TF^{2}P-growth: An Efficient Algorithm for Mining Frequent Patterns without any Thresholds

Yu HIRATE, Eigo IWAHASHI, and Hayato YAMANA
Graduate School of Science and Engineering, Waseda University
{hirate, eigo, yamana}@yama.info.waseda.ac.jp

Abstract

Conventional frequent pattern mining algorithms require some user-specified minimum support, and then mine frequent patterns with support values that are higher than the minimum support. As it is difficult to predict how many frequent patterns will be mined with a specified minimum support, the Top-k mining concept has been proposed. The Top-k Mining concept is based on an algorithm for mining frequent patterns without a minimum support, but with the number of most k frequent patterns ordered according to their support values. However, the Top-k mining concept still requires a threshold k. Therefore, users must decide the value of k before initiating mining. In this paper, we propose a new mining algorithm, called "TF^{2}P-growth," which does not require any thresholds. This algorithm mines patterns with the descending order of their support values without any thresholds and returns frequent patterns to users sequentially with short response time.

1. Introduction

Due to recent developments in network infrastructure and both price reduction and increases in capacity of storage devices, it has become commonplace to archive large amounts of data. It is important to analyze such large data sets because they may contain new knowledge. The discovery of such knowledge requires data mining technology.

Both the long response time when mining large amounts of data and usability with regard to specification of some threshold values are fundamental problems in data mining, especially in frequent pattern mining. Many algorithms have been proposed to resolve these problems. Conventional frequent pattern mining algorithms are classified into two categories: the candidate-generation-and-test approach and the pattern-growth approach.

Candidate-generation-and-test approach algorithms, such as Apriori[1], suffer from both the generation of huge numbers of candidates and scanning of the dataset many times to count the frequency of generated patterns, resulting in long response times.

Although most frequent pattern mining algorithms proposed prior to 2000 were based on the candidate-generation-and-test approach, another approach called pattern-growth has also been proposed. Pattern-growth approach algorithms, such as FP-growth[2], scan the dataset only a few times. Moreover, pattern-growth approach algorithms mine frequent patterns without generating any candidates. Thus, in most cases, algorithms based on the pattern-growth approach mine frequent patterns faster than those based on the candidate-generation-and-test approach. However, the user must still wait for a long time to mine large numbers of frequent patterns if users fail to specify a minimum support.

On the other hand, algorithms based on another concept have also been proposed. This is called "concept mining," examples of which include maximal pattern mining[3][4], closed pattern mining[5][6], and Top-*k* mining[7][8].

Using maximal pattern mining or closed pattern mining concept algorithms, users can mine frequent patterns containing only their superset patterns excluding any subset patterns. Using Top-*k* mining concept algorithms, users can mine the most k-frequent patterns in descending order of support without specifying a minimum support.

With regard to usability, the Top-*k* mining concept is important because the user does not have to specify a minimum support, which is usually difficult to choose when mining a moderate number of frequent patterns. The Top-*k* mining concept algorithms Itemset-Loop/Itemset-iLoop and TFP-Mining were proposed by Fu *et al.* [7] and Han *et al.* [8] in 2000 and 2002, respectively.

Itemset-Loop/Itemset-iLoop mines the most *k*-frequent patterns with lengths shorter than the user-specified value of m. In contrast, TFP-Mining mines the most *k*-closed frequent patterns with lengths longer than the user-specified value of m. However, such Top-*k* mining concept algorithms still require a threshold k before the initiation of mining. Therefore, users must decide the value of k. When the value of k is too large, mining takes a long time. In contrast, using a value of k that is

too small usually results in mining only useless patterns even though the mining time is short. Therefore, there are still some difficulties in specifying the value of k.

In this paper, we propose a new mining algorithm, called "TF^{2}P-growth," which does not require any threshold values. TF^{2}P-growth, based on FP-growth, mines patterns with descending order of support values. Then, it returns frequent patterns to users sequentially with a short response time.

The remainder of this paper is organized as follows. The terms used are explained in section 2. Related works are described in section 3. Then, we introduce our proposed method in section 4. Section 5 reports the performance evaluation of our proposed method. In section 6, we summarize our work and discuss some future research directions.

2.Terms definition

Let $I = \{i_1, i_2, \cdots, i_n\}$ be a set of items. An itemset X is a non-empty subset of I. An itemset with m items is called an m-itemset. Duple $< tid, X >$ is called a transaction where tid is a transaction identifier and X is an itemset. A transaction database TDB is a set of transactions.

Given a transaction database TDB, the support of an itemset X, denoted as $\sup(X)$, is the number of transactions including the itemset X. A frequent pattern is defined as the itemset whose support is higher than the minimum support *min_sup*. When itemsets are aligned by their support in descending order, the support of the k-th itemset is denoted as α. Then, the Top-k frequent patterns are defined as the itemsets whose support values are higher than α.

3. Related Works

In this section, researches on both basic frequent pattern mining and concept mining are described.

3.1. Basic frequent pattern mining algorithms

3.1.1. Apriori[1]. Apriori is a basic breadth first algorithm. The theory of Apriori is based on the fact that the itemset X' containing itemset X is never frequent if itemset X is infrequent. Based on the theory, Apriori iteratively generates a set of candidate frequent patterns whose lengths are $(k+1)$ from the k-itemsets (for $k \geq 1$). Then, their corresponding supports are checked.

There are many variants that have improved on Apriori by further reducing the number of candidates generated[9], or by reducing the number of *TDB* scans[10].

3.1.2. FP-growth[2]. In 2000, Han *et al.* proposed the FP-growth algorithm—the first pattern-growth concept algorithm. FP-growth constructs an FP-tree structure and mines frequent patterns by traversing the constructed FP-tree. The FP-tree structure is an extended prefix-tree structure involving crucial condensed information of frequent patterns.

a) FP-tree structure

The FP-tree structure has sufficient information to mine complete frequent patterns. It consists of a prefix-tree of frequent 1-itemset and a frequent-item header table.

Each node in the prefix-tree has three fields: *item-name*, *count*, and *node-link*.

- *item-name* is the name of the item.
- *count* is the number of transactions that consist of the frequent 1-items on the path from root to this node.
- *node-link* is the link to the next same item-name node in the FP-tree.

Each entry in the frequent-item header table has two fields: *item-name* and *head of node-link*.

- *item-name* is the name of the item.
- *head of node-link* is the link to the first same item-name node in the prefix-tree.

b) Construction of FP-tree

FP-growth has to scan the *TDB* twice to construct an FP-tree. The first scan of *TDB* retrieves a set of frequent items from the *TDB*. Then, the retrieved frequent items are ordered by descending order of their supports. The ordered list is called an F-list. In the second scan, a tree T whose root node R labeled with "null" is created. Then, the following steps are applied to every transaction in the *TDB*. Here, let a transaction represent $[p \mid P]$ where p is the first item of the transaction and P is the remaining items.

- In each transaction, infrequent items are discarded.
- Then, only the frequent items are sorted by the same order of F-list.
- Call insert_tree $(p \mid P, R)$ to construct an FP-tree.

The function insert_tree $(p \mid P, R)$ appends a transaction $[p \mid P]$ to the root node R of the tree T. Pseudo code of the function insert_tree $(p \mid P, R)$ is shown in Figure 1.

An example of an FP-tree is shown in Figure 2. This FP-tree is constructed from the *TDB* shown in Table 1 with *min_sup* = 3. In Figure 2, every node is represented by $(item-name : count)$. Links to next same item-name node are represented by dotted arrows.

Table 1. Sample *TDB*

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

```
function insert_tree (p | P, R) {
  let N be a direct child node of R, such that N 's
    item-name = p 's item-name.
   if ( R has a direct child node N ) {
     increment N 's count by 1.
   }
   else{
     create a new node M linked under the R .
     set M 's item-name equal to p .
     set M 's count equal to 1.
   }
   call insert_tree (P, N) .
}
```

Figure 1. Pseudo code of insert_tree $(p \mid P, R)$

c) FP-growth

FP-growth mines frequent patterns from an FP-tree. To generate complete frequent patterns, FP-growth traverses all the node-links from "head of node-links" in the FP-tree's header table. For any frequent item a_i, all possible frequent patterns including a_i can be mined by following a_i's node-link starting from a_i's head in the FP-tree header table.

In detail, a_i's prefix path from a_i's node to root node is extracted at first. Then, the prefix path is transformed into a_i's conditional pattern base, which is a list of items that occur before a_i with the support values of all the items along the list. Then, FP-growth constructs a_i's conditional FP-tree containing only the paths in a_i's conditional pattern base. It then mines all the frequent patterns including item a_i from a_i's conditional FP-tree.

For example, we describe how to mine all the frequent patterns including item p from the FP-tree shown in Figure 2. For node p, FP-growth mines a frequent pattern (p:3) by traversing p's node-links through node (p:2) to node (p:1). Then, it extracts p's prefix paths;

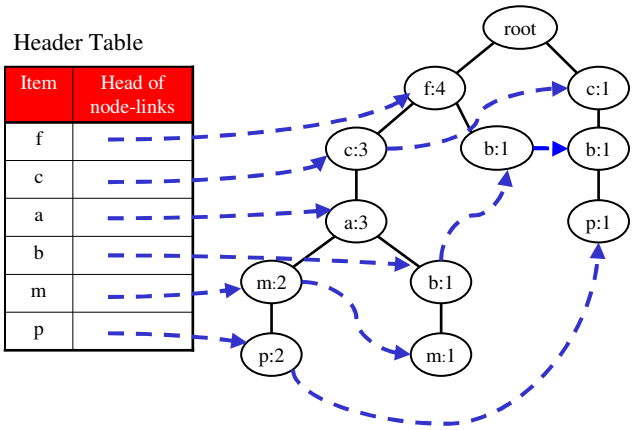

Figure 2. Example of an FP-tree

{(f:2,c:2,a:2,m:2),(c:1,b:1)}
p's conditional pattern base

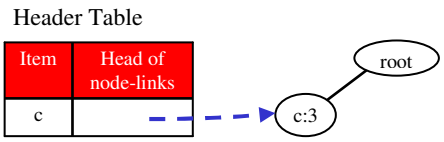

Figure 3. p 's conditional FP-tree

<f:4,c:3,a:3,m:2> and <c:1,b:1>. To study which items appear together with p, the transformed path <f:2,c:2,a:2,m:2> is extracted from <f:4,c:3,a:3,m:2> because the support value of p is 2. Similarly, we have <c:1,b:1>. The set of these paths {(f:2,c:2,a:2,m:2),(c:1,b:1)} is called p's conditional pattern base.

FP-growth then constructs p's conditional FP-tree containing only the paths in p's conditional pattern base as shown in Figure 3. As only *c* is an item occurring more than *min_sup* appearing in p's conditional pattern base, p's conditional FP-tree leads to only one branch (c:3). Hence, only one frequent pattern (cp:3) is mined. The final frequent patterns including item p are (p:3) and (cp:3).

3.2. Concept mining algorithms

3.2.1. Maximal Pattern Mining concept. Basic frequent pattern mining often mines a huge number of frequent patterns. However, it is difficult to find new knowledge from such huge numbers of frequent patterns. To resolve this problem, maximal pattern mining algorithms, such as Max-Miner[3] and FPmax[4], which mine only the maximal frequent patterns, were proposed.

Definition1 (Maximal frequent pattern) A pattern X is defined as a maximal frequent pattern iff the following two conditions are satisfied simultaneously:

(1) The support value of X is higher than *min_sup*.

(2) There exists no pattern X ' whose support value is higher than *min_sup*, where X ' is any superset of X .

3.2.2. Closed Pattern Mining concept. Similar to the maximal pattern mining concept, the closed pattern mining concept was proposed to reduce the number of patterns generated. Closed pattern mining algorithms, such as CLOSET[6] and FP-close[5], mine only closed frequent patterns.

Definition2 (Closed Frequent pattern) A pattern X is defined as a closed frequent pattern iff the following two conditions are satisfied simultaneously:

(1) The support value of X is higher than *min_sup*.

(2) There exists no pattern X ' whose support value is higher than *min_sup*, where X ' is a superset of X and is included in all the transactions that include X .

3.2.3. Top-*k* Mining concept. Generally, it is difficult to predict how many frequent patterns will be mined from a user-specified *min_sup*. If *min_sup* is low, a huge number of frequent patterns are mined. On the other hand, if the *min_sup* is large, a small number of frequent patterns are mined. Thus, it is difficult for users to decide the *min_sup* value.

To avoid such difficulties, the Top-*k* mining concept was proposed to mine the most k frequent patterns with descending order of support without specifying *min_sup*[7][8].

4. Proposed Algorithm

4.1. Problems with Conventional Algorithms

The Top-*k* mining concept is important to enhance usability for real applications for data mining. However, the Top-*k* mining concept still requires a threshold k and users must decide the value of k before initiating mining.

4.2. Overview of the proposed algorithm

Our proposed algorithm,"TF^{2}P-growth," mines patterns with descending order of support values without specifying any thresholds. Then, it returns frequent patterns to users sequentially with short response times. First, in section 4.3, we propose "Top-k FP-growth," which is a Top-*k* mining concept algorithm extended from FP-growth. Second, in section 4.4, we propose "TF^{2}P-growth" based on "Top-k FP-growth."

4.3. Top-k FP-growth

Our proposed Top-k FP-growth algorithm is the base algorithm of TF^{2}P-growth. This algorithm generates Top-*k* patterns without a threshold of *min_sup* but with a threshold k value.

4.3.1. Extension from FP-growth. To reduce additional computation, we extended FP-growth for Top-k FP-growth in three points, a) setting the internal threshold *Border_sup*, b) reducing the number of patterns generated from the FP-tree, and c) outputting frequent patterns.

a) Setting of Border_sup

Definition 3 (*Border_sup*) *Border_sup* is defined as the support value of *k*-th frequent 1-itemset. This means that there are at least k 1-itemsets with support values higher than *Border_sup*.

Top-k FP-growth constructs an FP-tree using *Border_sup* as a threshold. *Border_sup* is an internal threshold and its value is defined automatically. Thus, users do not have to be concerned with *Border_sup*. In concrete terms, frequent items, which are the primitives of FP-tree construction, are 1-itemsets whose support values are higher than *Border_sup*.

[Lemma4.1] If the support value of 1-itemset t is lower than *Border_sup*, t cannot be used to generate most k frequent patterns.

[Rationale] Let α be any 1-itemset whose support is lower than *Border_sup*. Let β be any itemset. Then, the following expression is satisfied.

$$\sup(\{\alpha, \beta\}) \leq \sup(\{\alpha\}) < Border\_sup$$

The above expression shows that the support values of any itemsets including the 1-itemset whose support value is lower than *Border_sup* are lower than *Border_sup*. In addition, it is clear that the number of itemsets whose support values are higher than *Border_sup* is more than k , based on the definition of *Border_sup*. Thus, we are able to limit the number of frequent 1-itemsets that are primitives of the FP-tree to the number of 1-itemsets whose support values are more than *Border_sup*.

For example, given the TDB shown in Table 1, with k = 6, *Border_sup* is 3 because the support value of the 6-th frequent 1-itemset is 3. Thus, the primitives of the FP-tree becomes *f*, *c*, *a*, *b, m,* and *p*.

b) Reducing the number of patterns generated from the FP-tree

Pattern generation from the constructed FP-tree by traversing all items' node-link drives more than k patterns.

To reduce both the number of patterns generated and the number of traversing node-links, a "Reduction Array" is adopted in Top-k FP-growth as shown in Figure 4. Top-k FP-growth stores both the patterns generated from the FP-tree and their support values sequentially into a "Reduction Array." The primitives of the Reduction Array are sorted by descending order of their support values after every traversal of one node-link.

Definition 4 (*Boundary_sup*) *Boundary_sup* is the support value of the stored *k*-th pattern in the Reduction Array. Initially, *Boundary_sup* is set to 0, but its value increases after the generation of k patterns from an FP-tree.

After traversing the node-link of each item α in an FP-tree, but before traversing the node-link of the next item β in the FP-tree, Top-k FP-growth compares the support value of item $\beta(=\sup(\beta))$ with *Boundary_sup*. If the expression $\sup(\beta) <$ *Boundary_sup* is satisfied, it terminates pattern generation from the FP-tree. On the other hand, if the expression $\sup(\beta) <$ *Boundary_sup* is not satisfied, it continues pattern generation from the FP-tree.

The reason why traversal of node-links is terminated if the support value of the next item is lower than *Boundary_sup* is described below.

None of the patterns generated after traversing the node-link of item β have support values higher than $\sup(\beta)$. Thus, if the expression $\sup(\beta) <$ *Boundary_sup* is satisfied, no patterns including item β have support values that are higher than *Boundary_sup*. Moreover, every item γ located under the item β in the FP-tree's header table and all of the patterns generated by traversing the node-link of item γ have support values lower than *Boundary_sup*. Thus, Top-k FP-growth can generate Top-*k* patterns in proportion even if the generation of patterns from the FP-tree terminates when the expression $\sup(\beta) <$ *Boundary_sup* is satisfied.

An example of the Reduction Array is shown in Figure 4. Figure 4 shows a Reduction Array after traversal of the node-link of an item *a*, given the TDB shown in Table 1 with *k*=6. In Figure 4, as the support value of the 6-th pattern {*c*, *a*} is 3, *Boundary_sup* is defined as 3. Before traversing the next node-link of an item {*b*}, Top-k FP-growth compares $\sup(b)$ with *Boundary_sup*. In this case, as $\sup(b)$ equals *Boundary_sup*, Top-k FP-growth continues traversing the node-link of item {*b*}.

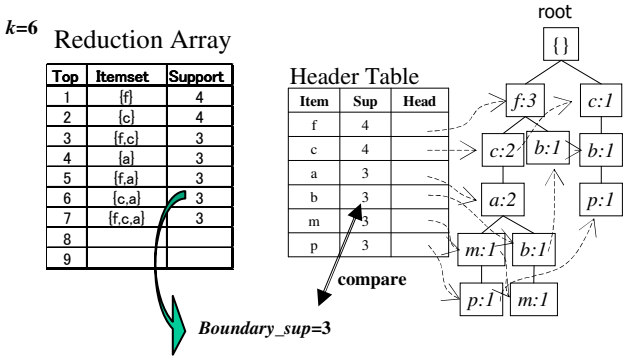

Figure 4. Example of a Reduction Array

c) Output of frequent patterns

Even if we adopt both extension a) and extension b), the number of frequent patterns generated from the FP-tree is still greater than *k*. Thus, Top-k FP-growth outputs the most *k* frequent patterns referring to the Reduction Array by descending order of their support values.

4.3.2. Top-k FP-growth Algorithm. The Top-k FP-growth algorithm is shown below.

INPUT TDB, k (number of frequent patterns)
OUTPUT most *k* frequent patterns (descending order)
METHOD

1. Scan TDB, count support of all 1-itemsets.
2. Set *Border_sup* to the support value of *k*-th 1-itemset (descending order). Then, generate an F-list.
3. Construct an FP-tree according to the F-list.
4. Generate frequent patterns from the FP-tree. During generation, at every traversal of a node-link, re-calculate *Boundary_sup* for reduction of the number of patterns generated.
5. Output the most *k* frequent patterns among generated patterns from the FP-tree, referring to the Reduction Array.

4.4. TF^{2}P-growth

Like other Top-k mining concept algorithms, users still have to specify the value *k* before execution of Top-k FP-growth proposed in 4.3. To resolve this problem, in this section we propose TF^{2}P-growth based on Top-k FP-growth. TF^{2}P-growth mines frequent patterns with descending order of support values without specifying any thresholds. The mined frequent patterns are output to users every n_c-patterns where n_c is the chunk size. By default, n_c is set to 1000[1].

The process of the TF^{2}P-growth algorithm described below. Users initiate TF^{2}P-growth without specifying any thresholds. Then, it sequentially returns the Top 1000

[1] Users may change the chunk size *n* to any number.

patterns, Top 1001 to 2000 patterns, Top 2001 to 3000 patterns, *etc*. Once users have received the Top 1000 patterns, they can initiate interpretation of these patterns. When the user is satisfied with the mined frequent patterns, they can terminate mining, or they may terminate mining whenever they want.

As the initial results are the Top-n_c patterns, which can be set to a small number, it is possible to shorten the response time from initiation of the mining until generation of the first part of the results.

4.4.1. TF^{2}P-growth algorithm. The TF^{2}P-growth algorithm is shown below.

INPUT *TDB*
OUTPUT frequent patterns (descending order of support)
METHOD
1. Scan *TDB* to count support of all 1-itemsets.
2. Set i to 1.
3. Set n to $n_c \times i$, where n_c is 1000 by default.
4. Set *Border_sup* to the support value of the *n*-th 1-itemset (descending order). Then, generate an F-list.
5. Construct an FP-tree according to the F-list.
6. Generate frequent patterns from the FP-tree. During generation, at every node-link traversal, re-calculate *Boundary_sup* for reduction of the number of patterns generated.
7. Output the $(n_c \times (i-1)+1)$th to $(n_c \times i)$th frequent patterns among the patterns generated from the FP-tree.
8. Increment i, then go to 3.

5. Performance Evaluation

In this section, we present performance evaluations of TF^{2}P-growth. We evaluated TF^{2}P-growth with regard to (1) comparison of the execution time of FP-growth, Top-k FP-growth, and TF^{2}P-growth, and (2) scalability of TF^{2}P-growth. We used T10I4D1000k, by IBM quest synthetic data generation code[11], as a dataset.

All of the experiments were performed on a 2.4 GHz Pentium4 PC machine with 1 GB of main memory, running RedHat Linux 9.0, kernel version 2.4.20. All of the programs were written in C++ and complied with gcc3.2.2.

5.1. Comparison of Execution time

We compared the performance of TF^{2}P-growth with both FP-growth and Top-k FP-growth. In real data mining applications, users have difficulty in setting the *min_sup* or the value *k* for the first time. Therefore, users must execute frequent pattern mining recursively changing the *min_sup* or the value of *k*.

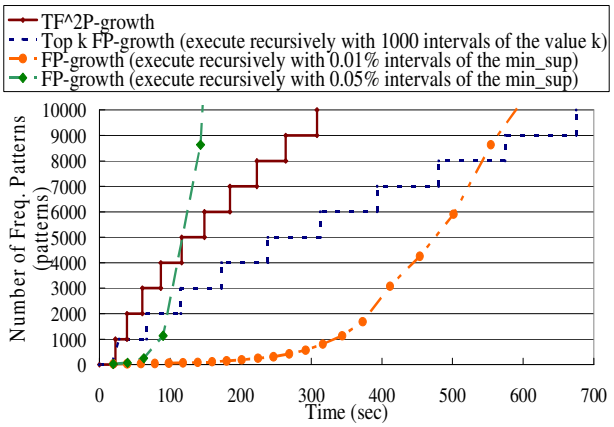

Figure 5. Execution Time *vs.* Mined Frequent Patterns

In this evaluation, we measured the execution time of FP-growth and Top-k FP-growth in the following manners. In the case of FP-growth whose threshold is the *min_sup*, we measured the execution time in two patterns—executing FP-growth recursively in the range of $0.3\% \geq min\_sup \geq 0.1\%$ with an interval equal to (1) 0.01% and (2) 0.05%. In the case of Top-k FP-growth whose threshold is the value *k*, we measured the execution time in the following pattern—executing Top-k FP-growth recursively by changing the value *k* from 1,000 to 10,000 with an interval of 1000.

The experimental results are shown in Figure 5.

First, as it is difficult to set the appropriate *min_sup* when users execute FP-growth, *min_sup* must initially be set high. Then, users can decrease the value of *min_sup* slowly, *e.g.*, with an interval of 0.01%. However, this results in slow generation of frequent patterns. On the other hand, using TF^{2}P-growth, users can obtain larger numbers of frequent patterns in the same time in comparison with using FP-growth[2].

Second, using TF^{2}P-growth, users can obtain large numbers of frequent patterns in the equivalent execution time in comparison with using Top-k FP-growth recursively. This is because TF^{2}P-growth reduces the number of *TDB* scans by reusing the F-list generated in the first cycle of Top-k FP-growth.

These experimental results indicate that use of TF^{2}P-growth has the advantage of requiring no threshold to be set. Moreover, users can obtain more frequent patterns in the equivalent execution time in comparison with using FP-growth or Top-k FP-growth.

---

[2] If users could know a relation of *min_sup* and the number of mined frequent patterns before the initiation of mining, FP-growth drives much more patterns in comparison to TF^{2}P-growth such as the result of the interval 0.05% in Figure 5. However, it is general that users don't know the relation.

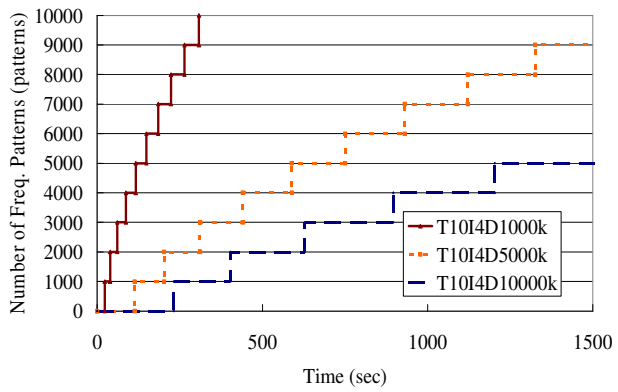

Figure 6. Execution time with different numbers of transactions

5.2. Scalability of TF^{2}P-growth

We also evaluated the scalability of TF^{2}P-growth. In this evaluation, we prepared the same dataset with different numbers of transactions, T10I4D5000k and T10I4D10000k. The experimental results are shown in Figure 6.

As shown in Figure 6, the execution time of TF^{2}P-growth increases linearly as the dataset size increases. This means that even if users apply TF^{2}P-growth for a very large dataset, they can obtain the Top 1000 patterns and can avoid the situation where they may obtain only a small number of frequent patterns after a long computation time.

6. Conclusions

In this paper, we proposed a new frequent pattern mining algorithm, called "TF^{2}P-growth," which mines frequent patterns in descending order of support without specifying any threshold values. By applying the proposed algorithm to the dataset T10I4D1000k, we confirmed that the following two advantages. First, users can execute the mining process without specifying any threshold values. Second, users can mine more frequent patterns in comparison with those mined by executing FP-growth recursively with changing *min_sup*. For example, TF^{2}P-growth mines the 9,000 most frequent pattern twice as fast as FP-growth executed recursively by changing *min_sup* from 0.3% to 0.1% with 0.01% intervals.

Future work will involve adopting the maximal or closed pattern mining concept into the proposed algorithm, and parallelizing the proposed algorithm.

Acknowledgments

This research was funded in part by both "e-Society: the Comprehensive Development Foundation Software" of MEXT (Ministry of Education, Culture, Sports, Science, and Technology) and "21-century COE Programs: ICT Productive Academia" of MEXT.

References

[1] R.Agrawl and R.Srikant, "Fast Algorithms for Mining Association Rules ," In Proc. of VLDB '94, pp. 487-499, Santiago, Chile, Sept. 1994.

[2] J. Han, J. Pei and P.S. Yu, "Mining Frequent Patterns without Candidate Generation," In Proc. of the ACM SIGMOD Conference on Management of Data, pp.1-12, 2000.

[3] R.J.Bayard, "Efficiently Mining Long Patterns from Databases ," In Proc. of the ACM SIGMOD Conference on Management of Data, pp. 85-93, 1998.

[4] G.Grahne and J.Zhu, "High Performance Mining of Maximal Frequent Itemsets," In Proc. of SIAM'03 Workshop on High Performance Data Mining, 2003.

[5] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets," In Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, 2003.

[6] J.Pei, J.Han and R.Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets," In Proc. of DMKD'00, 2000.

[7] A.W.-C Fu., R.W.-W. Kwong and J.Tang, "Mining N-most Interesting Itemsets," In Proc. of the ISMIS'00, 2000.

[8] J. Han, J. Wang, Y. Lu and P. Tzvetkov, "Mining Top-k Frequent Closed Patterns without Minimum Support," In Proc. of IEEE ICDM Conference on Data Mining, 2002.

[9] J.S. Park, M.Chen, P.S. Yu, "An effective hash-based algorithms for mining association rules," In Proc. of the ACM SIGMOD Conference on Management of Data, pp.175-186, 1996.

[10] A. Savasere, E. Omiecinski, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," In Proc. of VLDB'95, pp.423-444, 1995.

[11] IBM Quest Data Mining Project. Quest synthetic data generation code. http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html