Mining Fault-Tolerant Frequent Patterns in Large Databases

Shen-Shung Wang* and Suh-Yin Lee
Department of Computer Science and Information Engineering
National Chiao Tung University, Taiwan 30050, R.O.C.
Email: {sswang, sylee}@csie.nctu.edu.tw
Workshop on Databases and Software Engineering

Abstract

In view of real world data may be interfered with noise which leads data to contain faults. Besides, we may hope that the knowledge discovered is more general and can be applied to find more interesting information. Hence, FT-Aprori was proposed for fault-tolerant data mining to discover information over large real-world data. However, FT-Apriori which generates and tests candidates based on Apriori property is not so efficient.

In this paper, we develop memory-based algorithm FTP-mine which is based on the concept of pattern growth to mine fault-tolerant frequent patterns efficiently. In FTP-mine the table, STable, is designed to count the item support and FT-support of the k-length patterns which have the same prefix of length k-1. As to mining in a large database which is too large to fit in memory, FTP-mine also can be adopted by means of database partition. Since there might exist a large number of fault tolerant frequent patterns and some may be contained in others, we also focus on the finding of maximal FT-frequent patterns by extending the FTP-mine algorithm. Our study shows that FTP-mine has higher performance than FT-Apriori in various datasets. The empirical evaluations show the proposed method has good linear scalability and outperforms than FT-Apriori in various settings in the discovery of FT-frequent pattern.

Keywords: Data mining, Fault tolerant frequent pattern, FT-contain, support

1. Introduction

Data mining [16], which discovers non-trivial and potential useful information in large databases, has been an active research topic in recent years. The discovered knowledge can be applied to information management, query processing, decision making, process control and many other applications. The domain which data mining involves is very extensive, such as database systems, artificial intelligence, machine learning, statistics, and data visualization. However, much research in this field has focused on the mining of frequent patterns[2, 5, 13], because frequent pattern mining plays a fundamental role in many data mining skills, such as association rules [6, 7, 12], sequential patterns [3, 4, 10], maximal patterns[8, 11, 14], closed patterns[9], classification and clustering.

Generally speaking, algorithms to mine frequent patterns efficiently can be classified into

three categories. The first category is candidate generation-and-testing. The well-known algorithm is Apriori which is based on an anti-monotone Apriori property [6]. The main idea is that a pattern can not be frequent if there exists a sub-pattern which is not frequent. In other words, once a pattern is not frequent, its super-pattern would not be frequent any more. According to this idea, the algorithm reduces the size of candidates to be generated. However, it is still unavoidable to generate a large number of candidates, especially when 2-itemsets are generated or when the length of frequent patterns is long.

The second category is pattern growth method of which the typical algorithm is FP-growth [6]. Instead of generating candidates, FP-growth builds a FP-tree to compact the information of transactions and find frequent patterns by traversing FP-tree. Once the FP-tree is too large to fit into memory, FP-growth finds local frequent patterns in partition databases which are divided by prefix path of FP-tree to assemble to longer ones. However, if the database is large and sparse, FP-tree will be large and the space requirement for recursion is a challenge.

Another concept proposed recently is space-preserving method which suggests loading the transactions into memory initially. The typical approach is H-mine [2], which designs a hyperlink structure, H-struct, to dynamically adjust links in the mining process. Undoubtedly, the execution sequence of H-mine is some kind of a pattern growth approach. Unlike FP-growth, H-mine neither maintains FP-tree nor creates physical databases. However, the method has to maintain a header table in each level and adjusts links to form a queue which collects the transactions containing the same prefix before counting the supports of items.

Although much work has been done on frequent pattern mining, little research has been devoted to fault-tolerant frequent mining. Fault-tolerant frequent pattern mining is to discover approximate patterns from the real-world data, which is tend to be dirty and diverse. In some situation, data may be disturbed by noise or some uncontrolled environment factors. We would like to find the frequent patterns which may contain some faults. For example, because of mutation, the nucleic acid of DNA sequences in a gene database may be modified, or may be incorrectly distinguished during the experiment. So fault-tolerant frequent pattern mining can be applied to the datasets, such as scientific dataset or web log which contains unanticipated errors.

In addition, sometimes we may want to find more general rules. Take data mining for example, the related courses are AI, data structure, algorithm, and DBMS. It is not easy to discover the association between the four courses and data mining, because not all students take and do well in all of the four courses. However, if a user specified support is too low, many rules which are irrelative or uninteresting will be found. So fault-tolerant frequent pattern mining is necessary.

Jian Pei pointed out the problems and challenges of fault-tolerant frequent pattern mining [1] and extended Apriori to FT-Apriori to discovery them. However, it still needs to generate a large number of candidates, which wastes time in combining and checking sub-patterns. In this thesis, we adopt the concept of space-preserving and propose an algorithm, FTP-mine, which caches transactions in main memory and processes the data more efficiently by maintaining one table.

The remainder of this paper is organized as follows. Some background knowledge and related work in fault-tolerant frequent pattern mining are introduced in Section 2. FTP-mine algorithm is presented in Section 3. In Section 4, we describe experimental results that validate the effectiveness of FTP-mine. In Section 5, the conclusion and future works are presented.

2. Problem Definition

2.1 Frequent pattern mining

Let $I=\{i_1, i_2, \ldots, i_n\}$ be a set of items and $|I|$ means the cardinality of I. An itemset is a non-empty subset of I, and k-itemset is an itemset with k items. A transaction T=(tid, *t*) is 2-tutple, where tid is transaction-id and $t \subseteq I$. We say a transaction T contains itemset X if $X \subseteq t$.

A transaction database TDB is a set of transactions. The total number of transactions in TDB containing itemset X is called support of X, denoted as support(X). The *min_sup* is a user specified support threshold to decide if an itemset is a frequent pattern by support(X) ≥ *min_sup*. The problem of frequent pattern mining is to find the complete set of frequent patterns in a given transaction database with respect to a given support threshold. The set of all candidates with k items is denoted as C_k and frequent patterns with k items is denoted as L_k.

Example 2.1 (Frequent pattern):

An example transaction database TDB is given in Table 2.1, and user specified *min_sup* is 2. An itemset $X=\{a, d, e\}$ is contained in transactions whose TID are 200 and 500 and the support of X, support(X)=2. Because of support(X)≥*min_sup*, X is a frequent pattern.

TID	Items
100	*b d e g*
200	*a c d e*
300	*b e f g h*
400	*c g h*
500	*a b d e h*

Table 2.1 Example transaction database TDB

2.2 Fault-tolerant frequent pattern

Because faults are allowed, we notate the fault tolerance as δ (δ>0). Besides, the definition of **contain** need to redefined to **FT-contain**. A transaction T=(tid, *t*) is said to **FT-contain** itemset X iff there exists an itemset *x*, which is a subset of X and also a subset of *t* at the same time, such that $|X|-|x| \leq \delta$. FT-support of pattern X, denoted as $sup^{FT}(X)$, is the total number of

transactions in transaction database FT-containing itemset X. For an itemset X to be a fault-tolerant frequent pattern, except for FT-support counting, the support of each item in the FT-containing transactions also needs to be checked. B(X) is the collection of transactions which are FT-containing itemset X, and an itemset X is a fault-tolerant frequent pattern (denoted as FT-frequent pattern) iff

1. $\sup^{FT}(X) \geq min\_sup^{FT}$

2. for each item $x \in X$, $\sup_{B(x)}(x) \geq min\_sup^{item}$, where $\sup_{B(x)}(x)$ is the number of transactions containing item x in B(X)

In other words, there are at least min_sup^{FT} transactions FT-containing X, and each item of X must have appeared at least min_sup^{item} times in these FT-containing transactions.

Example 2.2(Fault tolerant frequent pattern):

Let us take TDB in Table 2.1 for example. Suppose the frequent-item support threshold min_sup^{item}=2, the FT-support threshold min_sup^{FT}=3 and one fault allowed (δ=1). X= {a, b, d, e} is FT-contained by transactions 100, 200 and 500. That is to say $\sup^{FT}(X)$=3, and the B(X) includes transaction 100, 200, and 500. Besides, each item in X appears at least in two transactions in B(X), respectively. Thus, {a:2, b:2, d:3, e:3 } is an FT-frequent pattern.

2. 3 Frequent pattern vs. FT-frequent pattern

Table 2.2 shows the complete set of frequent patterns and FT-frequent patterns. The *min_sup* of frequent patterns is 2, and the min_sup^{item}, min_sup^{FT} and δ of FT-frequent pattern is 2, 3, and 1 respectively. From this table we can discover the truth that fault-tolerant frequent mining can find more and longer patterns with high support than frequent pattern mining. Besides, since the FT-frequent patterns must be δ+1 to make sense, so the FT-frequent patterns do not include 1-itemset.

	Frequent patterns	FT-frequent patterns(δ=1)
1-itemset	a, b, c, d, e, g, h	
2-itemset	$\{a, d\}$ $\{a, e\}$ $\{b, d\}$ $\{b, e\}$ $\{b. g\}$ $\{b, h\}$ $\{d, e\}$ $\{e, g\}$ $\{e, h\}$ $\{g, h\}$	$\{a, b\}$ $\{a, c\}$ $\{a, d\}$ $\{a, e\}$ $\{a, g\}$ $\{a, h\}$ $\{b, c\}$ $\{b, d\}$ $\{b, e\}$ $\{b, g\}$ $\{b, h\}$ $\{c, d\}$ $\{c, e\}$ $\{c, g\}$ $\{c, h\}$ $\{d, e\}$ $\{d, g\}$ $\{d, h\}$ $\{e, g\}$ $\{e, h\}$ $\{g, h\}$
3-itemset	$\{a, d, e\}$ $\{b, d, e\}$ $\{b, e, g\}$ $\{b, e, h\}$	$\{a, b, d\}$ $\{a, b, e\}$ $\{a, c, h\}$ $\{a, d, e\}$ $\{a, e, g\}$ $\{a, e, h\}$ $\{b, d, e\}$ $\{b, d, g\}$ $\{b, d, h\}$ $\{b, e, g\}$ $\{b, e, h\}$ $\{b, g, h\}$ $\{c, d, g\}$ $\{c, d, h\}$ $\{c, e, g\}$ $\{c, e, h\}$ $\{d, e, g\}$ $\{d, e, h\}$ $\{d, g, h\}$ $\{e, g, h\}$
4-itemset		$\{a, b, d, e\}$ $\{b, d, e, g\}$ $\{b, d, e, h\}$ $\{b, d, g, h\}$ $\{b, e, g, h\}$ $\{d, e, g, h\}$
5-itemset		$\{b, d, e, g, h\}$

Table 2.2 the complete set of frequent patterns and FT-frequent patterns

3. Mining FT-frequent Pattern

In this section, the FTP-mine algorithm will be presented. In Section 3.1, we assume the transaction database can fit into memory, and use STable to count the FT-support and item support. In Section 3.2, large database is taken into account, and the FTP-mine is extended to mining FT-frequent patterns in large databases. Finally, the method to mine maximal FT-frequent patterns is also introduced in Section 3.3

3.1 FTP-mine

The candidate-generation-and-test approach will produce a large number of candidates which need extra time in generating candidates and checking whether they are frequent. Furthermore, because mismatches are allowed in FT-frequent pattern mining, FT-frequent pattern mining will generate more candidates and longer patterns than frequent patterns mining. In this way, the mining process will become inefficient. Therefore, we propose FTP-mine which takes advantage of pattern growth and space-preserving to mine FT-frequent patterns. In FTP-mine, we apply the pruning strategy to reduce the comparison times. This is justified by the following lemmas:

Lemma 1:

An itemset, *prefix*, which attempts to append suffix item *s* to become FT-frequent pattern with |*prefix*|+1 items(denoted as *prefix'*), and MN is the number of items which appear both in *prefix* and the transaction *t*. *t* FT-contains the pattern (*prefix* $\cup$ *s*) iff

1. MN > |*prefix*| - δ ($s \notin t$).

or

2. MN $\geq$ |*prefix*| - δ ($s \in t$).

Proof:

When *s* appends to *prefix*, if $s \in t$, MN will increase, else the number of faults will increase. We discuss these two conditions respectively.

Condition 1:

Because $s \notin t$ and in this stage the pattern length will increase to |*prefix*| + 1. At the moment there are (|*prefix*| + 1- MN) faults and only δ faults are allowed. If *t* FT-contains *prefix'*, then (|*prefix*| + 1- MN)$\leq \delta$ which can lead to (|*prefix*| - MN) < δ.

Condition 2:

Because $s \in t$ and in this stage the length of the pattern will grow to |*prefix*| + 1. At the moment there are (|*prefix*| + 1- (MN+1)) faults and only δ faults are allowed. If *t* FT-contains *prefix'*, then (|*prefix*| - MN)$\leq \delta$. ?

Lemma 2:

During the process of pattern growth, appending *s* to *prefix* continuously, *t* does not FT-contain the patterns any more which start with *prefix*, once MN < |*prefix*| - δ.

Proof:

Once MN <|*prefix*|-δ, the length of prefix in the next round will increase. No matter how MN increases, it is always smaller than |*prefix*|-δ. That is to say that there are (|*prefix*|-MN) faults which exceed the fault tolerant threshold δ and *t* will not contain any itemset which starts with *prefix* . ?

Lemma 3

If *t* does not contain *s* and MN = | *prefix*| - δ, then the extending pattern (*prefix* ∪ *s*) will change from FT-frequent into none FT-frequent. In other words, *t* FT-contains *prefix*, but does not FT-contain (*prefix* ∪ *s*). The *prefix* is a FT-frequent pattern previously, but due to the increasing length of pattern (*prefix* ∪ *s*), the pattern would not be FT-frequent any more. ?

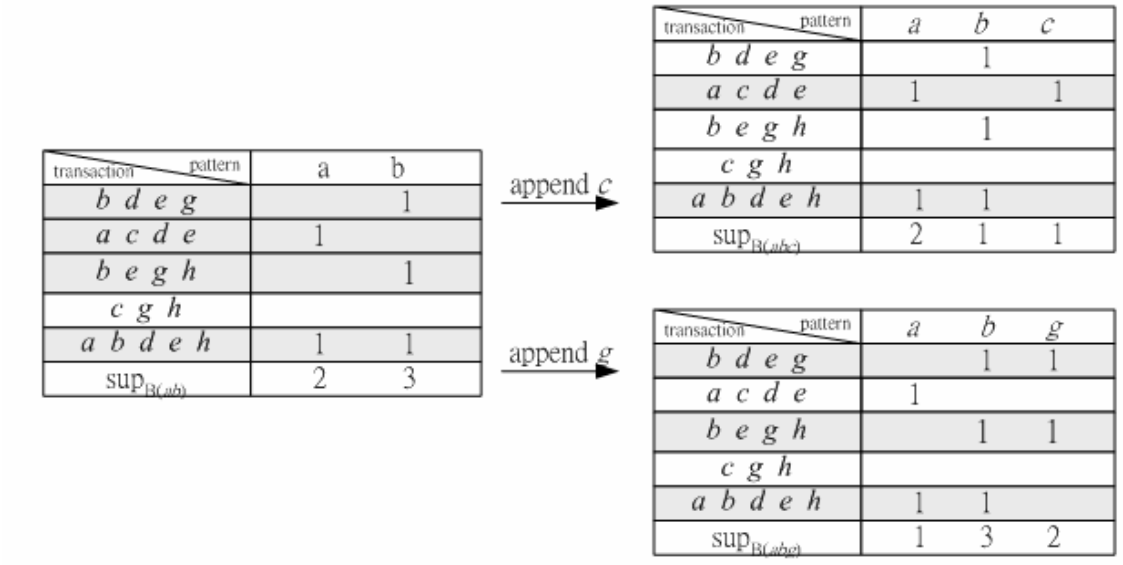


Fig 3.2 The changes of FT-containing transactions after appending *s*

From Fig 3.2, we can discover the fact that the number of FT-containing transactions will decrease as the suffix item (denoted as *s_item* briefly) appends to the *prefix*. Besides, each item support of *prefix* will reduce after appending *s_item* as the number of FT-containing transactions decreases. Therefore we have to take the item support of *prefix* into account when we design the structure to count all kinds of supports of *s_item*s with the same *prefix*.

We first consider the case that all the transactions can load into main memory. We use depth first search execution sequence to examine the lexicographic sequence to demonstrate the finding of FT-frequent patterns. Only one table, STable, need to be maintained in memory. STable is to count FT support and item support of each *s_item* which will append to *prefix*. Because the appending *s_item* may cause a FT-containing transaction not to FT-contain the pattern (*prefix* ∪*s_item*) any more, STable also records item support of the *prefix* for each

s_item. We will extend the proposed mechanism to apply to a large database later. In brief, there are three steps for each transaction in the scheme.

Step 1: Find the number of matching items of a transaction which FT-contains *prefix*.

We first count how many items match *prefix* in the transaction. However there are two phases to count the matching items of *prefix*.

Phase 1(initial):

Because there are δ faults allowed in *prefix*, we compare the first δ items of *prefix* with items whose lexicographic order is less than the δth item of *prefix* in the transaction and count how many items matching (denoted MN).

Phase 2(pattern growth):

We continue to check the patterns from length δ to |*prefix*|. When the MN is figured out, we can decide whether the pattern of *prefix* should continue to extend according to Lemma 2. Because there are over δ faults, the transaction does not FT-contain the *prefix*.

Example 3.1 (Find MN) A *prefix* P (*a, b, c, d, e*) and a transaction *t* (*a, c, e, f, g*) is given. Let fault tolerance δ=1.

Phase 1:

Because δ=1 and *a* is matching. MN=1.

Phase 2:

Let us think of *b* in P. Because it is not contained in *t*, we go on to check next item in P and there is a mismatch to *t*. Considering item *c*, Because *t* contains *c*, MN=2. We continue to take item *d* into account, MN= |*abc*|-δ=2 and *t* does not contain *d*, that is to say there are 2 faults. The transaction is impossible to FT-contain P according to Lemma 3, and it need not to be checked any more.

Step 2: Count all kinds of supports of suffix.

After MN of *prefix* has been calculated, the extending *s_item*s are taken into account to generate the patterns whose length is |*prefix*|+1 in the next step. At first, if MN is less than |*prefix*|-δ, the transaction will not to be checked, because the growth of pattern will lead to the transaction does not FT-containing the transaction anymore. Next, we compare each *s_item* to the items which are not checked in the transaction previously. If matching, increase the item support and FT-support of the item; if not matching but MN is greater than |prefix|-δ (that means the itemset (*prefix*∪ *s_item*) is still a FT-frequent pattern, because this fault still can be tolerated.), we add the FT-support of that itemset; if not matching but MN is equal to |*prefix*|-δ(that means appending this item to *prefix* will lead the itemset to be not FT-frequent), the item support of the items which appear both in *t* and *prefix* need to be decreased for the *s_item* in STable. The rules to fill the STable are summarized as follows

1. If MN <|prefix|-δ then read next transaction.

2. If MN >|prefix|-δ

   increment the FT-support of each suffix item.

   If item s∈ t, increment the item support.

3. If MN =|prefix|-δ

   If item s∈ t, increment the FT-support and item support of each suffix item.

   If item s∉ t, decrement the item support of the items in (*prefix*∩*t*).

Step 3: Decide the FT-frequent pattern

This step is to filter the FT-support and item support of each item in STable with min_sup^{FT} and min_sup^{item} respectively. If the conditions above are met, the related item supports of *prefix* are still checked in STable. We describe the main idea by running the following example.

Example 3.2 (FTP-mine): Let us mine FT-frequent patterns in the transaction database TDB (first two columns of table 1) in Fig 3.3, which is sorted by lexicographic order, with min_sup^{item}=2, min_sup^{FT}=3, and δ=1.

TID	Items	Frequent items
100	*b d e g*	*b d e g*
200	*a c d e*	*a c d e*
300	*b e f g h*	*b e g h*
400	*c g h*	*c g h*
500	*a b d e h*	*a b d e h*

Fig 3.3 Transaction database TDB and frequent items

In the beginning, we scan the TDB once to find global frequent items with the item support threshold, min_sup^{item}. The complete set of global frequent items{*a*:2 , *b*:3, *c*:2, *d*:3, *e*:4, *g*:3, *h*:3} can be collected. Due to FT-Apriori property, *f*'s support does not achieve frequent item support threshold, so *f* will be pruned and will not be loaded to memory. The initial stage of STable to mine *a-prefix*, and transactions in memory are shown in Fig 3.4.

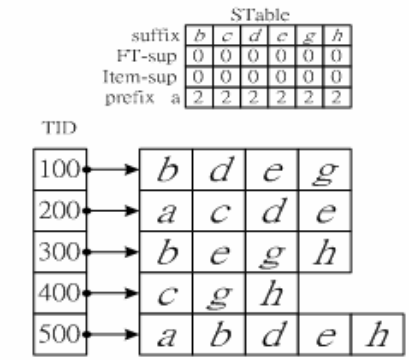


Fig 3.4 Initial stage for STable, and the transactions in memory to mine *a-prefix*

Frequent items of each transaction are loaded into memory and the execution sequence is by the lexicographic order of frequent items. The complete set of frequent patterns is divided into

several sets by the *prefix*. For example, *a-prefix* which means the FT-frequent patterns which contain *a*, and the set of *a-prefix* frequent patterns also can be separated into subsets by the *prefix*, such as *ab*-prefix, *ac*-prefix, …, etc. As the same way, the *b-prefix* means the FT-frequent patterns which include *b* or the items whose lexicographic order is greater than.

Initially, because the length of the FT-frequent pattern must be at least (δ+1) to make sense, the depth first execution sequence is adopted to decide whether the *prefix* is appended by *s_item*s. If the length of *prefix* is less than δ, then append *s_item* and go a step further to increase the length of prefix by depth first traversal method. We start *prefix* with δ items to scan the transactions in memory to determine length-(δ+1) patterns, and each item support of *prefix* in STable is global item support initially. For example, if δ=2, the FT-frequent patterns of *ab*-prefix are discovered, and then find the FT-frequent patterns of *ac-prefix*, *ad-prefix*, *ae-prefix*, …, and *ah-prefix* are found subsequently.

To take *a*-prefix for example, the mining process is to fill out the STable to check FT-supports and item support furthers. Considering TID=100, MN=0 can be calculated first by the step 1 in FTP-mine. Because MN = |*prefix*| - δ, the transaction need to be checked further. If the transaction contains the suffix item, then increase the items support and FT-support of that *s_item*, else decrease the item support of items in (*prefix*∩*t*). FT-support, item support of *s_item* and the item supported of each item in *prefix* after checking TID=100 can be consulted in Table *1* of Fig 3.6. Considering TID=200, because MN=1>|*prefix*|- δ, that means one fault can be allowed further, The FT-support of each *s_item* is increased by 1 no matter the transaction contains *s_item* or not. If the *s_item* is contained by the transaction, increase the item support of that s_*item*. The result after scanning TID=200 is consulted in Table *2* of Fig 3.6.

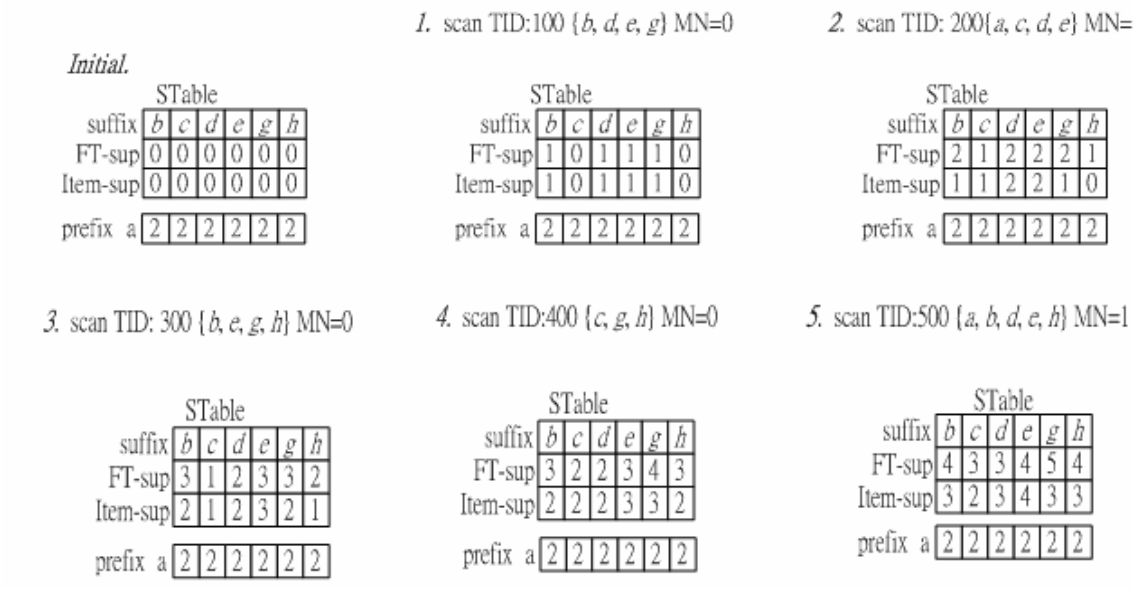


Fig 3.6 The flow chart for mining *a*-prefix

After all transactions are scanned, the FT-support and item support of each *s_item* can be resulted. Except for checking FT-support and item support of *s_item*, each item support of *prefix* for *s_item* can be found in STable, Table 5 of Fig 3.6. For example, the item support of a in $\{a, b\}$ is 2. Now the FT-frequent pattern $\{a: 2, b: 3\}$, $\{a: 2, c: 2\}$, $\{a: 2, d: 3\}$, $\{a: 2, e: 4\}$, $\{a: 2, g: 3\}$, and $\{a: 2, h: 3\}$ are generated. There is a phenomenon that when the pattern grows from length δ to length $\delta+1$, the item support of *prefix* must be the same as global item support. Because if the transaction contains one item of the pattern of length $\delta+1$, the transaction FT-contains the pattern. So each item support of *prefix* will not be influenced by appending the *s_item*.

Depth-first traversal path is adopted to go on to check patterns whose *prefix* is $\{a, b\}$. We also take the first transaction $\{b, d, e, g\}$ to explain briefly. Firstly, we find the number of items in transaction which match *prefix* (MN=1). Because MN = $|ab|-\delta$, the *s_item*s still need to be checked. If the transaction contains the *s_item*, add FT-support and item support to that corresponding item, such as d, e, and g. If the transaction doesn't contain the *s_item*, the item support of item in $(prefix \cap t)$ should be decreased by 1 for the *s_item* in STable. After scanning first transaction once, the table *1* in Fig 3.7 can be filled. The process of filling table shows as follow.

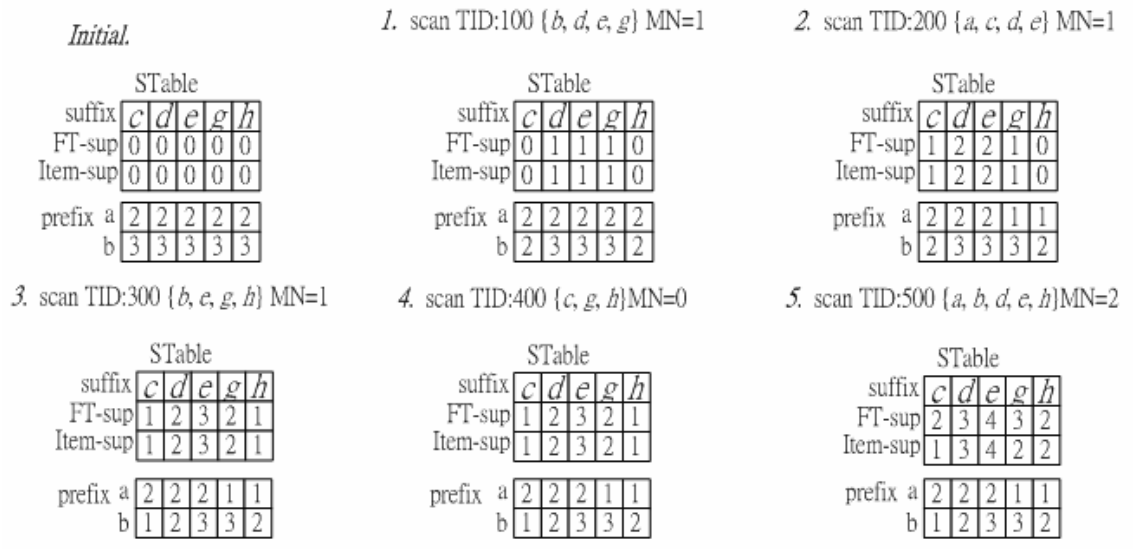


Fig 3.7 The flow chart for mining ab-prefix

In terms of the Table 5 of Fig 3.7, the step 3 in FTP-mine is used to filter FT-support and item support in STable with min_sup^{FT} and min_sup^{item}. Besides, we also have to check item supports of *prefix* which accompanies the *s_item*. For example, although g achieves min_sup^{FT} and min_sup^{item} threshold, the difference of corresponding item support to *prefix* are still taken into account. The item support of a in pattern $\{a, b, g\}$ is 1 and does not achieve min_sup^{item}, so $\{a: 1, b: 3, g: 2\}$ is not a FT-frequent pattern.

Lemma 4: Given a *prefix* and *s_item*, the set of transactions which FT-contain *prefix* is the superset of transactions which FT-contain (*prefix* $\cup$ *s_item*).

Proof:

It is heuristic that if $|prefix \cup s\_item| \leq \delta$, then all transactions FT-contain *prefix*. Now let us consider when $|prefix \cup s\_item| > \delta$. There are only δ faults allowed for each transaction. As the pattern grows, if the transaction contains the *s_item* and there are faults less than or equal to δ, then *prefix* must have chance to grow. Once the transaction does not FT-contains the *s_item*, the transaction may not FT-contain *prefix* any more because of the number of faults. So the set of transactions which FT-contain will reduce as the pattern grows. ?

In the stage, {*a*: 2, *b*: 2, *d* 3} and {*a*: 2, *b*: 3, *e*: 4} are determined to be FT-frequent patterns. Based on FT-Aproiri property [1], there is no need to continue traversing the *abc-prefix*, *abg-prefix* and *abh-prefix* subtrees. Besides, according to Lemma 4, since the *s_item*s are not FT-frequent in *ab-prefix*, the *s_item*s are neither FT-frequent in *abd-prefix* nor *in abe-prefix*. For example, *h* will not be FT-frequent in *abd-prefix*, because the set of transactions which FT-contain {*a*, *b*} is a superset of those transactions which FT-contain {*a*, *b*, *d*}. Since *h* is not FT-frequent in transactions which FT-contain {*a*, *b*, *h*} is also not FT-frequent in transactions which FT-contain {*a*, *b*, *d*}. So when we further check *abd-prefix*, we just have to check *s_item* *e*. Similarly, the mining goes along the traversal path, and then a FT-frequent pattern {*a*, *b*, *d*, *e*} is generated. There are 48 FT-frequent patterns of which the length of is greater than δ and {b, d, e, g, h} is the longest pattern.

FT-mine algorithm

Input: Transaction database TDB, frequent item support threshold min_sup^{item}, FT-support threshold min_sup^{FT}, and fault tolerance δ.
Output: The complete set of FT-frequent patterns.
Method:

1. Scan TDB once. Find the set $L_1 = \{i_1, i_2, \ldots, i_n\}$ of global frequent items with the item support $s_1, s_2, \ldots$ and s_n respectively, and sorted by lexicographic order. An item *i* is global frequent iff $sup(i) \geq min\_sup^{item}$.
2. Load those items which are global frequent in a transaction into memory. All transactions which are composed of global frequent items in memory are denoted MDB.
3. **For each** item $i \in L_1$ **do begin**
   DepthFirst (*i*, L_1| *i*); // L_1| *i* is the set of items in L_1 whose lexicographic order are greater // than *i*

end

Subroutine *DepthFirst*(*Prefix*, *Suffix*)

Input: $Prefix = \{i_{a1}, i_{a2}, \ldots, i_{ap}\}$ and $Suffix = \{i_{b1}, i_{b2}, \ldots, i_{bq}\}$ are two sets of items where $i_{ak} < i_{a(k+1)}$, $i_{bh} < i_{b(h+1)}$ and $(i_{ap} < i_{b1})$.

if |*Prefix*|<δ **then**
  For each item $i \in Suffix$ **do begin**
    DepthFirst (*Prefix*$\cup i$, *Suffix*|*i*);

```
    end
  end
  φ=Pattern-Grow ( Prefix, Suffix);
  For each pattern i∈φ do begin
  DepthFirst(Prefix∪i, φ|i);
  end
```

Subroutine *Pattern-Grow*(*Prefix*, *ISUP*, *Suffix*)

Input: *Prefix*⊆I , *Suffix*⊆I
Output: a set of items which can append to become FT-frequent. To collect the set of FT-frequent patterns and record whose item supports.

```
  Create STable.
 /*initialize the item support of Prefix in STable*/
  For each item i∈ Suffix do begin
     For each item j∈ Prefix do begin
     STable(i).j=Prefix(j).s    // Prefix(j).s is the item support of j in Prefix
                                // if |Prefix|=δ, Prefix(j).s=s_j
    end
  end
  φ={}
  For each transaction t∈ MDB do begin
     To find the cardinality of t∩Prefix (denoted as MN )
       if (MN<|Prefix| -δ) then
            continue to read next transaction
       end
       if MN>|Prefix| -δ then
          STable(i).ftsup++;
            For each item i∈ Suffix do begin
                 if i∈ t then
                 STable(i).itemsup++;
                end
             end
        else if MN=|Prefix| -δ then
             For each item i∈ Suffix do begin
                  if i∈ t then
                     STable(i).itemsup++;
                     STable(i).ftsup++;
                  else
                     STable(i).j--, where j∈ t∩Prefix.
                  end
             end
         end
  end
       For each item i∈ Suffix do begin
          if STable(i).itemsup≥min_sup^item and STable(i).ftsup≥min_sup^FT
           and each item j∈ Prefix STable(i).j ≥ min_sup^item then
                insert Prefix∪i into the set of FT-frequent patterns and record the item
                support of the pattern {STable(i).j1, STable(i).j2, …, STable(i).jk,
                STable(i).itemsup};
                   φ←φ∪i;
          end
       end
  return φ
```

3.2 Mining FT-frequent patterns in Large Databases

FTP-mine is efficient when the transactions and tables can fit into main memory. However, the transaction database is often too large for FTP-mine to load into memory. Analogous to H-mine [2] approach, the database is divided into several sub-databases to mine local FT-frequent patterns with local min_sup^{FT}. After local FT-frequent patterns are collected, scan database again to check whether the local FT-frequent patterns are the global FT-frequent patterns. We describe the method in details as follows.

Suppose that there are n transactions in the transaction database, TDB, and the min_sup^{item} and min_sup^{FT} are the support thresholds. We divide TDB into k partitions (P_1, P_2, ..., P_k), where P_i($1 \le i \le k$) has n_i transactions, and $\sum_{i=1}^{k} n_i = n$. First, we scan TDB once to find global frequent items L_1. For each partition database Pi, we load items which are global frequent in transactions into memory. Secondly, FT-mine is applied to mine the local potential FT-frequent patterns with local $min\_sup_i^{FT} = \left\lfloor min\_sup^{FT} \times \frac{n_i}{n} \right\rfloor$. In this stage, the item support would not be checked. If the item support is taken into account, some FT-frequent patterns may be ignored because of the item support threshold. For example, TDB is divided into 2 equal partitions, and we assume the the local $min\_sup^{item}=3$. If the pattern $\{a, b, c, d\}$ has achieved the local min_sup^{FT}, and the item supports of the pattern in each partition are {a:1, b: 4, c: 2, d:5} and {a:6, b:3, c:5, d:2} respectively. If we consider the item support threshold in mining partition databases, the pattern will be ignored. However {a, b, c, d} is a FT-frequent pattern when the partitions are assembled. After all local potential frequent FT-frequent patterns are found in each Pi, we scan database again to decide whether a local potential FT-frequent pattern is a global FT-frequent pattern by scanning database once again with min_sup^{item} and min_sup^{FT}.

3.3 Mining maximal FT-frequent patterns

Because the number of FT-frequent patterns is usually large and the length of patterns is usually longer than the length of frequent patterns, the maximal FT-frequent is needed to discovery the general rules. We extend the step 1 in FTP mine to find the maximal FT-frequent patterns. For each transaction, we keep the MN and append the *s_item* one by one. For example, an itemset $\{a, b, c, d,$ e, $g, h\}$ is given and sorted by lexicographic order. We consider the first δ (assume δ=1) items in the itemset, and count MN in every transaction. If matching a, the item support of a is increased by 1. Further, we consider appending $b, c, d, e, g,$ and h subsequently. If the appended *s_item* will lead the *prefix* to be not FT-frequent, the *s_item* is discarded.

Our main idea is that an itemset is given and sorted by lexicographic order, and we want to find the maximal FT-frequent pattern under the lexicographic sequence. There are two phases to

test the itemset. The itemset to extend is called *prefix*, and the appending item is *s_item*. For example, the itemset $\{a, b, c\}$ extends to $\{a, b, c, d\}$ by appending d. We call {a, b, c} the *prefix*, and d the *s_item*. There ae two phase in the process to check the itemset as follows.

Phase 1(initial):

We compare the first δ items of the itemset to the transaction, count how many items matching (denoted MN), and increase the item support of items in ($prefix \cap t$).

Phase 2(pattern growth):

We continue to grow the patterns from length δ to length of itemset. To compare the remaining *s_item*s of itemset subsequently. When the previous MN is figured out, we can decide whether the *prefix* should continue to extend according to Lemma 2. If $|prefix|$-MN $\leq \delta$ and a transaction contains the *s_item*, then MN and item support of the *s_item* are increased by 1 respectively. If $|prefix|$-MN = δ and the transaction does not contain the *s_item*, the item support of previous matching item has to be decreased, and the transaction is not checked further. Because there are over δ faults, the transaction does not FT-contain the *prefix*. After the MN of each transaction is counted, we figure out the FT-support of ($prefix \cap s\_item$) by counting the number of transactions whose MN$\geq$ |prefix|+1-δ. If the item support of ($prefix \cup s\_item$) or FT-support does not achieve min_sup^{item} or min_sup^{FT}, each item support of *prefix* and the MN of each transaction have to recover to previous status. In addition, we use memory index to indicate the next start point for each transaction to check. The rules are listed as follows:

For each transaction
  If MN <|prefix|-δ then read next transaction
  If MN >|prefix|-δ
    If item s$\in$ t, increment the MN and item support
  If MN =|prefix|-δ
    If item s$\in$ t, increment the MN and item support
    If item s$\notin$ t, decrement the item support in (t $\cap$ *prefix*)

We describe the main idea by running Example 3.2. When we get the set of global frequent items, we collect all the global items to form the itemset to test. The itemset $\{a, b, c, d, e, g, h\}$ is determined to decide the maximal FT-frequent pattern when the items are appended subsequently. Initially, because δ faults are allowed, the *prefix* grows to first δ items. The MN of each transaction is counted with *prefix a*, such as step *1* in Fig. 3.8. Next, considering item *b*, the pattern growth phase can apply to count MN and item support of *prefix*. Besides, the memory index still needs to be adjusted to the item whose lexicographic order is greater than or equal to *s_item*. The FT-support is checked by accumulating the number of MN which is greater than or equal to $|prefix|$+1-δ, so FT-support of $\{a, b\}$ is 4. Take *c* into account. Because the item support and FT-support do not achieve the respective threshold, the item supports of *prefix* and MN have to recover to previous status respectively and *c* will not append to *prefix*. As the same

way, g and h will not be appended to grow the pattern. After all items are considered, the maximal FT-frequent pattern $\{a, b, d, e\}$ is determined.

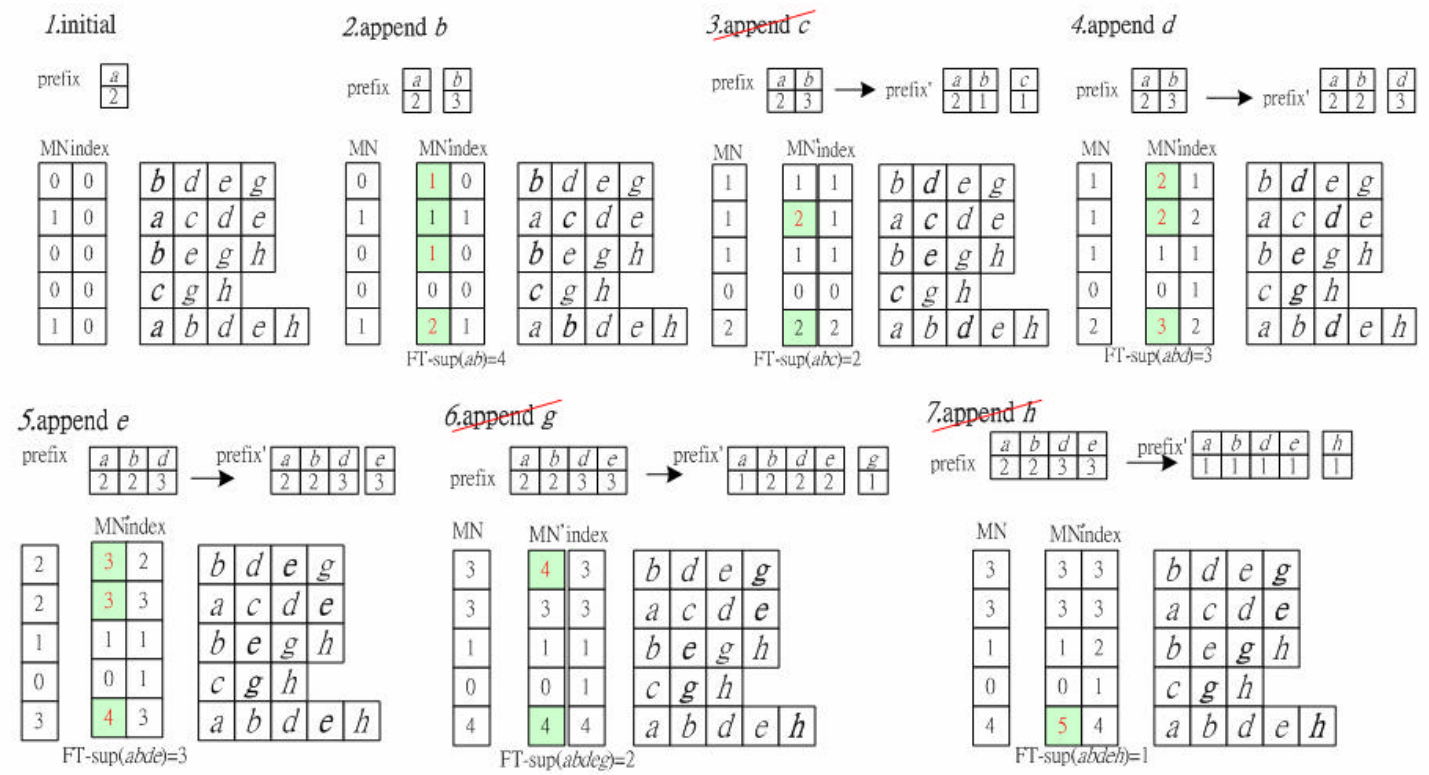


Fig 3.8 The process to find the maximal FT-frequent pattern

The next candidate is the itemset whose lexicographic order is next to *abde* and is a leaf of the lexicographic tree conceptually. Next issue is to generate this kind of candidate itemset? We just substitute the last item of the previous maximal FT-pattern by the frequent items whose lexicographic order is greater than the last one item. If the length of candidate itemset is less then δ+1 or the subset of previous maximal FT-patterns, we find the next candidate by deleting the last two items of previous maximal FT-pattern and by appending frequent items whose lexicographic order is greater than the last second item. By the same way, the candidate itemset can be found for testing. Then we need to confirm whether $\{a, b, d, e\}$ is a maximal FT-pattern. If there exists a maximal FT-frequent pattern which is a superset of {a, *b*, *d*, *e*}, then the path of that maximal pattern is in the left hand side of the path *a-b-d-e* in Fig. 3.9 or the extension of *abde-prefix*. It is impossible to find such patterns when we grow the pattern depending on the *prefix* and fault-tolerant Apriori property. Besides, any subset of the $\{a, b, d, e\}$ need not be checked again, due to $\{a, b, d, e\}$ is a maximal one. Therefore, the next candidate to be checked is $\{a, b, d, g, h\}$. However, the result of the candidates testing may be a subset of those maximal FT-frequent patterns which were found previously, and we have to drop the result. After all, the maximal FT-frequent patterns are generated: $\{a, b, d, e\}$, $\{a, c, h\}$, $\{a, e, g\}$, $\{b, c\}$, {*b*, d, *e*, *g*, *h*}, $\{c, d, g\}$, $\{c, e, g\}$.

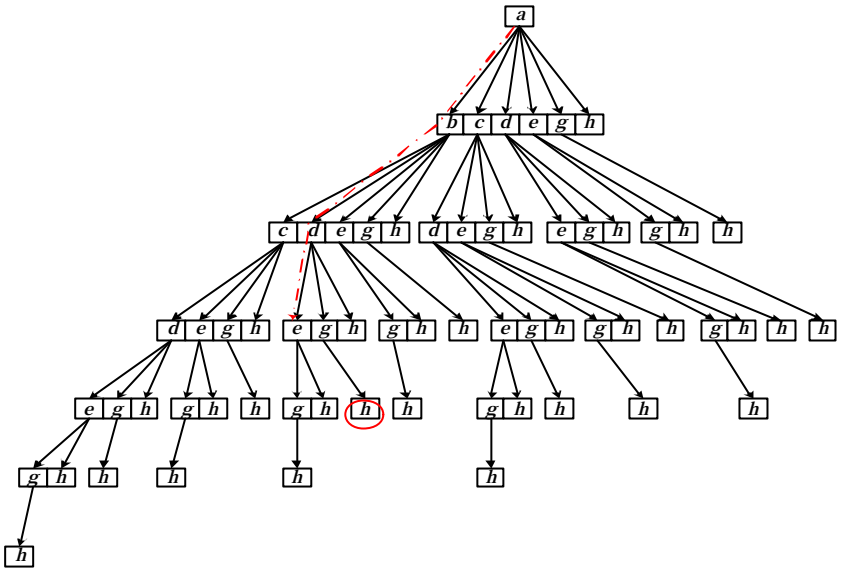


Fig 3.9 The lexicographic tree to find the next candidate

If the lexicographic tree is expressed by the increasing frequent order instead of alphabetic order, then the algorithm will be more efficient, because there are a lot of candidates pruned.

4. Experimental Result and Analysis

We implemented the FT-Apriori and FTP-mine algorithms using Microsoft C++ 6.0 .To evaluate impartially, we improve the FT-Apriori algorithm by loading the transaction database into memory. All the experiments were conducted on a PC with an Intel Pentium 4 1.6GHZ CPU and 256MB of RAM. Our data resource is from the synthetic dataset generator which is available in IBM web site [17] and the parameters shown in Table 5.1. In the following experiments, we aim at the influence of the parameters, min_sup^{item}, min_sup^{FT}, and fault tolerance δ, in the mining of fault tolerant frequent patterns respectively. Besides, we also assume the database is too large to load into memory to design an experimental by partition the database and observe the performance trend as the transaction database increasing.

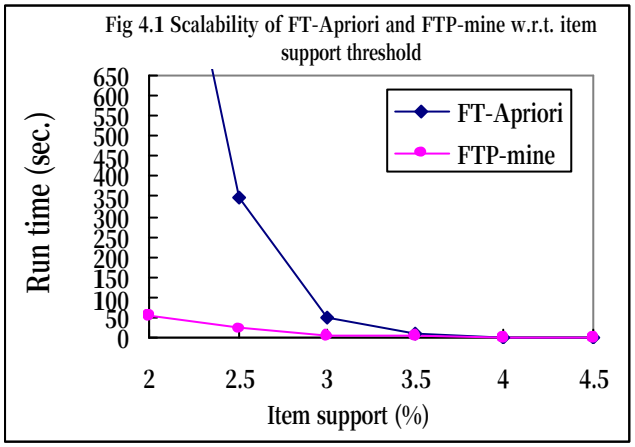
|D|	Number of transaction
|T|	Average size of the transactions
|I|	Average size of the maximal potentially large itemsets
N	Number of items

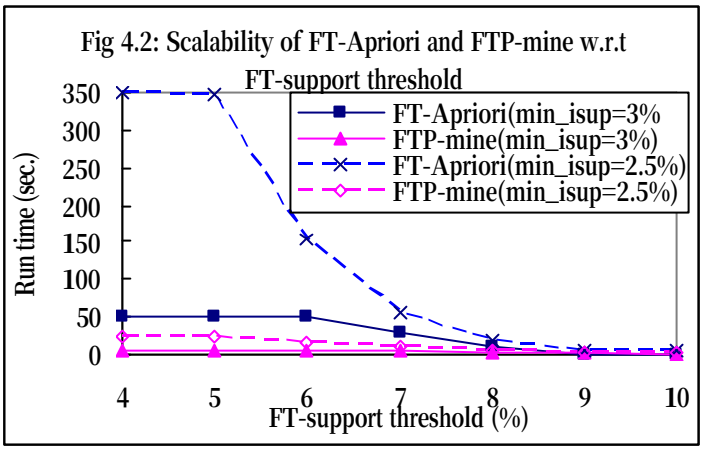
Table 5.1 Parameters

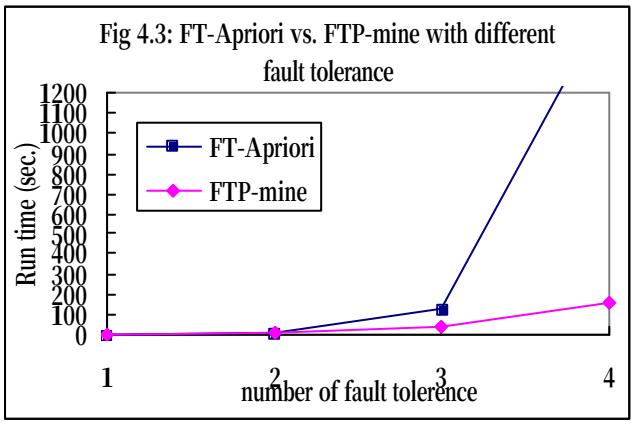
4.1 Experimental result

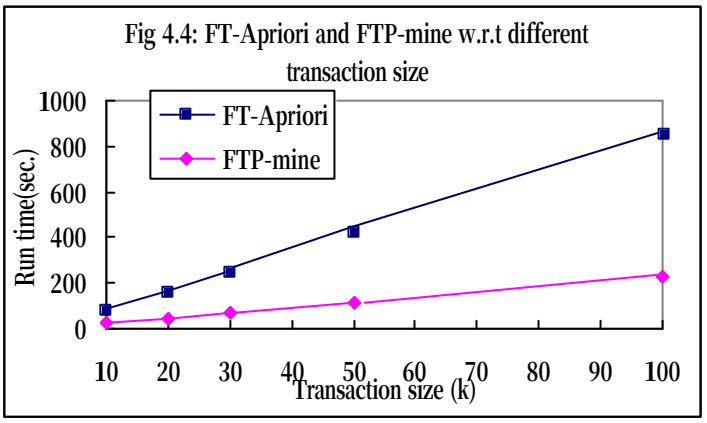
We use the dataset T10I8D10kN1k to observe the influence of the min_sup^{item} and min_sup^{FT}. We set the fault tolerance δ = 1, and min_sup^{FT} = 5%Fig 4.1 shows the run time of FT-Apriori and FTP-mine with respect to item support. As we can see from Fig 4.1, the run time increases as the frequent item support threshold goes down and FTP-mine outperforms FT-Apriori. As to FT-Apriori, when the item support is low, the number of patterns as well as candidates increase exponentially, and the cost would increase dramatically. On the contrary,

FTP-mine does not generate candidates by sub-patterns like FT-Apriori, so the curve is smooth and steady. The variation in run time over the FT-support threshold is shown in Fig 4.2. It seems that the run time increases as the FT-support decreases. FTP-mine also has better performance than FT-Apriori under all condition. Because the number of frequent items is determined by item support threshold, the min_sup^{item} is the critical factor to determine the size of candidates generated. The closer the FT-support is to item support, the less the candidates can be pruned, and the number of candidates to be FT-frequent patterns increases. Although the run time is increasing as the item support goes down, the increasing rate is slower. The FTP-mine algorithm is also more stable than FT-Apriori no matter what the item support is.



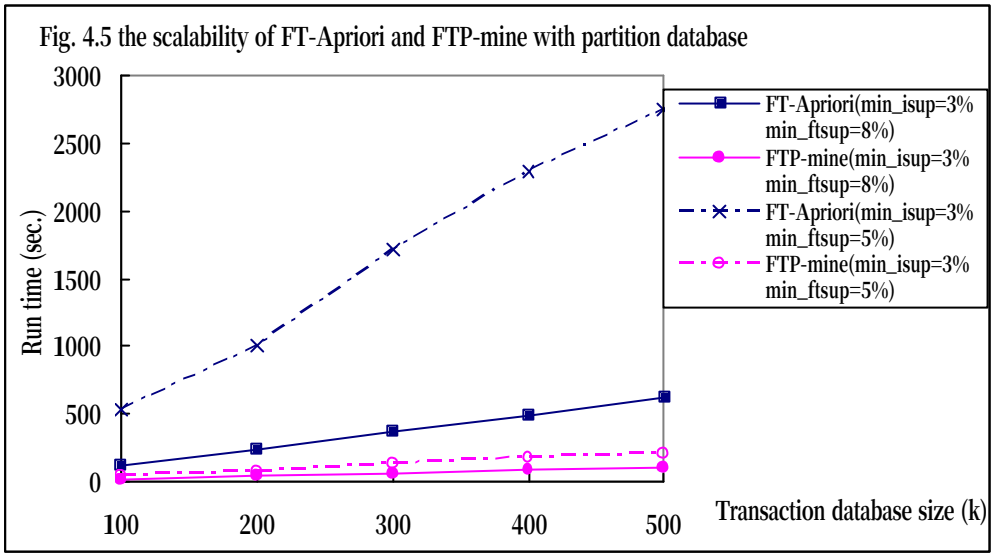






As to evaluate the influence of the parameter fault tolerance, we adopt another dataset T15I10D10kN1k. We set the min_sup^{item}=8% and min_sup^{FT}=10%. As the fault tolerance increases, the run time required is dramatically increasing especially in FT-Apriori. Because the length and the number of FT-frequent patterns increase as the fault tolerance increases, it seems inefficient for FT-Apriori to generate candidates and testing. In contrast to FTP-mine, which starts to traverse the level of length δ, FTP-mine will check the appended *s_item*s instead of generating candidates, so the trend does not increase dramatically as FT-Apriori.

We use another dataset T10I8D10kN1k to evaluate the impact of transaction size. Regarding to the parameters, we set min_sup^{item}=5%, min_sup^{FT}=7% and 1 mismatch allowed. Fig 4.4 presents that run time of FT-Apriori and FTP-mine with respect to different transaction database size. The performance is linearly scalable and FTP-mine always outperforms FT-Apriori. In order to observe the influence of the size of transaction database which can not be loaded into memory, we assume that the transaction database is divided into several subdatabases in which there are 50K transactions individually. In the experiment, FT-Apriori will mine FT-frequent patterns in disk file instead of in memory. The experimental result is shown in Fig 4.5.



4.2 Performance Analysis and Discussion

Obviously, FTP-mine outperforms FT-Apriori in all condition. Considering FT-Apriori, it takes much time in candidate generation and in testing whether a candidate is frequent to reduce the size of candidates pass by pass. On the other hand, FTP-mine checks the *s_item*s with the same *prefix* by comparing each transaction only once and according to the Lemma 2 to decide whether a transaction need to be compared further. FTP-mine just has to keep one table, STable, in memory and the space required is about O (L_1^2).

There is another approach to extend the FTP-mine to mine large databases. If the database is too large to load into memory, FTP-mine checks each transaction in disk file and project the transactions which FT-contains (*prefix* $\cup$ *s_item*) to sub-databases, For example, we intend to extend the *a-prefix* to {*a*, *b*} {*a*, *c*} {*a*, *d*} {*a*, *e*}{*a*, *g*} and {*a*, *h*}. After FTP-mine check transactions in disk, the FT-frequent patterns are determined. Next, we project a transaction to each sub-databases when the transaction FT-contains the FT-frequent pattern. So there is a projection database for each FT-frequent pattern. We go on to use FTP-mine to check transactions in disk until the projection database can fit in memory. Once the projection database can load into memory, the FTP-mine proposed in Section 3.1 is applied to mine FT-frequent pattern.

5. Conclusion and Future work

In this thesis, we have proposed a new mechanism, FTP-mine, to mine fault tolerant frequent patterns with depth first execution sequence and a special designed table, STable, which keep trace the FT-support, item support, and each item support of *prefix* for *s_item* and is space-saving. Besides, we also extend the algorithm to mine maximal FT-frequent patterns in memory and find the maximal FT-frequent patterns. Unlike FT-Apriori to generate candidates and test, FTP-mine adopts the nice feature of FP-growth and space-preserving to keep transactions in memory and determine the *s_item* with the same *prefix* by scanning each transaction only once. We conducted performance evaluation with respect to compare the efficiency of FTP-mine with FT-Apriori. The result showed that FTP-mine outperforms FT-Apriori in various settings and is linear scalable. Some problems are worth further investigation in the future.

- **Mining maximal FT-frequent patterns in large database.** The approach we proposed to mine maximal FT-frequent patterns is considering the case that all transactions can be loaded in the memory. However, if the approach extends to a large database, there are some difficulties. Because local maximal FT-frequent patterns may not be global maximal FT-frequent patterns, that does not mean their sub-patterns are not the global maximal FT-frequent patterns. It is inefficient to determine which sub-pattern is the global maximal FT-patterns.
- **Fault-tolerant sequential pattern mining.** It is difficult to formulate the problem, because the fault allowed may be in inter transactions or intra transactions for sequential pattern miming. How to define formally is a challenge, and the mining process may require large mount of CPU time. So the performance is also the factor need to be taken into account in the mining of fault-tolerant sequential patterns.

Reference

[1] J. Pei, A. K. H. Tung, and J. Han, "Fault-Tolerant Frequent Pattern Mining: Problems and Challenges ", *Proceedings of ACM-SIGMOD International Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'01)*, Santa Barbara, CA, May 2001,

[2] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases", *Proceedings of International Conference on Data Mining (ICDM'01)}*, San Jose, CA, Nov. 2001, pp. 441-448.

[3] . Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth", *Proceedings of International Conference on Data Engineering (ICDE'01)*, Heidelberg, Germany, April 2001, pp. 215-226.

[4] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *5th International Conference on Extending Databases Technology*, 1996, pp. 3-17.

[5] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation ", *Proceedings of ACM-SIGMOD International Conference on Management of Data (SIGMOD'00*), Dallas, TX, May 2000, pp. 1-12.

[6] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proceedings of the 20th VLDB Conference Santiago*, Chile, 1994, pp. 487-499.

[7] H. Mannila, H. Toivonen, and A. I. Verkamo, "Efficient Algorithms for Discovering Association Rules", *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, Seattle, Washington, July 1994, pp.181-192.

[8] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", *Proceedings of ACM SIGMOD*, 1998, pp. 85-93.

[9] J. Pei, J. Han, and R. Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets", *Proceedings of ACM-SIGMOD International Workshop on Data Mining and Knowledge Discovery (DMKD'00)*, Dallas, TX, May 2000, pp. 11-20.

[10] Mohammed J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences", *Journal of Machine Learning, special issue on Unsupervised Learning*, Vol. 42 Nos. 1/2, Jan/Feb 2001, pp 31-60.

[11] Karam Gouda, Mohammed J. Zaki, "Efficiently Mining Maximal Frequent Itemsets", *Proceedings of IEEE International Conference on Data Mining (ICDM' 01)*, San Jose, November 2001, pp. 163-170.

[12] M. J. Zaki, "Generating Non-Redundant Association Rules", *Proceedings of 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, MA, August 2000, pp 34-43.

[13] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme and L. Lakhal, "Mining Frequent Patterns with Counting Inference", *Journal of SIGKDD Explorations*, Vol. 2, No. 2, Dec. 2000, pp. 66-75.

[14] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, "Depth First Generation of Long Patterns", Proceedings of International Conference on Knowledge Discovery & Data Mining, Boston, USA, Auguest 2000, pp. 108-118.

[15] W. Wang, J. Yang, and P. S. Yu, "Mining Patterns in Long Sequential Data with Noise", *Journal of SIGKDD Explorations*, Vol. 2, No. 2, Dec. 2000, pp. 28-33.

[16] M. S. Chen, Jiawei Han and Philip S. Yu, "Data mining: an overview from a database perspective", *Journal of IEEE Transaction on Knowledge And Data Engineering*, Vol. 8, Dec. 1996, pp. 866-883.

[17] IBM inc. http://www.almaden.ibm.com/cs/quest/syndata.html.