Max-FTP: Mining Maximal Fault-Tolerant Frequent Patterns from Databases

Shariq Bashir and A. Rauf Baig

National University of Computer and Emerging Sciences, Islamabad, Pakistan
shariq.bashir@nu.edu.pk, rauf.baig@nu.edu.pk

Abstract. Mining Fault-Tolerant (FT) Frequent Patterns in real world (dirty) databases is considered to be a fruitful direction for future data mining research. In last couple of years a number of different algorithms have been proposed on the basis of Apriori-FT frequent pattern mining concept. The main limitation of these existing FT frequent pattern mining algorithms is that, they try to find all FT frequent patterns without considering only useful long (maximal) patterns. This not only increases the processing time of mining process but also generates too many redundant short FT frequent patterns that are un-useful. In this paper we present a novel concept of mining only maximal (long) useful FT frequent patterns. For mining such patterns algorithm we introduce a novel depth first search algorithm **Max-FTP** (Maximal Fault-Tolerant Frequent Pattern Mining), with its various search space pruning and fast frequency counting techniques. Our different extensive experimental result on benchmark datasets show that Max-FTP is very efficient in filtering un-interesting FT patterns and execution as compared to Apriori-FT.

Keywords: Fault Tolerant Frequent Patterns Mining, Maximal Frequent Patterns Mining, Bit-vector Representation, and Association Rules.

1 Introduction

Mining frequent patterns from transactional or relational datasets with support greater than a certain user defined threshold, plays an important role in many data mining applications such as intrusion detection, finding gene expression patterns, web log patterns etc. In recent years, a number of algorithms have been proposed for efficient mining of such frequent patterns, on the basis of Apriori property proposed by Agrawal et al. [1]. These algorithms take a transactional dataset and support threshold *(min_sup)* as an input and output those exact matching frequent patterns which contain support greater than *min_sup*, with assuming that the dataset is very well pre-processed and noise free. However, the real world datasets are dirty and contain missing and noisy values. In such situations, users face difficulties in setting this *min-sup* threshold to obtain their desired results. If *min-sup* is set too large, then there may be a small number of frequent patterns, which does not give any desirable result. If the *min-sup* is set too small, then there may be many redundant short un-useful frequent patterns, which not only take a large processing time for mining but also increase the

R. Cooper and J. Kennedy (Eds.): BNCOD 2007, LNCS 4587, pp. 235–246, 2007.

complexity of filtering un-interesting frequent patterns. In both situations, the ultimate goal of mining interesting frequent patterns is undermined.

For handling such situations, J. Pei at el. in [6] introduced a new application of finding only interesting frequent patterns in a real world dirty datasets, instead of finding exact patterns. This approach is known as fault-tolerant (FT) frequent pattern mining. The problem of mining all FT frequent patterns from a dirty transactional dataset can be considered from the following two conditions [6].

1. Under user defined fault tolerance factor δ, a pattern X with cardinality greater than δ is called a FT frequent pattern, if it appears in at least k number of FT-transactions. A transaction t is called a FT-transaction under fault tolerance factor δ, if it contain at least $|X|$-δ number of items of X. The number k is called the frequency of X which must be greater or equal than the minimum FT support threshold (min_sup^{FT}).
2. Each individual single item i of X must be appeared in at least l number of FT-transaction of X, where l is called the minimum item support threshold under fault tolerance factor δ ($\text{item\_sup}^{FT}{}_{\delta}$).

For example, with $\text{min\_sup}^{FT} = 3$ and $\text{item\_sup}^{FT}{}_{\delta} = 2$, the pattern $\langle A, B, C, D \rangle$ is a FT frequent pattern under fault tolerance factor $\delta = 1$, since 3 out of 4 items are present in FT-transaction T1, T3 and T5 which qualifies min_sup^{FT} threshold and each single item A, B, C and D is present in at least 2 transactions with qualifies $\text{item\_sup}^{FT}{}_{\delta}$ threshold. In [6] they also proposed an Apriori-FT algorithm for finding all type of such patterns. The Apriori-FT was extended from the Apriori approach, in which downward closure property is used for mining FT frequent patterns. Similar to Apriori algorithm, Apriori-FT applies a bottom-up search that enumerates every single FT frequent pattern. This implies that in order to produce a FT frequent pattern of length l, it must produce all 2^l of its subsets, since they too must be frequent FT. This exponential complexity fundamentally restricts Apriori-FT like algorithms in discovering only useful interesting FT frequent patterns in a reasonable time limit. Moreover, mining FT frequent patterns are very complex than mining all frequent patterns, in terms of both search space exploration and frequency counting of candidate patterns. In frequent pattern mining, a candidate pattern X is declared to be frequent, by checking its frequency in only one dataset scan. While in FT frequent pattern mining, a number of dataset scans are needed to declare a candidate FT pattern X as frequent, which depends on the cardinality of pattern X. In addition to frequency counting, most of the search space pruning techniques, such as parent equivalence pruning (PEP) and 2-Itemset Pair of frequent pattern mining can not be applied on mining FT frequent patterns for filtering infrequent FT patterns.

To overcome these limitations, in this paper we have introduced a novel maximal or long FT frequent pattern mining (MFP^{FT}) concept. Similar to maximal frequent pattern mining [3], a pattern X is called a maximal FT frequent pattern, if it has no superset that is also a maximal frequent FT pattern. Mining only MFP^{FT}s has many advantages over mining all FT frequent patterns. Firstly, long patterns are very useful in some very important data mining applications such as biological data from the field of DNA and protein analysis and clustering. Secondly, different search space pruning techniques such as FHUT and HUTMFI (Section 5) can be also applied easily on

mining MFP^{FT}s algorithm, which dynamically prune the irrelevant search space during mining process. Thirdly, we know MFP^{FT}s are supersets of all FT frequent patterns; therefore Max-FTP output implicitly and concisely represents all FT frequent patterns. Finally, a single dataset scan can collect all the FT frequent patterns, once we have MFP^{FT}s.

2 Problem Definition

To consider the problem of mining MFP^{FT}s, let us take a sample transactional dataset of Table 1. It consists of 8 transactions with 12 different items. Let us take the $min\_sup^{FT} = 3$ and $item\_sup^{FT}_{\delta} = 2$ with fault tolerance factor $\delta = 1$. Since some of the items contain frequency less than $item\_sup^{FT}_{\delta} = 2$. Therefore, according to subset-superset property of Apriori-FT [6], these items can be safely removed from the dataset transactions and list of single frequent FT patterns, before starting the actual FT mining algorithm. Column 3 of the Table 1 dataset shows the modified representation of actual dataset transactions, which consist of only those items which have frequency greater than $item\_sup^{FT}_{\delta}$.

The search space of FT frequent pattern mining can be considered as a lexicographical order [7], where root node contains an empty pattern, and each lower level k contains all the k-patterns. Figure 1 shows the search space of Table 1 dataset, where

Table 1. A sample transactional dataset with 8 transactions and 12 items

Transaction ID	Items	Frequent Items
T1	A, B, C, I	A, B, C
T2	B, C, E, J, K	B, C, E
T3	A, C, D, L	A, C, D
T4	A, C, E, G	A, C, E
T5	A, B, C	A, B, C
T6	D, E	D, E
T7	D, E	D, E
T8	D, E	D, E

each node is composed of head and tail elements. Head denotes the FT frequent pattern of that node, and tail consists of those items which are possible extensions of new candidate FT patterns. For example node P in Figure 1 contains a head *⟨A, B, C⟩* and tail *⟨D, E⟩*, which generates 2 child nodes or candidate FT patterns: node *⟨A, B, C, D⟩ and node ⟨A, B, C, E⟩*. This FT frequent pattern mining search space can be traversed using either depth-first-search (DFS) or breadth-first-search (BFS) approach.

The problem of mining FT frequent patterns can be considered as a finding of cut in the search space, where the patterns above the cut are frequent FT and patterns below the cut denote infrequent FT. While the problem of mining MFP^{FT}s is to mine only those frequent FT patterns above the cut, which are not subset of any other frequent FT pattern. For example the node S1 with head *⟨A, B, C, D⟩* , node S2 with head *⟨A, B, C, E⟩* and node S3 with head *⟨D, E⟩* in Figure 1 are those patterns which are not subset of any other frequent FT pattern.

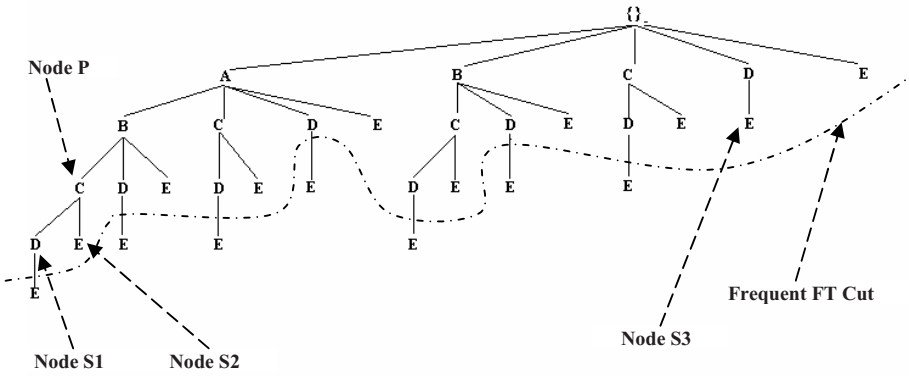

Fig. 1. The FT Search Space of Table 1 dataset with $min\_sup^{FT} = 3$ and $item\_sup^{FT}{}_{\delta} = 2$ under fault tolerance factor $\delta = 1$

To decrease the processing time of mining useful FT frequent patterns, different search space pruning techniques can also be applied during MFP^{FT}s mining on the basis of known MFP^{FT}s list. The known MFP^{FT}s list at any node n consists of only those maximal FT patterns which are discovered priori the traversal of n. For example in Table 1 with $min\_sup^{FT} = 3$, $item\_sup^{FT}{}_{\delta} = 2$ and fault tolerance factor $\delta = 2$. Once a pattern *⟨A, B, C, D, E⟩* is known to be MFP^{FT}, then using this information the search space consisting of sub trees *⟨A, B, C, E⟩*, *⟨A, B, D⟩*, *⟨A, B, E⟩*, *⟨A, E⟩*, *⟨A, D⟩*, *⟨A, E⟩*, *⟨B⟩*, *⟨C⟩*, *⟨D⟩* and *⟨E⟩* can be safely pruned away. This is because they are all subsets of known MFP^{FT} *⟨A, B, C, D, E⟩*. Where, a pure Apriori-FT like algorithm will have to done extra work and will traverse and generate these many short redundant FT frequent patterns.

3 Related Work

Mining FT frequent patterns in a real world dirty datasets has remained as a central core of attention in last couple of years. The basic concept behind FT frequent pattern mining is to discover more general and interesting patterns instead of finding exact frequent patterns. J. Pei et al. in [6] proposed an Apriori like FT pattern mining algorithm. Apriori-FT uses a complete, bottom up search, with a horizontal layout and prune away infrequent FT patterns using anti-monotone Apriori-FT property heuristic: if any length k FT pattern is not frequent, then all of its length $(k+1)$ supersets will be also infrequent. The major weakness of Apriori is its difficulty in mining long FT patterns. For example, to find a frequent FT pattern of $X = \{1,\ldots, 200\}$ items. Apriori-FT has to generate-and-test all candidate 2^{200} FT patterns.

J. L. Koh et al. in [4] proposed their VB-FT-Mine FT patterns mining algorithm. In their approach, FT appearing vector are designed to represent the distribution of the candidate FT patterns under fault tolerance factor δ using bit-vector representation approach. VB-FT-Mine algorithm applies DFS traversal to generate candidate FT patterns. The major feature of VB-FT-Mine is its fast frequency counting operation.

The algorithm decides quickly that, whether a candidate FT pattern is frequent or infrequent by performing bitwise operations on appearing vectors.

4 Max-FTP: Algorithmic Description

In this section, we describe our maximal frequent FT pattern mining algorithm Max-FTP with its various techniques used for search space pruning and performance improvements. For FT pattern representation at any node, in Max-FTP we use the vertical bit-vector representation approach. The major advantage of using vertical bit-vector approach over any other pattern representation approach is that, it is more efficient in pattern frequency counting. Since the frequency of any candidate FT pattern can be obtained by performing the bitwise-AND operations on head and tail bit-vectors. Therefore, on a processor which supports 32-bits per AND operation, there 32 transactions are checked for support (frequency) counting in only one bitwise-AND operation. This optimizes the candidate FT pattern frequency counting cost with a factor of 1/32. Firstly, we describe a simple DFS traversal algorithm with no pruning. Then, we use this algorithm to motivate the pruning and dynamic reordering of tail items in section 5.

Max_FTP_SimpleDFS (Node n)

```
1. for each item X in n.tail
2.     m.tail = n.tail ∪ X
3.     m.tail = n.tail – X
4.     n.tail = n.tail – X
5.     if ( m.head is frequent )
6.             Max_FTP_SimpleDFS (m)
7. if ( n is a leaf and n.head in not in list (known MFP^FTs))
8.     Add n.head in (known MFP^FTs)
```

Fig. 2. Pseudo code of Max-FTP with simple DFS traversal

4.1 Max-FTP with Simple DFS Traversal

In a simple DFS traversal we generate the candidate FT patterns by following simple lexicographical order [7]. At any node n, the candidate FT patterns or child nodes of n are generated by performing join operation on *n's* head pattern with each item of *n's* tail, and checked for frequency counting. If the frequency of candidate FT pattern *{n's head ∪ n's tail item}* is less than min_sup^{FT} or the frequency of any single item of pattern *{n's head ∪ n's tail item}* is less than $\text{item\_sup}^{FT}_{\delta}$. Then we can stop by the Apriori-FT principle, since the superset of any infrequent FT pattern will be also infrequent FT. If all of the candidate FT patterns of any node are found to be infrequent, then that node is considered to be a candidate maximal FT frequent pattern. To check whether the current candidate maximal FT pattern X is not subset of any known MFP^{FT}, we must check its maximality in known MFP^{FT}s list. If none of the known MFP^{FT} is superset of this candidate pattern X, then X is considered to be a valid

MFP^{FT} pattern and added in the known MFP^{FT}s list. The set of all patterns in known MFP^{FT}s list denotes the required long or maximal frequent FT patterns. Figure 2 shows the pseudo code of Max-FTP with simple DFS traversal.

4.2 Dataset Representation and Fast Frequency Counting

As introduced earlier, in Max-FTP we use the vertical bit-vectors for FT pattern representation at any node. In contract to tradition bit-vectors representation used in [2], Max-FTP associates more than one bit-vectors at each node for FT pattern representation, depending on the size of fault tolerance factor δ. The first bit-vector at any node n represents FT transactions containing pattern P of node n under fault tolerance factor δ, which is called as bitmap(n). This bit-vector representation is almost similar to pattern representation that was used in [4]. In bitmap(n) there is one bit for each transaction of the dataset. If the pattern P of node n does not contain a FT transaction at position j under fault tolerance factor δ. Then the y bit of bitmap(n) is set to 0, otherwise to 1.

In addition to the representation of FT transactions containing pattern P of any node n under fault tolerance factor δ, we also have to count the number of items of pattern P that are not present at any transaction position j. For example in Figure 1, the pattern $\langle A, B, C, E\rangle$ at node S1 does not contain item $\langle E\rangle$ at transaction position 1 and 5, so the count of missing items must be 1 at these position. In Max-FTP we use the FT bit-vectors FT-$bitmap_1$(n), FT-$bitmap_2$(n), …., FT-$bitmap_\delta$(n) for this purpose. If at any node n, the pattern P does not contain its $k \leq \delta$ items at transaction position j. Then the bits at position j of exactly k FT bit-vector (FT-$bitmap_1$(n),…., FT-$bitmap_k$(n)) are set to 1, which indicates that exactly the k items of pattern P are not present at this position.

Candidate FT Pattern Frequency Counting. Before considering, how Max-FTP count the frequency of any candidate FT pattern at any node n by performing the bitwise operations on *n'head* and *n'tail* item bit-vectors, let us consider the following definition.

Definition 1. Let $n' = n \cup \{x\}$ denotes a candidate FT pattern at node n, where x is the tail item and n is the frequent FT pattern. A transaction T contains FT under fault tolerance factor δ with pattern n', if at least one of the following two properties holds.

1. T contains FT under fault tolerance factor $(\delta\text{-}1)$ with pattern n, or
2. T contains x and FT under fault tolerant factor (δ) with pattern n'.

Candidate FT pattern frequency counting using bit-vectors. The property 1 of definition 1 can be obtained by performing bitwise-AND operations on bit-vectors bitmap(n) with bitmap(x). While the property 2 of definition 1 can be obtained by performing bitwise-OR operations on bit-vectors bitmap($n \cup x$) with FT-$bitmap_1$(n),…,FT-$bitmap_\delta$(n). The number of ones in the resulting bit-vector, after performing the bitwise-AND and bitwise-OR operations of property 1 and property 2 of definition 1 denotes the frequency of candidate FT pattern $n' = n \cup \{x\}$.

Single Item support checking. The procedure that we describe above calculates only the frequency of FT transactions contain pattern *n'* under fault tolerance factor δ. If this frequency is greater than min_sup^{FT}, then the frequency of each individual item *i* of pattern *n'* can be calculated by performing bitwise-AND operations on bit-vectors bitmap(*n'*) with bitmap(*i*). While, the number of ones in the resulting bit-vector denotes the frequency of item *i* in pattern *n'*, which should be greater than $\text{item\_sup}^{FT}_{\delta}$, otherwise the pattern is declared as infrequent FT pattern.

5 Improving Max-FTP Performance

The depth-first-search traversal algorithm that we present in section 4.1 is ultimately no better than the Apriori-FT algorithm. Since exactly the same number of candidate FT patterns are generated and checked for frequency counting. Therefore, for gaining performance, in Max-FTP we used two search space pruning techniques name as FHUT (frequent head union tail) and HUTMFP^{FT} (head union tail maximal frequent FT pattern). In addition to the two search space pruning techniques, Max-FTP also reorders the tail items using dynamic reordering heuristic. This heuristic was first used in [3], which keeps the search space as small as possible.

FHUT (Frequent Head Union Tail). In [3], it was observed that the longest pattern that can be mined using DFS traversal is the *head* $\cup$ *{all items of tail}*. If at any node, it has been known that the entire *head* $\cup$ *{all items of tail}* or left most sub tree is found to be frequent. Then a substantial amount of performance can be obtained by pruning rest of the sub tree of *head* $\cup$ *{all items of tail}*. In Max-FTP we track this information at each node by using a boolean flag, which determines whether the left most item of tail is frequent or infrequent FT. If all of the left most childs of any node *n's* subtee is found to be frequent FT, or in other words the left most path of the node *n's* subtee is found frequent, then the pruning can be obtained under the concept of FHUT.

HUTMFP^{FT} (Head union Tail Maximal Frequent FT Pattern). Before checking the frequencies of any candidate FT patterns at node *n*, if it is found that the entire *head* $\cup$ *{all items of tail}* is already a subset of any known MFP^{FT}. Then by following the Apriori-FT property [6], we can safely prune away the entire sub tree of *head* $\cup$ *{all items of tail}*. In Max-FTP we found that a substantial amount of performance can be obtained by using this HUTMFP^{FT} pruning technique.

Dynamic Reordering of Tail Items. If any candidate FT pattern *y* at any node *n* is found to be frequent, then according to the Apriori-FT property all the supersets of *y* will be also infrequent FT and can be safely pruned away. For achieving this look ahead pruning technique, in Max-FTP we calculate the frequency of all the tail items of node *n*, before traversing any of them. Then by using this tail items frequencies information, Max-FTP reorders the node *n* tail items and removes all those which are found infrequent FT. Obviously, in one side this creates an additional burden of counting frequencies of node n's tail items by two times. First, at the time of reordering tail items and second, at the time of creating bit vectors of bitmap(*n* $\cup$ *tail item*)

```
Max_FTP_withPruning (Node n, FHUT_Flag)

1.  HUT = n.head ∪ n.tail
2.  if ( HUT is in list (known (known MFP^FTs))
3.      stop searching and return

4.  remove infrequent items from n.tail and recorder them by increasing frequency.
5.  for each item X in n.tail
6.      m.tail = n.tail ∪ X
7.      m.tail = n.tail – X
8.      n.tail = n.tail – X

9.      FHUT_Flag = true if X is the left most item of n.tail
10.     if ( m.head is frequent )
11.             Max_FTP_SimpleDFS (m)

12. if ( n is a leaf and n.head in not in list (known MFP^FTs))
13.     Add n.head in (known MFP^FTs)
14. if ( HUT_Flag and all extensions are frequent )
15.     stop exploring this subtree and go back up tree to where FHUT_Flag was
        changed to True.
```

Fig. 3. Pseudo code of Max-FTP with search space pruning techniques

and FT-bitmap$_1$($n \cup$ *tail item*), …,FT-bitmap$_\delta$($n \cup$ *tail item*). However, on the other side most of the infrequent FT nodes are pruned away using this look ahead pruning technique.

In addition to the removing of infrequent items from node *n*'s tail, Max-FTP also arranged the frequent tail items by increasing support. This heuristic was first used by Bayardo in [3], which keeps the search space as small as possible. Figure 3 shows the pseudo of Max-FTP with embedded search space pruning techniques and dynamic reordering heuristic.

6 Candidate Maximal FT Pattern Maximality Checking

In our opinion efficient mining of MFP^{FT}s depend upon two main factors. First, at the search space traversal approach and using different search space pruning techniques. Second, at the maximal candidate FT pattern maximality (superset) checking approach which takes O(known MFP^{FT}s) in worst case. Let list (known MFP^{FT}s) be the currently known maximal FT frequent patterns. To check whether pattern *P* is subset of any known MFP^{FT} pattern. We must perform a maximal superset checking, which takes O(MFP^{FT}s) in worst case. In Max-FTP for speeding up the cost of superset checking, we used the local maximal frequent pattern (LMFI) superset checking approach with the name (LMFP^{FT}) [5]. LMFP^{FT} is a divide and conquer strategy which contains only those maximal FT frequent patterns at any node *n*, in which *n*'s head appears as a prefix.

Definition 1. Any MFP^{FT} pattern P can be a superset of $P \cup_{subsets(P)}$ *or* $P \cup_{freq\_ext(P)}$. The set of $P \cup_{freq\_ext(P)}$ is called the $LMFP^{FT}$ with respect to P, denoted as $LMFP^{FT}_p$. To check whether P is a subset of any existing MFP^{FT}s, we can checked them in only $LMFP^{FT}_p$, which takes $O(LMFP^{FT}_p)$ cost which is smaller than $O(MFP^{FT}s)$. If $LMFP^{FT}_p$ is empty, then P will be our new maximal FT pattern, otherwise it is the subset of $LMFP^{FT}_p$.

6.1 Child ($LMFP^{FT}$) Construction and Representation

Max-FTP used vertical bitmap representation approach for $LMFP^{FT}$ representation. In a vertical bitmap, there is one bit for each item of known maximal frequent FT pattern. This is the same concept as we have described for candidate FT pattern representation in section 4.2. If the pattern P at node n is the subset of any existing known maximal FT pattern at position j, then the bit j of bitmap($LMFP^{FT}_p$) is set to one; otherwise zero.

The $LMFP^{FT}$ of tail item X at node n with pattern P *(n's head)* can be constructed simply by taking bitwise-AND of bitmap($LMFP^{FT}_p$) with bitmap(X in the known maximal frequent FT pattern vertical bitmap representation).

bitmap($LMFP^{FT}_{(P \cup X)}$) = **bitmap**($LMFP^{FT}_p$) bitwise-AND **bitmap**(X in the known maximal frequent FT vertical bitmap representation).

7 Experiments

We performed 3 experiments to evaluate the performance of Max-FTP. In first experiment we compare the number of FT frequent patterns mine from the two different algorithms Apriori-FT and Max-FTP. The second experiment compares the running time of Max-FTP and Apriori-FTP. Finally, the effectiveness of different components of Max-FTP is evaluated in third experiment. All the source code of Max-FTP and Apriori-FT is written in C language. The experiments are performed on 3.2 GHz processor with main memory of size 512 MB, running windows XP 2005 professional. For experiments, we used the benchmark datasets available at http:// fimi.cs.helsinki.fi/data/. These datasets are frequently used in many frequent pattern mining algorithms. Unfortunately, due to lack of space we could not show our experiments that we have performed on all the available datasets, therefore, we select the three best datasets from different sparse, dense and large categories. Table 2 shows the description of our experimental datasets.

Table 2. Datasets use in our experimental results

Dataset	Items	Average Length	Records
BMS-POS	1658	7.5	515,597
Mushroom	119	23	8,124
Kosarak	20,753	8.1	900,000

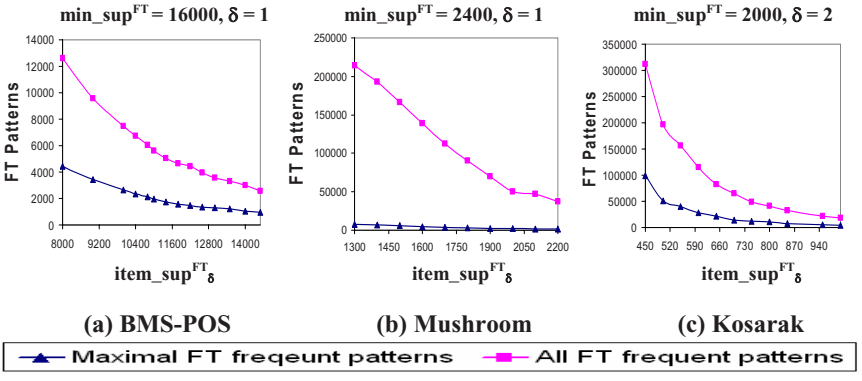

Fig. 4. Number of FT patterns mines using Max-FTP and Apriori-FT algorithms

Experiment 1. Figure 4 shows the number of FT frequent patterns mined by the two algorithms Apriori-FT and Max-FTP with different min_sup^{FT}, item_sup$^{FT}_{\delta}$ and δ thresholds. As clear from Figure 4, when the values of these thresholds decreases, the gap between number of useful FT frequent patterns mined with maximal and all FT frequent pattern mining algorithms becomes wider. In our experiments we found that as the average transactional length of dataset increases Max-FTP performs more well behaved as compared to Apriori-FT, since it mines only long pattern instead of small redundant patterns.

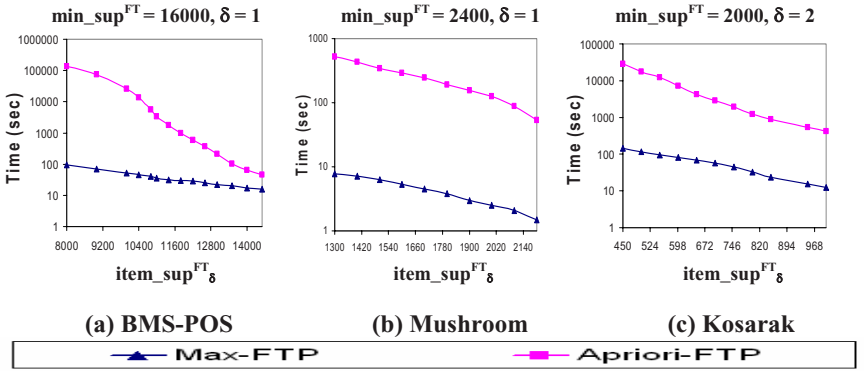

Fig. 5. Exectuion Time of Max-FTP and Apriori-FT on different FT thresholds levels

Experiment 2. In this experiment, we evaluate the performance of execution time of both algorithms with different min_sup^{FT}, item_sup$^{FT}_{\delta}$ and δ thresholds. As clear from Figure 5, the Max-FTP outperforms the Apriori-FT algorithm on almost all threshold levels, due to its effective search space pruning and frequency counting techniques. The Apriori-FT lacks of these effective search space pruning techniques; therefore, a much large set of candidate FT patterns are generated in order to build the complete set of frequent FT patterns. In experiments we found that on sparse datasets allowing

more noise (δ) and smaller item_sup$^{FT}_{\delta}$ threshold in Apriori-FT results millions of frequent FT patterns; but consequently, more candidate FT patterns are explored, and computational cost increases exponentially with respect to the dimensionality of the patterns. While, Max-FTP with a very high rate of δ and small threshold values of min_sup^{FT} and item_sup$^{FT}_{\delta}$ generates only useful interesting maximal frequent FT pattern in a very small amount of processing cost.

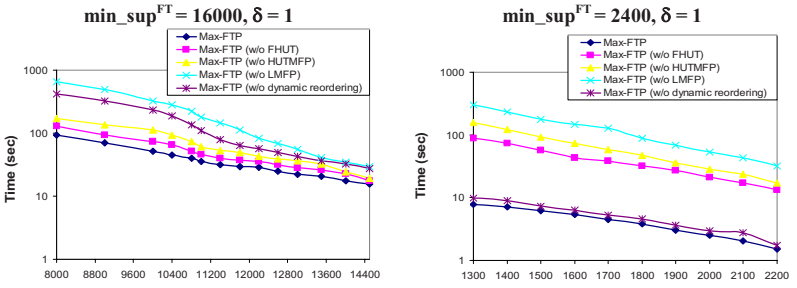

Fig. 6. Performance analysis of Max-FTP components on BMS-POS and Mushroom datasets

Experiment 3. In this experiment, we present a full analysis of each component of Max-FTP algorithm. There are three types of search space pruning techniques used to trim the tree: FHUT, HUTMFP^{FT} and dynamic reordering (of tail items). Moreover, LMFP^{FT} is used for fast maximal superset checking. Figure 6 shows the effects of each component of Max-FTP on Mushroom (dense) and BMS-POS (sparse) datasets. As clear from results, each component yields some saving in running time, but LMFP^{FT} is more effective than other techniques. The effect of FHUT and HUTMFP^{FT} is more encouraging on dense dataset (Mushroom) than sparse (BMS-POS) dataset; due to small number of items and larger average transactional length. The advantage of dynamic reordering is more effective on sparse dataset (BMS-POS) than dense dataset (Mushroom). Since most of the infrequent tail items are pruned earlier at lower level nodes of search space.

8 Conclusion

Mining fault tolerant frequent patterns in a real world dirty datasets is considered to be a fruitful direction for future data mining research. Previous FT frequent patterns mining algorithms such as Apriori-FT, generates not only long (interesting) FT patterns but also their many redundant short subset (un-useful) patterns. This not only increases the processing time of pattern mining process but also increases the size of rules. In this paper, we introduce a new concept of mining only maximal (long) frequent FT patterns. For mining such long patterns, we present a maximal FT pattern mining algorithm called Max-FTP with its various search space pruning techniques and fast frequency counting of candidate maximal FT pattern. Our different results on

benchmark datasets show that Max-FTP mines not only a small number of interesting frequent FT patterns as compared to Apriori-FT, but it is also efficient in term of execution due to limited number of candidate FT patterns generation.

References

[1] Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proc. Int'l Conf. Very Large Data Bases, pp. 487–499 (September 1994)
[2] Burdick, D., Calimlim, M., Gehrke, J.: Mafia: A maximal frequent itemset algorithm for transactional databases. In: Proc. of ICDE Conf, pp. 443–452 (2001)
[3] Bayardo, R.J.: Efficiently mining long patterns from databases. SIGMOD, 85–93 (1998)
[4] Koh, J.L., Yo, P.: An Efficient Approach for Mining Fault-Tolerant Frequent Patterns based on Bit Vector Representations. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 17–20. Springer, Heidelberg (2005)
[5] Gouda, K., Zaki, M.J.: Efficiently mining maximal frequent itemsets. In: ICDM, pp. 163–170 (2001)
[6] Pei, J., Tung, A.K.H., Han, J.: Fault-Tolerant Frequent Pattern Mining: Problems and Challenges. In: The proceedings of ACM-SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'01) (2001)
[7] Rymon, R.: Search through Systematic Set Enumeration. In: Proc. Of Third Int'l Conf. On Principles of Knowledge Representation and Reasoning, pp. 539–550 (1992)