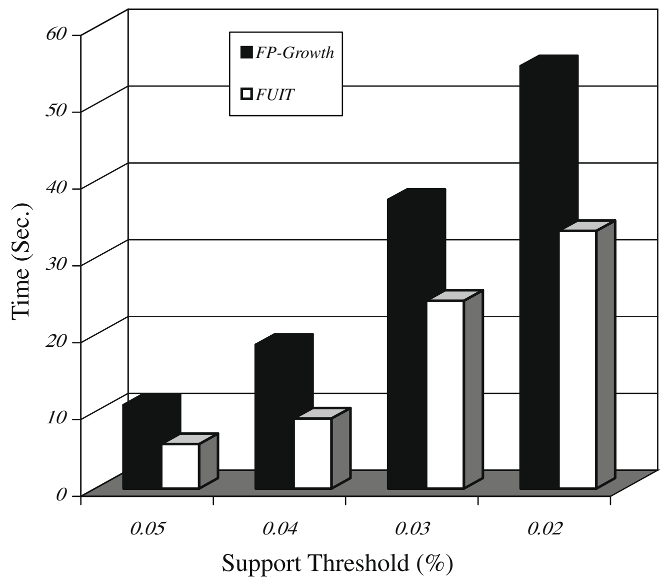


Fig. 3. Runtime on 15,000 transactions from FoodMart 2000.

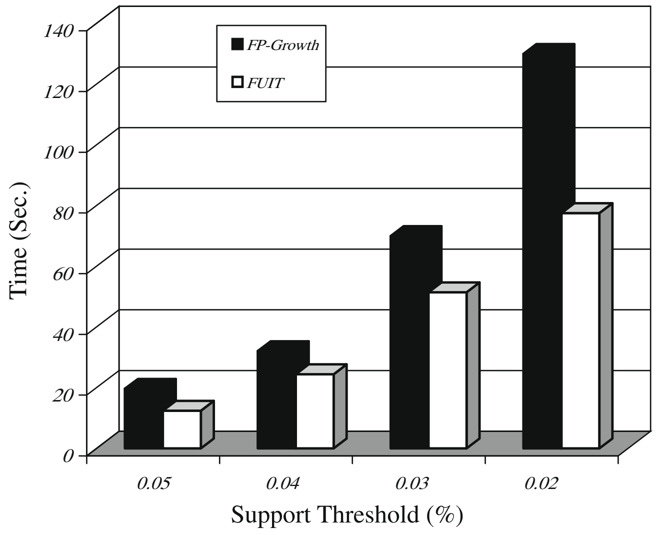


Fig. 4. Runtime on 20,000 transactions from FoodMart 2000.

1.2. FP-growth-like methods

Han et al. [9] proposed the FP-growth method to avoid generating candidate itemsets by building a FP-tree with only two scans over the database. This milestone development of frequent itemsets mining avoids the costly candidate itemsets generation phase, which overcomes the main bottlenecks of the Apriori-like algorithms. Some algorithm analogy for FP-growth without generating candidate itemsets is briefly described as follows. The H-mine method [15] uses a memory-based hyper structure to store a sparse database in the main memory, and builds an H-structure to invoke FP-growth in mining dense databases. An inverted matrix approach uses an inverted matrix to store the transactions in a special layout, then builds and mines relatively small structures, which are called COFI-trees [8]. The CFP-tree structure [14] is designed to store pre-computed frequent itemsets on a disk to save space. A CFP-tree stores discovered frequent itemsets, but a FP-tree stores transactions. They both use prefix and suffix sharing in the CFP-tree, but only prefix sharing in the FP-tree. Maximum length frequent itemsets are generated by adapting a pattern fragment growth methodology [12] based on the FP-tree structure.

This paper proposes an efficient FIUT algorithm, based on FIU-tree structure, to identify frequent itemsets. The FIUT algorithm is similar to FP-growth-like method. It first scans the database twice to locate all frequent items and prunes infrequent items from the database. Meanwhile, it divides the transactions into k-itemsets in the pruned database. Next, new k-itemsets are collected by decomposing each h-itemset into k-itemsets, where $h \geqslant k + 1$, and union original k-itemsets. Then, the

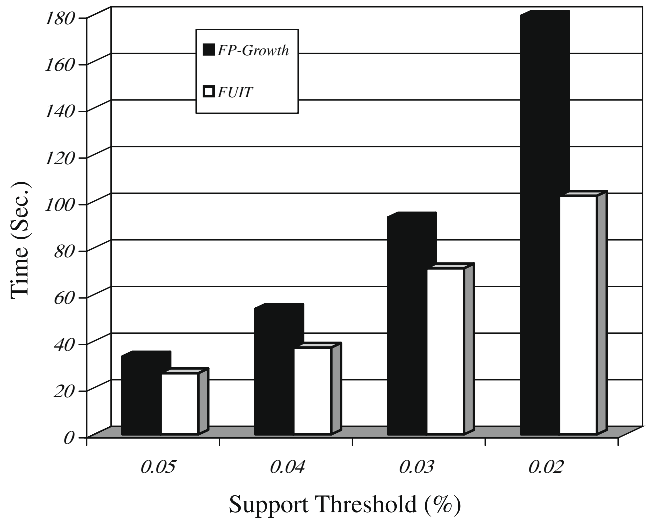


Fig. 5. Runtime on 30,000 transactions from FoodMart 2000.

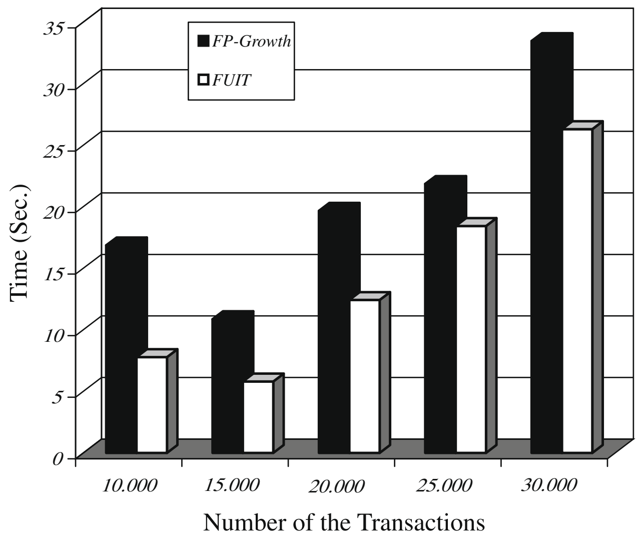


Fig. 6. Runtime of minimum support threshold 0.05%.

k-FIU-tree is constructed based on the k-itemsets. Next, all k-FIU-trees are searched to generate all frequent k-itemsets by checking the leaves of k-FIU-tree only. The FIUT proposed by this study scans the databases only twice, which solves the problems of time-consuming iteratively scans of databases, and the excessive candidate itemsets generated by Apriori. All frequent itemsets are generated by checking the leaves of each FIU-tree, in order to overcome the weakness of recursively traversing the FP-tree in the FP-growth.

1.3. Outline of the paper

The remainder of the paper is organized as follows: Section 2 provides an overview for mining frequent itemsets and briefly describes the FP-growth method [9], which is a more efficient algorithm of association rules recently proposed. Section 3 provides definitions for this mining method, and a detailed introduction on the FIUT algorithm in mining frequent itemsets. An example is demonstrated in Section 4. In Section 5, the experiments and performance analysis are shown. Section 6 offers conclusions.

2. Background

2.1. Definition of association rule

In what follows, we introduce the notion of an association rule. Let $I = \{i_1, i_2, i_3, \ldots, i_N\}$ represents a set of N distinct data items. Generally, a set of items is called an itemset, and an itemset with k items is denoted as a k-itemset. Database D is a set of transactions, where the ith transaction T_i denotes a set of items, such as $T_i \subseteq I$. $|D|$ is the total number of transactions in D, and $|T_i|$ is the number of distinct items in transaction T_i. Each transaction is associated with a unique identifier, which is termed as *TID*. An association rule is an implication of the form $X \to Y$, where $X, Y \subseteq I$, and $X \cap Y = \phi$. There are measures of quality for each rule in support of itemset $X \cup Y$ and confidence of rule $X \to Y$. First, we need to calculate the support of itemset $X \cup Y$, which is the ratio (denoted by s%) of the number of transactions that contain the $X \cup Y$ to $|D|$. Next, the confidence of rule $X \to Y$ is the ratio (denoted by c%) of the number of transactions containing $X \cup Y$ to the number of transactions that contain X in database D. The problems of association mining rules from database D can be processed in two important steps: (1) locate all frequent itemsets whose supports are no less than the user-defined *minimum support threshold* ξ, where $\xi \in (0, 1]$, and, (2) obtain association rules directly from these frequent itemsets with confidences no less than the user-defined *minimum confidence threshold*. The most time-consuming part of mining association rules is to discover frequent itemsets. Therefore, the remainder of this paper focuses on obtaining frequent itemsets.

2.2. The Apriori algorithm

In conventional Apriori-like methods, the level wise process of identifying sets of all frequent itemsets is in a combination of smaller, frequent itemsets [1–7,11,17–20]. In the kth level, the Apriori algorithm identifies all frequent k-itemsets, denoted as L_k. C_k is the set of candidate k-itemsets obtained from L_{k-1}, which are suspected frequent k-itemsets. For each transaction in D, the candidate k-itemsets in C_k contained within the transaction are determined, and their support count is

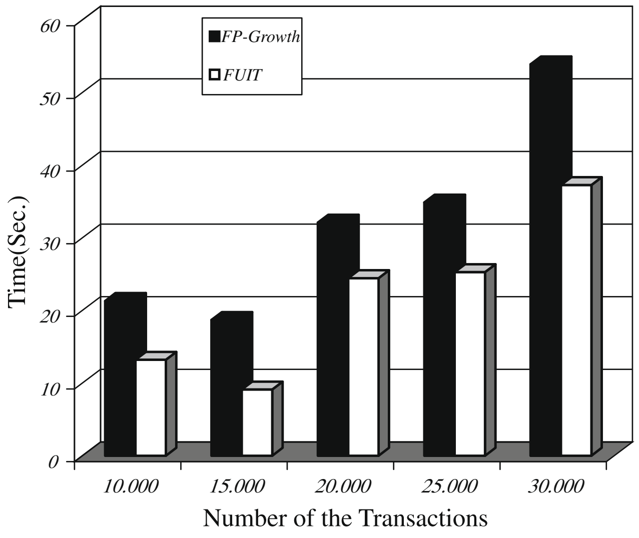


Fig. 7. Runtime of minimum support threshold 0.04%.

increased by $1/|D|$. Following scanning (reading) and contrasting with the entire D, when the supports of candidate k-itemsets are greater than or equal to user-defined minimum support threshold ξ, they immediately become frequent k-itemsets. At the end of level k, all frequent itemsets of length k or less have been discovered. During the execution, numerous candidate itemsets are generated from single itemsets, and each candidate itemset must perform contrasts on the entire database, level by level, while searching for frequent itemsets. However, the performance is significantly affected because the database is repeatedly read to contrast each candidate itemset with all transaction records of the database.

2.3. The FP-growth algorithm

Several algorithms have been proposed by creating various tree-structures for mining frequent itemsets [8,9,12,14,15]. The most famous key algorithm for enumerating frequent itemsets is FP-growth [9], which procedures are as follows: (1) find the frequent items and their frequencies by scanning the whole D database, and sort items of frequency in descending order for each transaction; (2) construct a novel FP-tree structure, which is a large database compressed into a highly condensed FP-tree; (3) use the FP-tree structure to build conditional trees recursively to mine frequent itemsets that are of the same order of magnitude in number as the frequent patterns.

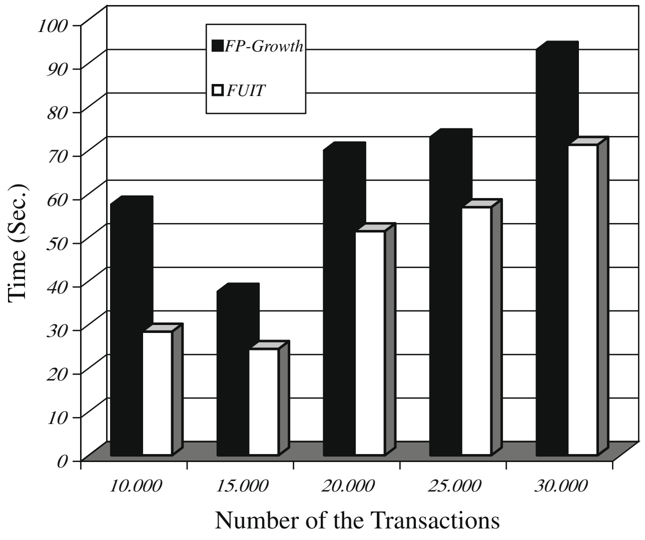


Fig. 8. Runtime of minimum support threshold 0.03%.

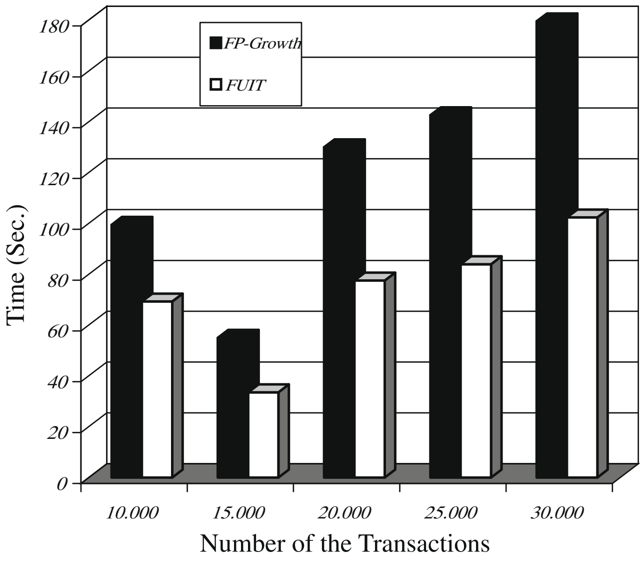


Fig. 9. Runtime of minimum support threshold 0.02%.

The FP-growth has three main characteristics: (1) it uses the FP-tree, which is usually smaller than the original database, thus is space-efficient; (2) it applies a pattern growth method to avoid the costly candidate generation phase by concatenating frequent items in the FP-tree; (3) a partitioning-based divide-and-conquer search technique is proposed in mining frequent itemsets. However, the bottleneck of FP-growth is the reduction of traversal and construction cost of the FP-tree and its conditional FP-trees. Thus, there are two major disadvantages of the FP-growth. First, it is not feasible to construct a main memory-based FP-tree if the database is large and sparse. Secondly, it needs to build a conditional pattern base and conditional sub-FP-trees for each shorter pattern, in order to search for longer patterns, and thus the acquisition of results are a very time and space consuming operations as the number of recursions and patterns increases.

3. Preliminaries

This section introduces basic definitions of the FIUT approach.

Definition 1. A *path* from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ without repeating nodes.

Definition 2. Denoting an itemset $PathItem(v_s) = \{v_1, v_2, \ldots, v_s\}$ by traversing $v_1, v_2, \ldots, v_s$ of nodes in a path, beginning with the child v_1 of the root and ending with leaf v_s in a tree.

Definition 3. An ultrametric tree is a rooted tree, and all of its leaves are at the same height [16].

Definition 4. A k-itemset is obtained by pruning infrequent items from a transaction that contains k frequent items.

Definition 5. A frequent item ultrametric tree (FIU-tree) is a tree structure, as constructed below.

(1) The root is labeled as "*null*", then an itemset $\{v_1, v_2, \ldots, v_s\}$ of frequent items is inserted as a path, beginning with the child v_1 of the root and ending with leaf v_s in the tree.
(2) A FIU-tree is constructed by inserting all itemsets as its paths and each itemset contains the same number of frequent items.
(3) Each leaf in the FIU-tree consists of two fields, named *item-name* and *count*(*item-name*). The non-leaf nodes in the FIU-tree consist of two fields, named *item-name* and *node-link*, where *item-name* registers the frequent item this node represents, *count*(*item-name*) registers the number of transactions containing the itemset *PathItem*(*item-name*), and *node-link* links to the next node in the FIU-tree.

Definition 6. A FIU-tree is denoted k-FIU-tree if it is constructed by all k-itemsets in the database.

Corollary 1. *A k-FIU-tree is rooted tree, whose leaves are at the same height k.*

4. Frequent items ultrametric trees method: FUIT

The FIUT (frequent items ultrametric trees) method employs an efficient ultrametric tree structure, known as the FIU-tree, to present frequent items for mining frequent itemsets. The proposed FIUT consists of two main phases within two scans of database D. Phase 1 starts by computing the support for all items occurring in the transaction records. Then, a pruning technique is developed to remove all infrequent items, leaving only frequent items to generate the k-itemsets, where the number of frequent items of a transaction is k in a database. Meanwhile, all the frequent 1-itemsets are generated. Phase 2 is the repetitive construction of small ultrametric trees, the actual mining of these trees, and their release. The approach for mining frequent k-itemsets, as proposed in this study, first builds an independent, relative k-FIU-tree for all k-itemsets, where k from M down to 2, and M denotes the maximal value of k among the transactions in the database. Then, each of the trees is mined separately, without generating candidate k-itemsets. The k-FIU-tree is discarded immediately after frequent k-itemsets are mined. At any given time, only one k-FIU-tree is present in the main memory. Algorithm 1 illustrates the algorithmic form of the FIUT. Contrary to the FP-growth, this method is able to overcome the main drawback of the FP-growth method, which searches and traverses recursively in conditional FP-trees.

Algorithm 1. Main program for the FIUT algorithm.

```
Algorithm FIUT ($D$, $\xi$)
Input: A transaction database $D$, where the items in each transaction are lexicographically sorted, and a minimum support threshold $\xi \in (0, 1]$.
Output: All sets of frequent $k$-itemsets $L_k$, where $1 \leqslant k \leqslant M$.
Begin
1) $h$-itemsets = $k$-itemsets generation ($D$, $MinSup$);
2) for ($k = M$ down to 2) do {
3)    $k$-FIU-tree = $k$-FIU-tree generation ($h$-itemsets);
4)    frequent $k$-itemsets $L_k$ = frequent $k$-itemsets generation ($k$-FIU-tree);
5) }
   End
```

4.1. Generating the frequent items and k-itemsets

Phase 1 requires two scans of database D. The goal of this phase is to generate all frequent items (frequent 1-itemsets), then prune infrequent items from each transaction, and denote as a k-itemset only if it contains k frequent items, $2 \leqslant k \leqslant M$, where M denotes the maximal value of k among the transactions. More importantly, the frequent items in each transaction retain their lexicographical order. In the first scan, only frequent items play a role in frequent itemsets mining, it is necessary to perform scan D once to identify the set of frequent items, that is, the frequent 1-itemsets, L_1. This scan begins by enumerating the items appearing in the transactions. After reading the whole database, infrequent items are eliminated and the remaining frequent items are stored in lexicographical order, that is, the frequent 1-itemsets, L_1. In the second scan, for each transaction, infrequent items are pruned and denoted as k-itemsets if the number of frequent items is k, $2 \leqslant k \leqslant M$. Based on the k-itemsets construction process, we offer the following k-itemsets construction procedure, as shown in Algorithm 2.

Algorithm 2. Generating the frequent items L_1 and k-itemsets.

```
Algorithm k-itemsets generation ($D$, $\xi$)
Input: A transaction database $D$, and a minimum support threshold $\xi \in (0,1]$.
Output: A set of frequent items (1-itemsets) $L_1$, all $k$-itemsets by pruning the infrequent items from transactions, and $M$ of the maximal value $k$.
Begin
(1) Scan the transaction database $D$ once, collect the frequent items (1-itemsets) $L_1$;
(2) $M = 0$;
(3) for (each transaction $t \in D$) do {
(4)    prune the infrequent items from $t$;
(5)    put $t$ into $|t|$-itemsets;
(6)    if ($|t| > M$) then
(7)       $M = |t|$;
(8) }
  End
```

4.2. Frequent items ultrametric tree (FIU-tree): constructing and mining

Phase 2 is the repetitive construction of the k-FIU-tree, and the mining of all the frequent k-itemsets based on the leaves of k-FIU-tree, where k is from M down to 2. The k-FIU-tree construction begins by decomposing each h-itemset into k-itemsets, where $k + 1 \leqslant h \leqslant M$, and union original k-itemsets. Meanwhile, these k-itemsets are used to construct the k-FIU-tree as follows. One root is first created for k-FIU-tree, labeled with "*null*". Second, for each k-itemset, check if the first frequent item exists as one of the children of the root, then denote the child as a temporary 1st root. If it does not exist, then add a new node for this item as a child of the root node, and denote it as a temporary 1st root. Third, for the *s*th frequent item of the *k*-itemset, where *s* is from 2 to $k-1$, check if the *s*th frequent item exists as one of the children of the temporary $(s-1)$th root, then denote the child as a temporary *s*th root. If it does not exist, then add a new node for this item as a child of the temporary $(s-1)$th root and denote it as a temporary *s*th root. This process is repeated until the *k*th frequent item is determined. If one of the children of the temporary $(k-1)$th root exists, increase support for the node, since the node is a leaf of the *k*-FIU-tree. Otherwise, add a new node for the corresponding item as a child of the temporary $(k-1)$th root node, and the new node is a new leaf, with 1 as support in the *k*-FIU-tree. This is a straightforward implementation of the algorithm, as presented in Algorithm 3.

Algorithm 3. Building the *k*-FIU-tree.

```
Algorithm k-FIU-tree generation (h-itemsets)
Input: All h-itemsets, where $k \leqslant h \leqslant M$.
Output: A k-FIU-tree.
Begin
(1) Create the root of a k-FIU-tree, and label it as "null" (temporary 0th root);
(2) for ($k+1 \leqslant h \leqslant M$) do {
(3)    decompose each h-itemset into all possible k-itemsets, and union original k-itemsets;
(4)    for (each k-itemset) do {
(5)       for ($1 \leqslant s \leqslant k-1$) do {
(6)          if (sth frequent item $\in$ one of the children of the temporary $(s-1)$th root) then
(7)             denote the child as temporary sth root;
(8)          else
(9)             add a new node for this item as a child for the temporary $(s-1)$th root,
                and denote the new node as the temporary sth root;
(10)      }
(11)      if (kth frequent item $\in$ the children of the temporary $(k-1)$th root) then
(12)         increment the support for the leaf node;
(13)      else
(14)         add a new node for this item as a child of the temporary $(k-1)$th root,
             and the new node is a new leaf with 1 as support;
(15)   }
(16) }
End
```

The frequent *k*-itemsets are mined by checking the *count*(*item-name*) value of each leaf in *k*-FIU-tree, without candidate generation and in contrast with the entire database. If the value *count*(*item-name*)/$|D|$ is greater than or equal to the user-specified minimum support threshold ξ, then the *PathItem*(*item-name*) automatically becomes a frequent *k*-itemset, L_k. Algorithm 4 illustrates the method for mining frequent *k*-itemset, L_k. This iterative process is repeated until the value of *k* is equal to 2.

Algorithm 4. Mining the frequent *k*-itemsets from *k*-FIU-tree.

```
Algorithm frequent k-itemsets generation (k-FIU-tree)
Input: A k-FIU-tree.
Output: A set of frequent k-itemsets $L_k$.
Begin
(1) for (each leaf with item name v of k-FIU-tree) do {
(2)    if ($count(v)/|D| \geqslant \xi$) then
(3)       put PathItem (v) into frequent k-itemsets $L_k$;
(4) }
End
```

4.3. An example of FIUT method

To explain each phase of the FIUT in details, an example is shown below. The example includes 5 transactions in database *D*, as shown in Fig. 1. The user-defined minimum support threshold is equal to 60%.

In Phase 1, a set of frequent 1-itemsets is found by the first scan of the database. In addition, the second scan of database *D* finds the *k*-itemsets, $2 \leqslant k \leqslant 5$, 2-itemset = {(*bf*:1)}, 3-itemset = {(*bcp*:1)}, 4-itemset = $\emptyset$, and 5-itemsets = {(*abcfm*:1), (*acfmp*:2)}, as shown in Fig. 1.

Phase 2 constructs *k*-FIU-tree and discovers frequent *k*-itemsets, L_k, where *k* ranges from 5 down to 2, as shown in Fig. 1.

(1) When $k = 5$, only insert 5-itemsets (*abcfm*:1) and (*acfmp*:2) into the 5-FIU-tree. First, the root of 5-FIU-tree can be created, and labeled "*null*". Then, for the 5-itemsets (*abcfm*:1), five nodes are created, *a* is a temporary 1st root of *b*, *b* is a temporary 2nd root of *c*, *c* is a temporary 3rd root of *f*, *f* is a temporary 4th root of *m*, and *m* is a leaf in path *abcfm* with

count(*m*) = 1. For the 5-itemsets (*acfmp*:2), the first item *a* shares the child node of the root with path *abcfm*, and a new branch is formed starting from the temporary 1st root *a* of *c*; *c* is a temporary 2nd root of *f*, *f* is a temporary 3rd root of *m*, *m* is a temporary 4th root of *p*, and *p* is a leaf in path *acfmp* with *count*(*p*) = 2. Finally, both ration values of *count*(*m*)/|*D*| = 1/5 = 20% and *count*(*p*)/|*D*| = 2/5 = 40% is below the user-defined minimum support threshold 60%, thus $L_5 = \emptyset$.

(2) When $k = 4$, decompose the 5-itemsets (*abcfm*:1 and *acfmp*:2) into 4-itemsets, and union the original 4-itemset (∅). These 4-itemsets are (*abcf*:1, *abcm*:1, *abfm*:1, *acfm*:1, *bcfm*:1, *acfm*:2, *acfp*:2, *acmp*:2, *afmp*:2, and *cfmp*:2). Meanwhile, these 4-itemsets are used to construct a 4-FIU-tree, and L_4 = {*acfm*}, by checking each leaf of 4-FIU-tree.

(3) When $k = 3$, decompose the 4-itemset (∅), and 5-itemsets (*abcfm*:1 and *acfmp*:2) into 3-itemsets, and union the original 3-itemset (*bcp*:1). These 3-itemsets are (*bcp*:1, *abc*:1, *abf*:1, *abm*:1, *acf*:1, *acm*:1, *afm*:1, *bcf*:1, *bcm*:1, *bfm*:1, *cfm*:1, *acf*:2, *acm*:2, *acp*:2, *afm*:2, *afp*:2, *amp*:2, *cfm*:2, *cfp*:2, *cmp*:2, and *fmp*:2). Meanwhile, these 3-itemsets are used in constructing the 3-FIU-tree and L_3 = {*acf, acm, afm, and cfm*} by checking each leaf of 3-FIU-tree.

(4) When $k = 2$, decompose the 3-itemset (*bcp*:1), 4-itemset (∅), and 5-itemsets (*abcfm*:1 and *acfmp*:2) into 2-itemsets, and union the original 2-itemset (*bf*:1). These 2-itemsets are (*bf*:1, *bc*:1, *bp*:1, *cp*:1, *ab*:1, *ac*:1, *af*:1, *am*:1, *bc*:1, *bf*:1, *bm*:1, *cf*:1, *cm*:1, *fm*:1, *ac*:2, *af*:2, *am*:2, *ap*:2, *cf*:2, *cm*:2, *cp*:2, *fm*:2, *fp*:2, and *mp*:2). Meanwhile, these 2-itemsets are used in constructing the 2-FIU-tree and L_2 = {*cp*, *ac*, *af*, *am*, *cf*, *cm*, *fm*} by checking each leaf of 2-FIU-tree.

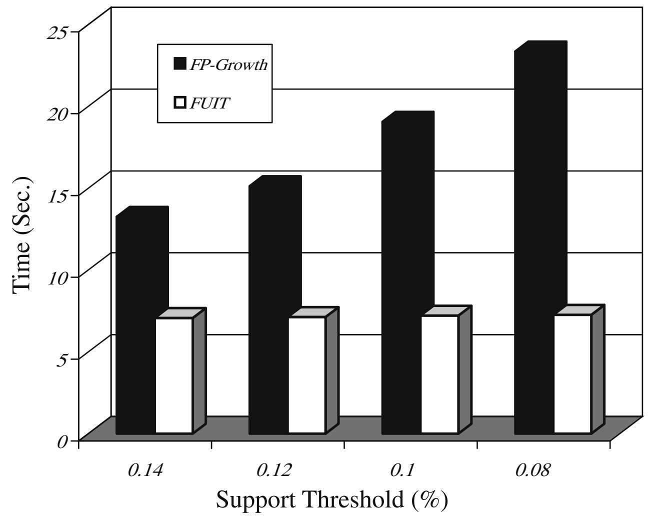


Fig. 10. Runtime on 10,000 transactions from T10I4D100K.

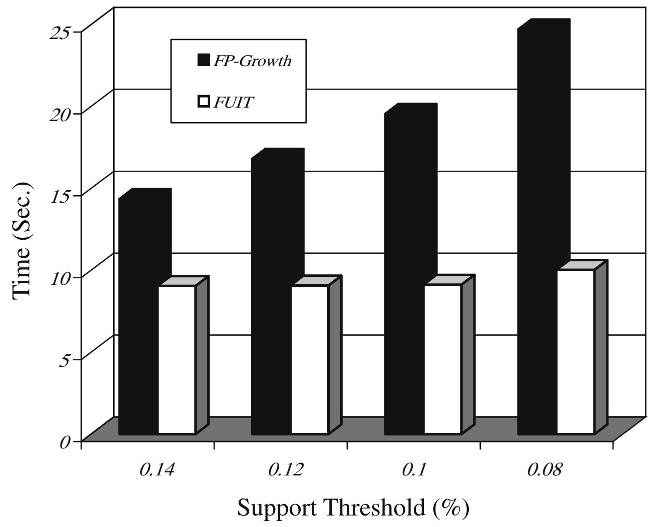


Fig. 11. Runtime on 15,000 transactions from T10I4D100K.

5. Experiments

In order to evaluate the efficiency of the FIUT method, several experiments were conducted to compare with the well-known FP-growth algorithm [7]. All the experiments were executed on a 1.60 GHz Pentium PC machine with 512 MB of main memory, running on a Windows 2000 operating system. All the programs were written in C++. The first testing case was from the FoodMart 2000 transaction database provided by a Microsoft SQL Server 2000, and the second testing case was from T10I4D100K database, provided by the IBM Almaden Quest research group (http://fimi.cs.helsinki.fi/data/). In order to compare with FP-growth, the run time used here indicates the total execution time, that is, the period between input and output [9]. Two types of experiments were designed to test the effects on run time by changing the minimum support threshold and the number of transactions.

5.1. Experiment 1

The comparison results were generated from the FoodMart 2000 transaction database provided by a Microsoft SQL Server 2000, as shown in Figs. 2–9.

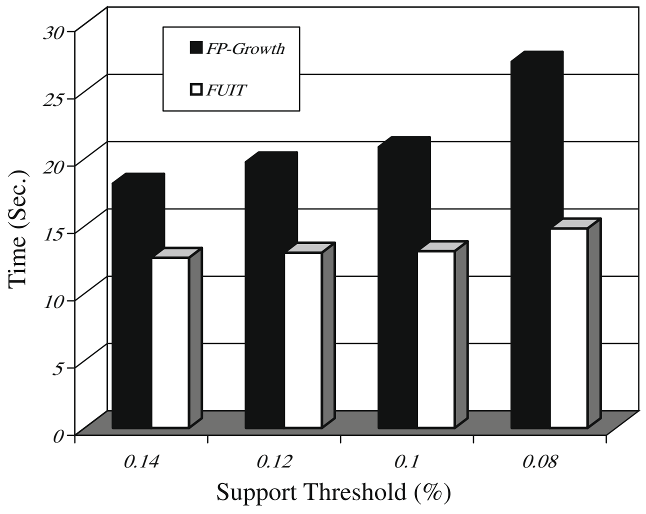


Fig. 12. Runtime on 20,000 transactions from T10I4D100K.

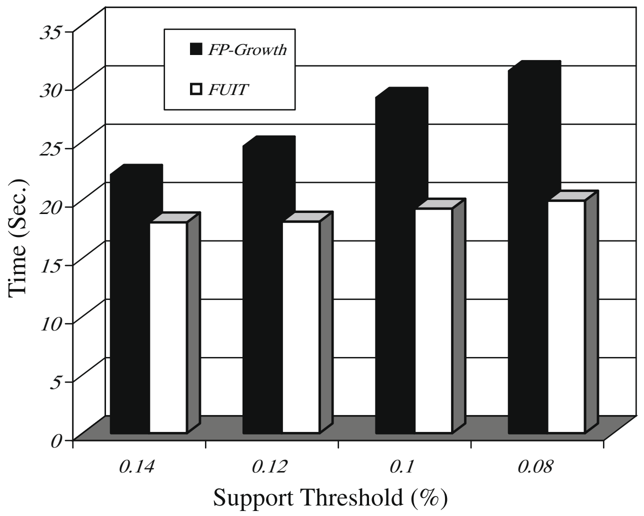


Fig. 13. Runtime on 30,000 transactions from T10I4D100K.

In the first type, we evaluated the performance of the FIUT and the FP-growth method on the same data set with various minimum support thresholds, including 0.05%, 0.04%, 0.03%, and 0.02%. The experimental results of the second test are shown in Figs. 2–5. As seen, both the execution time of the FIUT and the FP-growth increase when the value of minimum support decreases.

In the second type, we compared the relative performance of the two methods on four data sets under the same minimum support threshold. The numbers of transactions in five data sets were 10,000, 15,000, 20,000, 25,000, and 30,000, respectively. The maximal length of transaction was 20, total number of items was 3568, and the maximal length of frequent itemset was 12. As shown in Figs. 6–9, both the execution time of the FIUT method and the FP-growth increased when the number of transactions was increased. However, FIUT used less time than the FP-growth in both tests.

5.2. Experiment 2

The comparison results were generated from the T10I4D100K database (http://fimi.cs.helsinki.fi/data/), provided by IBM Almaden Quest research group, as shown in Figs. 10–17.

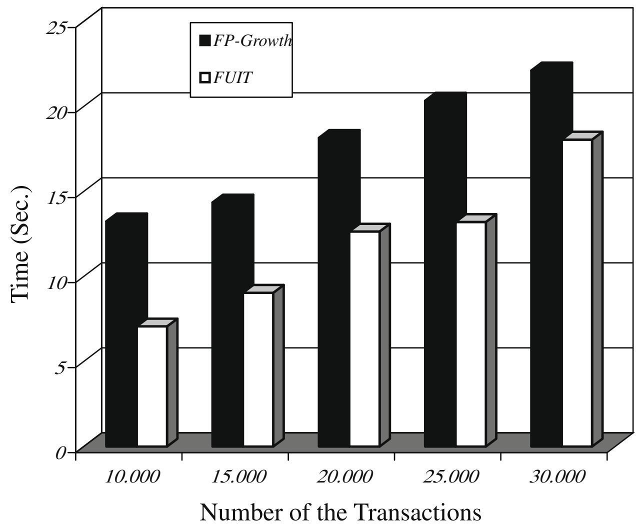


Fig. 14. Runtime of minimum support threshold 0.14%.

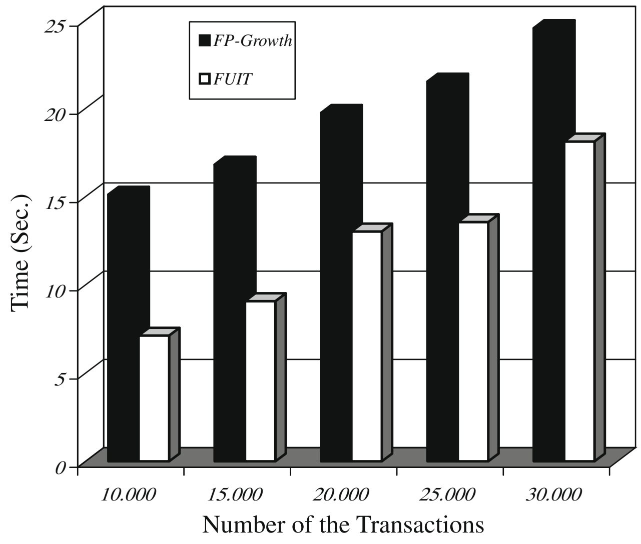


Fig. 15. Runtime of minimum support threshold 0.12%.

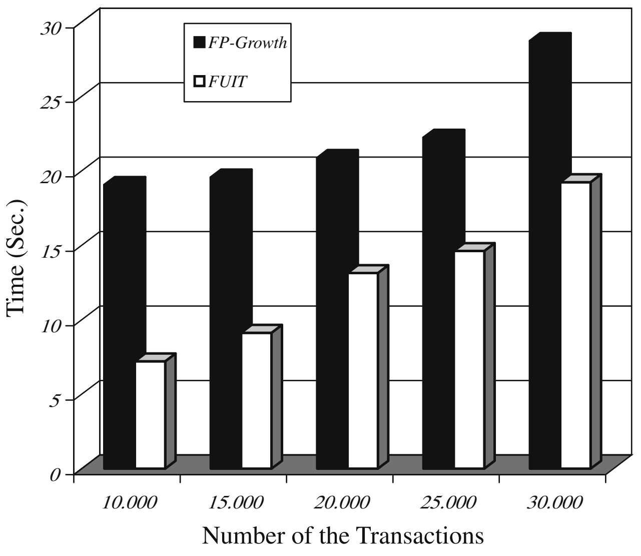


Fig. 16. Runtime of minimum support threshold 0.10%.

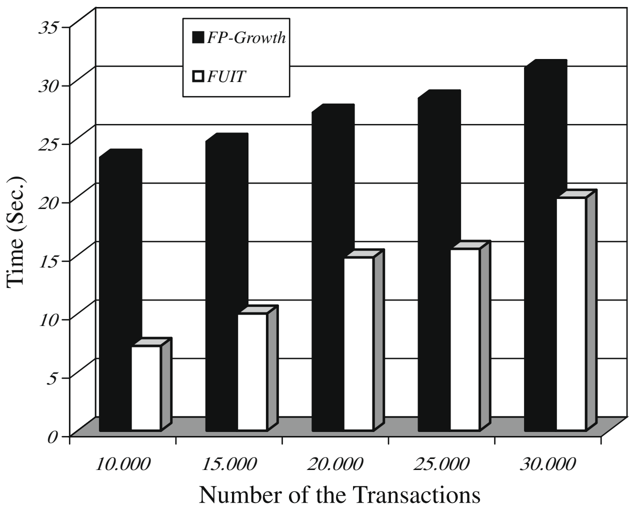


Fig. 17. Runtime of minimum support threshold 0.08%.

In the first type, various minimum support thresholds of 0.14%, 0.12%, 0.10%, and 0.08% were set, respectively, to evaluate the performances of FIUT and FP-growth. The results are shown in Figs. 10–13. As seen, both the execution time of the FIUT and the FP-growth increased when the value of minimum support decreased, however, the FIUT used less time.

In the second type, based on the same minimum support threshold, the numbers of transactions in five data sets were 10,000, 15,000, 20,000, 25,000, and 30,000, respectively, and all were scanned. The maximal length of transaction was 29, total number of items was 1983, and the maximal length of frequent itemset was 10. As shown in Figs. 14–17, both the execution time of the FIUT and the FP-growth increased when the number of transactions increased. The proposed method used less time in the second experiment than the FP-growth.

6. Conclusions

This study proposes an efficient FIUT method, based on our FIU-tree structure for mining complete frequent itemsets, which outperforms the FP-growth, a well-known and widely used algorithm. The FIUT has the following characteristics. First, the FIU-tree is an improved method to partition a database by clustering the transactions and significantly reducing the search space. Only frequent items in each transaction are inserted as a node into its FIU-tree for storing compressed databases. Then, a novel method to mine all frequent itemsets is used to check the leaves of each FIU-tree, without traversing the tree recursively, which significantly reduces the computing time. Contrary to the FP-growth method, this proposed method solves the main drawbacks of the FP-growth method, namely, searching and traversing recursively in conditional FP-trees. The experimental results showed that the proposed FIUT method is highly efficient, and can outperform the FP-growth method. Our future study will implement a parallel version of the proposed algorithm. By using the FIUT algorithm to build *k*-FIU-trees in advance, each CPU can be utilized to process a *k*-FIU-tree; hence, frequent itemsets can be immediately mined even when the database is very large.

Acknowledgements

The authors gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the quality of the paper. This research is supported by the National Science Council, Taiwan (NSC 97-2410-H-020-006-MY2).

References

[1] R. Agrawal, T. Imielinski, A. Swami, Data mining: a performance perspective, IEEE Transactions on Knowledge and Data Engineering 5 (1993) 914–925.
[2] R. Agrawal, R. Srikant, Fast algorithm for mining association rules in large databases, in: Proceedings of the 1994 International Conference VLDB, Santiago, Chile, 1994, pp. 487–499.
[3] R. Agarwal, C. Aggarwal, V.V.V. Prasad, A tree projection algorithm for generation of frequent itemsets, Journal of Parallel and Distributed Computing 61 (2001) 350–361.
[4] F. Berzal, J.C. Cubero, N. Marin, J.M. Serrano, TBAR: An efficient method for association rule mining in relational databases, Data and Knowledge, Engineering 37 (2001) 47–64.
[5] S. Brin, R. Motwani, J.D. Ullman, S. Tsur, Dynamic itemset counting and implication rules for market basket data, in: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tuscon, Arizona, 1997, pp. 255–264.
[6] D. Cheung, J. Han, V. Ng, A. Fu, Y. Fu, A fast distributed algorithm for mining association rules, in: Proceedings of 1996 International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, 1996, pp. 31–44.
[7] J. Dong, M. Han, BitTableFI: an efficient mining frequent itemsets algorithm, Knowledge-Based Systems 20 (2007) 329–335.
[8] M. EI-Hajj, O.R. Zaiane, Inverted matrix: efficient discovery of frequent items in large datasets in the context of interactive mining, in: The ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 109–118.
[9] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the 19th ACM SIGMOD International Conference on Management of Data, 2000, pp. 1–12.
[10] J. Han, M. Kamber, Data Mining Concepts and Techniques, Morgan Kaufmann, San Francisco, 2000.
[11] J.D. Holt, S.M. Chung, Mining association rules using inverted hashing and pruning, Information Processing Letter 83 (2002) 211–220.
[12] T. Hu, S.Y. Sung, H. Xiong, Q. Fu, Discovery of maximum length frequent itemsets, Information Sciences 178 (2008) 68–87.
[13] M. Kantardzic, DATA MINING: Concepts, Models, Methods, and Algorithms, Wiley Inter-science, NJ, 2003.
[14] G. Liu, H. Lu, J.X. Yu, CFP-tree: a compact disk-based structure for storing and querying frequent itemsets, Information Sciences 32 (2007) 295–319.
[15] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, D. Yang, H-mine: hyper-structure mining of frequent patterns in large database, in: Proceedings of the 2001 IEEE International Conference on Data Mining, San Jose, CA, 2001, pp. 441–448.
[16] M. Parnas, D. Roh, Testing metric properties, Information and Computation 187 (2003) 155–195.
[17] J.S. Park, M.S. Chen, P.S. Yu, An effective hash-based algorithm for mining association rules, in: Proceedings of 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, 1995, pp. 175–186.
[18] A. Savasere, E. Omiecinski, S. Navathe, An efficient algorithm for mining association rules in large databases, in: Proceedings of the 21th VLDB Conference, Zurich, Switzerland, 1995, pp. 432–444.
[19] Y.J. Tsay, Y.W. Chang-Chien, An efficient cluster and decomposition algorithm for mining association rules, Information Sciences 160 (22) (2004) 161–171.
[20] Y.J. Tsay, J.Y. Chiang, CBAR: an efficient method for mining association rules, Knowledge-Based Systems 18 (2005) 99–105.
[21] S. Zhang, J. Zhang, C. Zhang, EDUA: an efficient algorithm for dynamic database mining, Information Sciences 177 (2007) 2756–2767.