Expert Systems with Applications 38 (2011) 5154–5161

Contents lists available at ScienceDirect

Expert Systems with Applications

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

An improved frequent pattern growth method for mining association rules

Ke-Chung Lin, I-En Liao*, Zhi-Sheng Chen

Department of Computer Science and Engineering, National Chung-Hsing University, Taichung, Taiwan

ARTICLE INFO

Keywords:
FP-tree
Frequent itemset mining
Association rules

ABSTRACT

Many algorithms have been proposed to efficiently mine association rules. One of the most important approaches is FP-growth. Without candidate generation, FP-growth proposes an algorithm to compress information needed for mining frequent itemsets in FP-tree and recursively constructs FP-trees to find all frequent itemsets. Performance results have demonstrated that the FP-growth method performs extremely well. In this paper, we propose the IFP-growth (improved FP-growth) algorithm to improve the performance of FP-growth. There are three major features of IFP-growth. First, it employs an address-table structure to lower the complexity of forming the entire FP-tree. Second, it uses a new structure called FP-tree$^+$ to reduce the need for building conditional FP-trees recursively. Third, by using address-table and FP-tree$^+$ the proposed algorithm has less memory requirement and better performance in comparison with FP-tree based algorithms. The experimental results show that the IFP-growth requires relatively little memory space during the mining process. Even when the minimum support is low, the space needed by IFP-growth is about one half of that of FP-growth and about one fourth of that of nonordfp algorithm. As to the execution time, our method outperforms FP-growth by one to 300 times under different minimum supports. The proposed algorithm also outperforms nonordfp algorithm in most cases. As a result, IFP-growth is very suitable for high performance applications.

1. Introduction

Mining association rules is a very important problem in the data mining field. It consists of identifying the frequent itemsets, and then forming conditional implication rules among them. This information is useful in improving the quality of many business decision-making processes, such as customer purchasing behavior analysis, cross-marketing and catalog design.

The task of mining association rules is formally stated as follows: let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items and D be a multiset of transactions, where each transaction T contains a set of items in I. We call a subset $X \subseteq I$ an itemset and call X a k-itemset if X contains k items. The support of itemset X, denoted as $sup(X)$, is the number of transactions in D that contain all items in X. If $sup(X)$ is not less than the *min_sup* (user-specified minimum support), we call X a frequent itemset. An association rule is a conditional implication among itemsets, $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$, $X \neq \phi$, $Y \neq \phi$, and $X \cap Y = \phi$.

The confidence of the association rule, given as $sup(X \cup Y)/sup(X)$, is the conditional probability among X and Y, such that the appearance of X in T implies the appearance of Y in T. The problem of association rule mining is the discovery of association rules that have support and confidence greater than the *min_sup* and the user-defined minimum confidence.

Discovering frequent itemsets is the computationally intensive step in the task of mining association rules. The major challenge is that the mining often needs to generate a huge number of candidate itemsets. For example, if there are n items in the database, then in the worst case, all $2^k - 1$ candidate itemsets need to be generated and examined. The step of rule construction is straightforward and less expensive. Thus, most researchers concentrate on the first phase for finding frequent itemsets (Agrawal, Imielinski, & Swami, 1993; Agrawal & Srikant, 1994; Han, Pei, & Yin, 2000; Li & Lee, 2009; Orlando, Lucchese, Palmerini, Perego, & Silvestri, 2003; Park, Chen, & Yu, 1997; Zaki, Parthasarathy, Ogihara, & Li, 1997).

Agrawal and Srikant (1994) proposed the Apriori algorithm to solve the problem of mining frequent itemsets. Apriori uses a candidate generation method, such that the frequent k-itemset in one iteration can be used to construct candidate $(k + 1)$-itemsets for the next iteration. Apriori terminates its process when no new candidate itemsets can be generated. DHP, proposed by Park et al. (1997), improves the performance of Apriori. It uses a hash table to filter the infrequent candidate 2-itemsets and employs database trimming to lower the costs of database scanning. However, the aforementioned methods cannot avoid scanning the database many times to verify frequent itemsets.

Unlike Apriori, the FP-growth method (Han, Pei, Yin, & Mao, 2004; Han et al., 2000) uses an FP-tree to store the frequency

* Corresponding author.
E-mail addresses: phd9212@cs.nchu.edu.tw (K.-C. Lin), ieliao@nchu.edu.tw (I-En Liao), nickey@ms50.url.com.tw (Z.-S. Chen).

0957-4174/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2010.10.047

information of the transaction database. Without candidate generation, FP-growth uses a recursive divide-and-conquer method and the database projection approach to find the frequent itemsets. However, the recursive mining process may decrease the mining performance and raise the memory requirement. FPgrowth* (Grahne & Zhu, 2005) uses an FP-array technique to reduce the need to traverse FP-trees. Nevertheless, it still has to generate conditional FP-trees for recursive mining. The experimental results show that running time and memory consumption of FPgrowth* is almost equal to that of FP-growth. Nonordfp (Racz, 2004) improves FP-growth and it employs the tree structure to raise the mining performance. According to the result in Racz (2004), nonordfp outperforms FPgrowth* (Grahne & Zhu, 2005) and eclat (Zaki et al., 1997) in most cases. We will review nonordfp algorithm in Section 2.2.

In this paper, we propose the IFP-growth (Improved FP-growth) algorithm to improve the performance of FP-growth. First, the IFP-growth employs an address-table structure to lower the complexity of mapping frequent 1-itemsets in an FP-tree. Second, it uses a hybrid FP-tree mining method to reduce the need for rebuilding conditional FP-trees. Memory space can be saved and the cost of re-constructing conditional FP-trees can be reduced. We also present experimental results, and compare our methods to several existing algorithms, including FP-growth and nonordfp. Simulation results show that IFP-growth mines frequent itemsets efficiently with less memory space requirement. Under various minimum supports, IFP-growth can outperform FP-growth and nonordfp in execution time.

The remainder of this paper is organized as follows: Section 2 reviews FP-growth and related work. Section 3 and Section 4 present the IFP-growth mining algorithms and experimental results, respectively. Finally, Section 5 draws conclusions from this study.

2. Related work

FP-growth is a well-known frequent itemsets mining algorithm. It only scans database twice and finds all frequent itemsets efficiently compared to the Apriori algorithm. Section 2.1 shows the original FP-growth algorithm and Section 2.2 describes an improved FP-growth algorithm, namely nonordfp, which can efficiently derive frequent itemsets from a database.

2.1. The FP-growth method

The FP-growth method (Han, 2000; Han et al., 2004) requires two database scans when mining all frequent itemsets. The first scan of the database derives the set of frequent 1-itemsets, denoted as F_1, and then inserts F_1 into a header table in the decreasing order of items' supports, where this order list is denoted as L list. The header table not only stores items and their supports, but also contains a pointer to a list that links all corresponding nodes of frequent 1-itemset in the FP-tree. In the second scan, the items of F_1 in each transaction are inserted into an FP-tree in L order as a path. Distinct paths that have the same prefix path are merged to construct the FP-tree for reducing memory space.

Since an FP-tree stores all information needed for mining frequent itemsets, mining the database becomes mining FP-tree. By following a node-link, which starts at frequent 1-item x in the header table, all paths that contain x are visited. The paths from the root to the parent nodes of x form the conditional pattern base of x. Based on this conditional pattern base, FP-growth recursively constructs a new FP-tree (conditional FP-tree) and forms a new conditional pattern base. By concatenating x with the frequent itemsets generated from conditional FP-trees, the frequent patterns with x as a suffix are found. By performing such mining for each frequent 1-item, all of the frequent itemsets are discovered within the FP-tree.

There are three advantages to FP-growth. First, FP-growth compresses the entire database into a smaller data structure (FP-tree), resulting in the need to only scan the database twice. Second, it develops a frequent pattern growth method to avoid the generation of massive numbers of candidate itemsets. Third, it generates the conditional pattern tree to mine frequent itemsets, and therefore reduces the search space. According to the experimental results, FP-growth is faster than the Apriori algorithm and several methods of mining frequent itemsets.

2.2. The nonordfp method

The FP-growth method is efficient, but it has a potential drawback. The major work of mining frequent itemsets in FP-growth is recursively constructing new conditional FP-trees and traversing paths on these trees. When the FP-tree becomes huge, the recursive mining on the tree may decrease the mining performance drastically. Thus, Park et al. (1997) proposed an implementation, called nonordfp, to improve the FP-growth algorithm.

Nonordfp is based on a more compact structure called tree for constructing FP-tree. Each node in the tree contains a counter and a pointer to the parent. The nodes are stored in an array. Thus, the task of traversing paths becomes reading the array sequentially. Another major feature of nonordfp is that the recursive mining processes can be replaced by building new tree structures, which avoids rebuilding each conditional pattern base. The experimental results demonstrate that nonordfp is more efficient than the FP-growth method. Nevertheless, nonordfp incurs a space problem, as it requires continuous memory space for the array structure.

In this paper, we propose an efficient approach, IFP-growth (Improved FP-growth Algorithm), to improve the performance of FP-growth. Our approach employs an address-table structure to lower the complexity to form the entire FP-tree. Besides, it adopts a hybrid FP-tree mining method to reduce the need for rebuilding conditional FP-trees. We describe IFP-growth in the next section.

3. The improved FP-growth (IFP-growth) algorithm

The proposed algorithm utilizes the address-table structure to speed up tree construction and a hybrid FP-tree mining method for frequent itemset generation. We introduce the address-table and the hybrid FP-tree mining method in Sections 3.1 and 3.2. Then, the entire IFP-growth algorithm is presented in Section 3.3.

3.1. Address-table

The task of constructing the FP-tree and the conditional FP-trees affects the mining performance of FP-growth. Suppose that we want to add a transaction T with n frequent items $\{f_1, f_2, \ldots, f_n\}$ into an FP-tree, and the number of items in the L list is m. Without loss of generality, we let the support descending order be $f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_n$ and the depth of the root node in the FP-tree be 0. Starting at depth 0 in the FP-tree, FP-growth must check whether f_1 has existed at depth 1 m times for the worst case. Similarly, it must verify the existences of f_2 and f_3 in depths 2 and 3, respectively, $(m-1)$ and $(m-2)$ times for the worst case. Therefore, in the worst case scenario, the complexity of constructing a new path is $(m + (m-1) + \cdots + (m-(n-1)))$.

In this paper, an effective data structure called address-table is built to reduce the complexity of tree construction. An address-table contains a set of items and pointers. The pointer of an item points to the corresponding node of that item at the next level of

```
Procedure: Generate-Path([p|P], R, H)
Input: p and P are the first element and the remaining list of a sorted frequent item
       list respectively; initially, R is the root node in FP-tree; H_1 is the set of items in
       the header table.
Output: a path in FP-tree
/* ATable is an address-table*/
Method:
1.  if R does not have an address-table
2.      create an ATable and let R→ATable= ATable;
3.      for each item h in H_1
4.          ATable [i].name=h;
5.          ATable [i].pointer=null;
6.      Assume ATable[k].name=p;
7.      if ATable [k].pointer ≠ null
8.          (ATable[k].pointer)→count=(ATable
        [k].pointer)→count+1;
9.      else
10.         create a new node Node;
11.         Node.name=p;Node.count= 1; Node.pointer=null;
12.         Node is linked to the nodes with item p in the
        header table via the node-link structure;
13.         ATable [k].pointer=Node;
14.     if P ≠ φ
15.         Generate-Path (P, Node, H_1-p)
```

Fig. 1. The procedure of generate-path.

Table 1
The transaction database.

Transaction ID	Items bought	Ordered frequent items
100	*e, c, d, b, h, a, z*	*c, d, b*
200	*x, d, c, b, k*	*c, d, b, k*
300	*c, d, j, k*	*c, d, k*
400	*s, t, u, c*	*c*
500	*g, b, a, y, z*	*b*
600	*h, d, k*	*d, k*

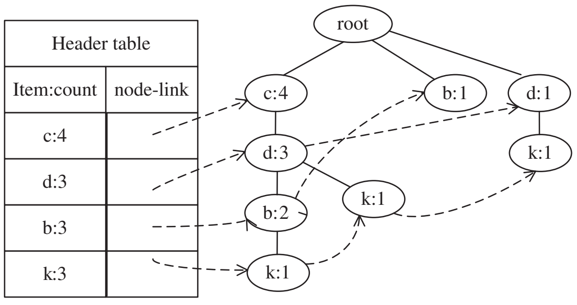

Fig. 2. An FP-tree for Table 1.

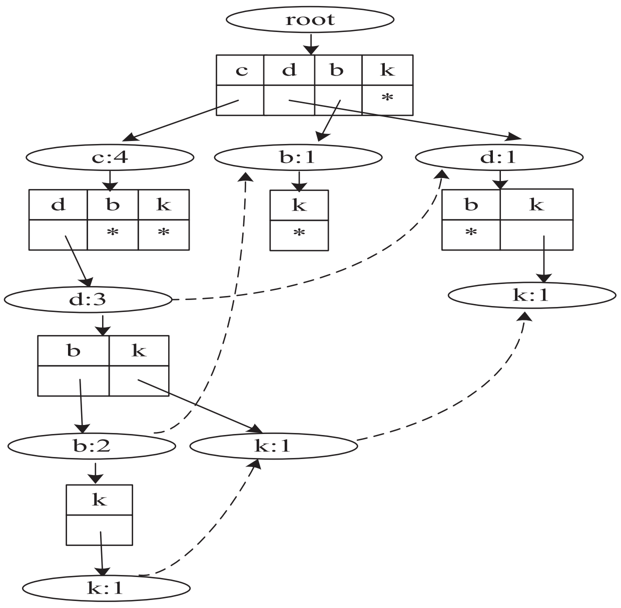

Fig. 3. FP-tree with address tables.

FP-tree. Thus, the task of finding the child node in the FP-tree becomes easy. Each node checks its address-table to confirm whether its child exists in the tree. Fig. 1 shows how our approach employs the address-table to insert a path into the FP-tree.

Table 1 shows an example database and sets the *min_sup* as 3. Fig. 2 illustrates the original FP-tree for Table 1 and Fig. 3 is a new FP-tree constructed using the address-table. Each node of the new FP-tree contains a Node structure and an address table. The Node structure has item name, count, node-link, and a pointer to the address table. The address table contains a set of items and pointers. The pointer of an item points to the corresponding node of that item at the next level of FP-tree. A "*" for a pointer in an address-table denotes null pointer.

3.2. The hybrid FP-tree mining method

In this section, we introduce an FP-tree$^+$ technique. The FP-tree$^+$ is used for reducing the needs for rebuilding FP-trees in each conditional pattern mining step. By combining the FP-tree$^+$ technique and the conditional FP-tree technique, we propose a hybrid FP-tree mining method to efficiently discover frequent itemsets.

The mining procedure of FP-tree$^+$ is similar to that of the conditional FP-trees, but the direction of mining an FP-tree$^+$ is opposite to that of mining a conditional FP-tree. It discovers frequent itemsets by traversing the items from the top to the bottom of a header table. According to the feature, each FP-tree$^+$ can be built on the original FP-tree and the memory requirements can be reduced.

The construction of an FP-tree$^+$ and the mining procedure in the FP-tree$^+$ are presented as follows: Each node of an FP-tree$^+$ contains item_name, count, node-link and pointer to an address-table. Initially, when we want to discover frequent itemsets including item x in an FP-tree, all paths, including x, are found first. Let the support of x in a path p_i be $p_i(x)$. In a path including x, each ancestor's support value is set to be the same with $p_i(x)$. If two paths, p_i and p_j, share a common prefix, the shared paths are merged and each node's support in the prefix is accumulated by $p_i(x)$ and $p_i(y)$. The support of each item in these paths is accumulated to the corresponding item in the header table. The chain of node-links to point to these item's occurrences in the FP-tree is also modified. Thus, the x's FP-tree$^+$ is formed. In particular, the infrequent nodes in x's FP-tree$^+$ do not be eliminated. Based on the accumulated support information in the header table, the mining process then constructs a frequent item y's FP-tree$^+$ on x's FP-tree$^+$. By performing the mining process recursively, all frequent itemsets associated with x are derived.

Fig. 4 shows the procedure of constructing an FP-tree$^+$ from the original frequent pattern tree generated from the original database. The support of each parent is accumulated from the children nodes (lines 12 and 16). The node-link of each node joins nodes with the same item via the node-link structure (lines 13 and 14).

For example, we use FP-tree$^+$ to mine frequent itemsets in the FP-tree in Fig. 2. The frequent itemsets associated with c, d, b, and k are mined consecutively. We show the mining process of item k's FP-tree$^+$ in Fig. 5. For node k, it derives a frequent item "k" and derives three paths: $\{\langle c{:}1, d{:}1, b{:}1, k{:}1\rangle, \langle c{:}1, d{:}1, k{:}1\rangle, \langle d{:}1, k{:}1\rangle\}$. The chain of node-links in k's FP-tree$^+$ is updated. The supports of node c (i.e., 2) and node d (i.e., 2) in the tree are accumulated from path $\langle c, d, b, k\rangle$ and $\langle c, d, k\rangle$. In k's FP-tree$^+$, since d is frequent in header table, a frequent itemset $\{kd\}$ is output and kd's FP-tree$^+$ is built for mining recursively. In the example, no frequent itemsets are derived from kd's FP-tree$^+$. Thus, the frequent itemsets associated with k are $\{k, kd\}$.

```
Procedure: FP-tree+ Construction
Input: Tree: a frequent pattern tree;
Output: item k's FP-tree+
Parameter: k: a target item; F1: the set of frequent 1-items; HTable: a header-table;
Method:
1.     H1 : = {c | c ∈ F1, sup(c) ≥ sup(k), c ≠ k };
2.     for (i=0;i<=|H1|;i++) do begin
3.         HTable[i].count = 0;
4.         HTable[i].node_link = NULL;
5.     end
6.     Assume HTable[n].name=k;
7.     CNode= HTable[n].node_link;
8.     while (CNode != NULL) do begin
9.           PNode= CNode.parent;
10.    while (PNode !=Root) do begin
11.            Assume HTable[x].name=PNode.name;
12.            HTable[x].count + = CNode.count;
13.            if (PNode has only one child node)
14.                  PNode.count = CNode.count;
15.                  PNode.node_link=HTable[x].node_link;
16.                  HTable[x].node_link= PNode;
17.            else
18.                  PNode.count += CNode.count;
19.            PNode =PNode.parent;
20.     end
21.     CNode=CNode.node_link;
22. End
```

Fig. 4. The procedure of FP-tree$^+$ construction.

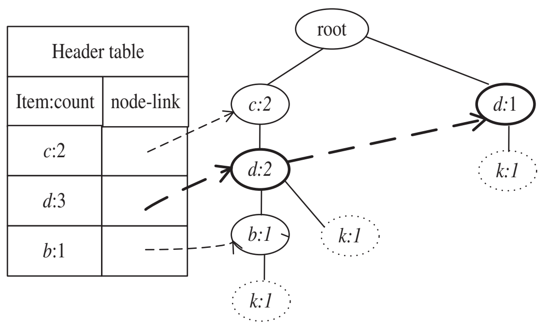

Fig. 5. The FP-tree$^+$ of item k.

The analysis of the FP-tree$^+$ is described as follows: First, without rebuilding conditional FP-trees as FP-growth, memory usage is saved. Second, building the FP-tree$^+$ is fast because it does not require extra costs to eliminate infrequent items. However, this feature of containing infrequent items may cause potential performance problems in traversing FP-tree$^+$.

To effectively raise mining performance, we propose a new method which combines the conditional FP-tree technique and the FP-tree$^+$ technique. The important characteristic of the method is that it not only reduces the times of tree rebuilding but saves the memory requirement. This method uses a tree-level value to divide the items of the header table into two parts. Then, it employs the conditional FP-tree technique and the FP-tree$^+$ technique to mine these items' conditional bases. We call the set of items under the tree-level value the "top-half," and the other is the "bottom-half." If an item x belongs to the top-half, an FP-tree$^+$ is constructed to discover its original conditional base. Otherwise, a conditional FP-tree is used. We present the entire mining process in the next section.

3.3. Algorithm IFP-growth

Fig. 6 gives the algorithm of IFP-growth, which is separated into two parts. Part 1 receives frequent 1 items and builds a frequent pattern tree *FP-tree* by scanning the database *DB* twice. Part 2 generates all frequent itemsets from the *FP-tree* by performing the *IFP-Mining* procedure.

IFP-growth employs a hybrid method to mine frequent itemsets. Its main program is shown in Fig. 7. The set S is output whenever new frequent itemsets are found (lines 3 and 7). From line 1 to line 3, the frequent itemsets in a single path is first discovered. In line 8, IFP-Mining determines the type of conditional pattern tree being used. If the target item belongs to the bottom-half, a conditional FP-tree is constructed to recursively mine (lines 9–11). Otherwise, an FP-tree$^+$ is formed for further mining (lines 13–15).

4. Experimental results

To access the performance of IFP-growth, we used three algorithms, IFP-growth, FP-growth (Han, 2000; Han et al., 2004) and nonordfp Racz (2004) to mine frequent itemsets from various databases. The experiments were performed on an Intel Core2 Duo processor 1.66 GHz with 512 MB memory, running the Redhat AS3.0 GUN/Linux. We used the C language to code IFP-growth. Furthermore, both FP-growth and nonordfp were downloaded from http://fimi.cs.helsinki.fi/src/ and were compared with IFP-growth to present the performance comparison.

We chose several real and synthetic data to test the performance of the algorithms. The data sets were often used in previous studies of frequent itemsets mining and were downloaded from FIMI'04 website http://fimi.cs.helsinki.fi/data/. Table 2 summarizes the characteristics of these data sets with the number of items, average transaction length and the number of transactions in each database.

We conducted experiments to observe the influence of various tree-level values in IFP-growth. Fig. 8 shows the runtime of dense data set pumsb, with respect to the various minimum supports. We observe that the value of tree-level has great effects on the performance of IFP-growth, and that the best tree-level value may be affected by the minimum support. In this paper, we set the default value of tree-level to 20 in our IFP-growth algorithm. In most our test data sets, this setting leads the IFP-growth algorithm has a better performance than others two approaches.

Figs. 9 and 10 show the runtime and highest memory requirement of the three algorithms on sparse data set T10I4D100K, with respect to various minimum supports. As indicated in the

```
Procedure: IFP-growth
Input : DB: a transaction database; min_sup: user-specified minimum support;
Output : The set of frequent itemsets;
Parameter : l: a tree-level value at which different mining strategies are applied;
Method:
/* Part 1 */
1.  Scan DB once and find all frequent 1-items $F_1$. Sort $F_1$ in descending order as L;
2.  Create a root R of an FP-tree FP-Tree and label it as "null";
3.  for each transaction T in DB
4.      Generate-Path(T, R, L);
/* Part 2 */
5.  IFP-Mining(FP-Tree, null);
```

Fig. 6. Main procedure of IFP-growth.

```
Procedure: IFP-Mining
Input: Tree: a frequent pattern tree; $\alpha$ : a frequent itemset; /* the value of $\alpha$ is set
as "null" initially*/

Output: The set of frequent itemsets;

Parameter: l: the value of tree-level

Method:
1.  if Tree contains a single path B
2.       for each subset S of the nodes in B do
3.     output $S \cup \alpha$ with support = smallest support of nodes in B;
4.  else
5.       for each item i in the header-table do
6.            if header-table[i].count>=min_sup
7.                 output $S = i \cup \alpha$ with support equal to that of i;
8.                 if $i \in$ bottom-half do
9.                      construct S's conditional FP-tree $Tree_s$;
10.                if $Tree_s \neq \phi$ then
11.                     IFP-Mining($Tree_s$, S);
12.           else
13.                construct S's FP-tree$^+$ $Tree_s^+$;
14.                if $Tree_s^+ \neq \phi$ then
15.                     IFP-Mining($Tree_s^+$, S);
```

Fig. 7. The procedure of IFP-growth.

Table 2
Database characteristic.

Database	#Transactions	#Items	Size (kB)
Accidents	340,183	468	34,678
Pumsb	49,046	2113	16,299
Connect	67,557	129	9039
T10I4D100K	100,000	1000	3928

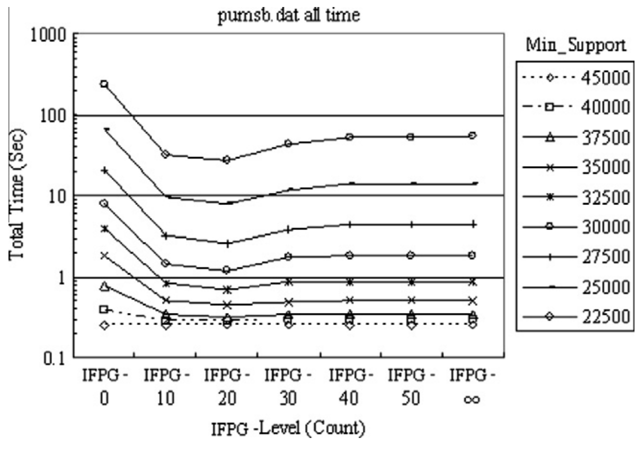

Fig. 8. Runtime of IFP-growth with respect to various tree-level values.

experimental results, IFP-growth is always faster than the FP-growth method and nonordfp, and nonordfp always requires higher memory usage than IFP-growth and FP-growth. In most cases, IFP-growth and FP-growth consume almost the same amount of memory. However, when the minimum support is low, the number of frequent itemsets increases and FP-growth

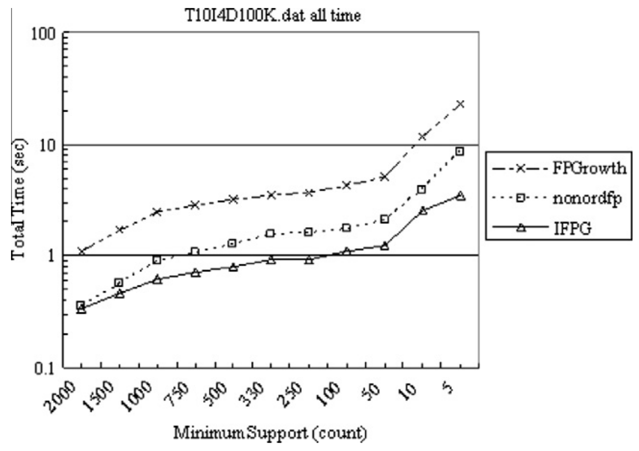

Fig. 9. Runtime of mining frequent itemsets on T10I4D100K.

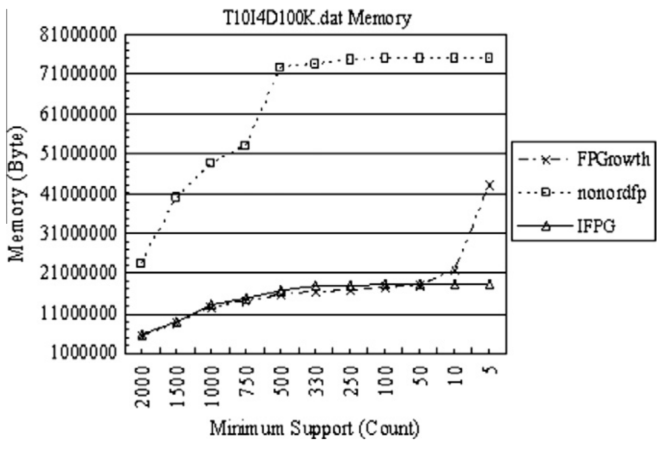

Fig. 10. Memory usage of mining frequent itemsets on T10I4D100K.

builds more conditional FP-trees for frequent itemset generation. This causes FP-growth to raise its memory usage dramatically.

We conducted experiments to verify the influence of various tree-level values in IFP-growth. Figs. 11 and 12 depict the runtime results and memory amount results on accidental data sets. The value of *i* in the parameter "IFP-growth-*i*" means the value of tree-level of IFP-growth. IFP-growth-0 employs conditional FP-trees only and IFP-growth-∞ employs the FP-tree⁺. We observe that IFP-growth-∞ has the best performance in this data set because the appropriate setting of a tree-level is dependant on the distribution of database. Due to the address-table, the performance of IFP-growth-0 is better than that of FP-growth. When the minimum support is low, the performance of IFP-growth-20 and IFP-growth-0 is worse than that of nonordfp because they need to take extra cost to rebuild conditional FP-trees.

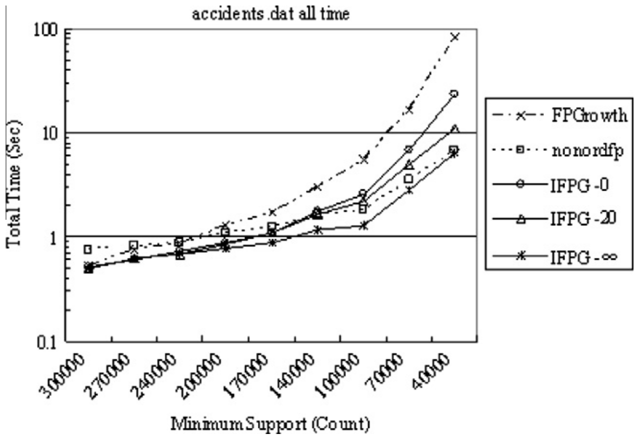

Fig. 11. Runtime of mining frequent itemsets on accidents data set.

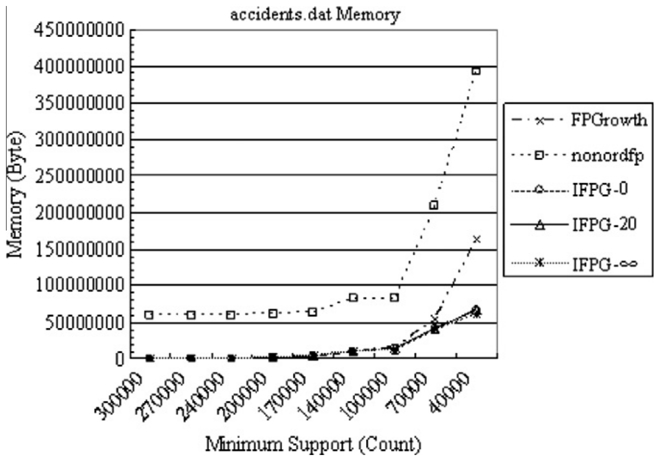

Fig. 12. Memory usage of mining frequent itemsets on accidents data set.

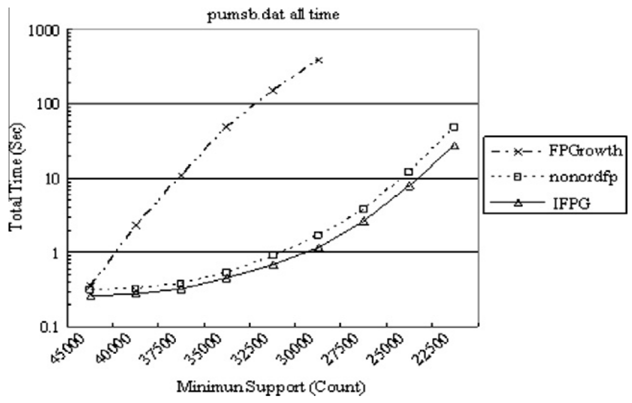

Fig. 13. Runtime of mining frequent itemsets on pumsb data set.

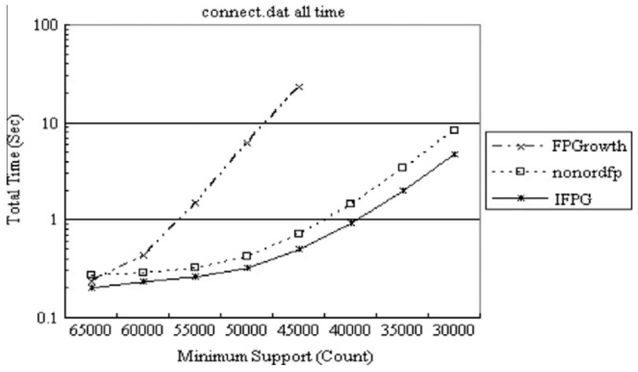

Fig. 14. Runtime of mining frequent itemsets on connect data set.

Figs. 13 and 14 show the performances of all algorithms in the pumsb data set and the connect data set. IFP-growth outperforms all other methods in the two cases. In particular, IFP-growth outperforms FP-growth in execution time. In this data set, the number of frequent itemsets is greater than five billion and FP-growth stores 161,583 nodes in the FP-tree. In such huge FP-tree, recursively constructing conditional FP-trees makes the mining performance of FP-growth become worse.

In conclusion, IFP-growth is a very efficient algorithm for mining all frequent itemsets, based on the experimental results of runtime and memory consumption. When the data set is dense, IFP-growth has a better speed performance than FP-growth and nonordfp, and its memory requirement is lower. Even if the minimum support becomes low, IFP-growth remains efficient. Thus, IFP-growth is very suitable for high performance applications.

5. Conclusion

By incorporating the FP-tree⁺ mining technique and the address-table into FP-growth, we propose the IFP-growth algorithm for frequent itemsets generation. The major advantages of FP-tree⁺ and the address-table are that they reduce the need to rebuild conditional trees and facilitate the task of tree construction. The memory requirement of IFP-growth is also lower than that of FP-growth and nonordfp. Experimental results showed that our algorithm is more than an order of magnitude faster than the FP-growth algorithm.

However, IFP-growth incurs a potential problem in the hybrid FP-tree mining method. An appropriate tree-level value in this method is important, because it affects the mining performance of our algorithm. As a result, our future work will involve finding an appropriate tree-level value between the conditional FP-tree technique and the FP-tree⁺ technique in the hybrid FP-tree mining method.

References

Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD conference on management of data* (pp. 207–216).

Agrawal, R., & Srikant, R. (1994). Fast algorithm for mining association rules in large databases. In *Proceedings of 20th VLDB conference* (pp. 487–499).

Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using FP-trees. *IEEE Transactions on Knowledge and Data Engineering, 17*(10), 1347–1362.

Han, J., Pei, J. & Yin, Y. (2000). Mining frequent patterns without candidate generation. In *Proceedings of the ACM-SIGMOD conference management of data* (pp. 1–12).

Han, J., Pei, J., Yin, Y., & Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery, 8*(1), 53–87.

Li, H. F., & Lee, S. Y. (2009). Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Systems with Applications*, 1466–1477.

Orlando, S., Lucchese, C., Palmerini, P., Perego, R., & Silvestri, F. (2003). kDCI: A multi-strategy algorithm for mining frequent sets. In *Proceedings of IEEE ICDM workshop on frequent itemset mining implementations*.

Park, J. S., Chen, M. S., & Yu, P. S. (1997). Using a hash-based method with transaction trimming for mining association rules. *IEEE Transactions on Knowledge and Data Engineering, 9*(5), 813–825.

Racz, B. (2004). Nonordfp: An FP-growth variation without rebuilding the FP-tree. In *Proceedings of IEEE ICDM workshop on frequent itemset mining implementations.*

Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W. (1997). New algorithms for fast discovery of association rules. In *Proceedings of 3rd knowledge discovery and data mining conference* (pp. 283–286).