Fault Tolerance Patterns Mining in Dynamic Databases

Delvi Ester and Guanling Lee(✉)

Department of Computer Science and Information Engineering,
National Dong Hwa University, Shoufeng, Taiwan, R.O.C.
guanling@mail.ndhu.edu.tw

Abstract. Mining of frequent patterns in database has been studied for several years. However, real-world data tends to be dirty and frequent pattern mining which extracts patterns that are absolutely matched is not enough. An approach, called frequent fault-tolerant pattern (FT-pattern) mining, is more suitable for extracting interesting information from real-world data that may be polluted by noise. Previous research on frequent fault-tolerant pattern mining has been widely studied. However, all of the researches focus on static database. In this paper, we propose an efficient framework to analyze the frequent FT-patterns mining in dynamic database. To avoid re-scanning the whole database, beside of keeping the fault-tolerance pattern, we will also keep the potential fault-tolerance pattern that has higher possibility of becoming a fault-tolerance pattern. The experimental results show that by re-using the existing pattern that had been generated, the proposed algorithms are highly efficient in terms of execution time and maximum memory usage for mining fault-tolerance frequent pattern in dynamic database compare to FFM algorithm.

Keywords: Item support · FT support · Fault-tolerant frequent pattern · Data mining · Dynamic database

1 Introduction and Related Work

Frequent pattern mining from a transactional datasets with support greater than a certain user defined threshold plays an important role in many data mining applications such as finding web log pattern, intrusion detection, etc. However, real-world databases contain noise that can make important information ambiguous; resulting in it will not appear in the mining result. Therefore, we need a method that copes with such variations in an association pattern (within predefined limits), which is called a fault-tolerant pattern. For example, coughing, fever, a headache, and a sore throat are all signs of catching a cold. However, these symptoms are seldom present at the same time, and hence a doctor will not diagnose the disease exactly following the rule RI: {coughing, fever, headache, sore throat} → {catch a cold}. Instead, a better rule corresponding to the real world situation would be R2: Patients who have at least three of the following symptoms {coughing, fever, headache, sore throat} are catching a cold. R2 requires matching just part of the data, which illustrates the sense of allowing for fault tolerance in data mining.

© Springer International Publishing Switzerland 2015
D. Palacios-Marqués et al. (Eds.): GIKA 2015, LNBIP 222, pp. 122–130, 2015.
DOI: 10.1007/978-3-319-22204-2_12

[5] is the first to propose the discovering of frequent FT pattern to find frequent groups of transactions instead of just focusing on the item themselves. Unfortunately, their approach may generate sparse patterns, which may contain sub-patterns that do not appear frequently. [4]developed FT-Apriori for frequent FT-pattern mining which allows a complete set of FT-patterns to be mines out. [1]uses bit vector representation to represent data and developed a vector-based mining algorithm, VB-FT-Mine. [2] introduces the problem of mining proportional FT-pattern which number of faults tolerable in the pattern is proportional to the pattern length. Moreover, the concept and proposed methods are demonstrated in [3] for predicting epitopes of spike proteins of SARS-CoV and concludes that the patterns reported by proportional FT-patterns mining are more concise than that of fixed FT-patterns mining for this application. [6][7]propose a proportional and fixed FT-pattern with candidate pruning that produce a better result than previous research. However, all the previous papers related to fault-tolerant are assume that the rules having been found in the datasets are valid all the time and do not change, as it consider as a static database.

Mining for association rules between items in a large database of transactions is an important database mining problem. However, all the previous researches related with fault-tolerant pattern were conducted in static database. That is, when new transactions are added or old transaction are deleted, the mining process must start all over again, without taking the advantages of previous execution and results of the mining algorithm. In this paper, we propose an efficient framework to analyze the frequent FT-patterns mining in dynamic database. This framework can solve the problems of mining patterns that tolerate fixed numbers of faults as well as to avoid re-scanning of entire database when there are new additional data or deletion of necessary transactions. The remainder of this paper is organized as follows. Section 2 introduces the problem definition and preliminaries. Section 3 describes the main idea and the proposed algorithms in detail. Section 4 discusses the experimental results and analysis. Conclusions are finally drawn in Section 5.

2 Problem Definition and Analysis

The goal of this work is to find the frequent FT-pattern if there are new transactions coming in or any deletion in the databases without the need of re-scanning the entire database.We use the following definitions and lemmas to explain the main idea of *dynamic frequent FT-pattern*.

Definition 2.1 (Frequent Fault Tolerance patterns/ FFT)[6]
Let P be a pattern and a given FT parameter δ *is defined as a fixed number (* $smaller\ than\ 0.5 \times |P|$*). A transaction T = (tid, X) is said to be FT-contain pattern Piff there exists* $P' \subseteq P$ *such that* $P' \subseteq X$ *and* $|P'| \geq \lceil |P| - \delta \rceil$*.The number of transactions in a database FT-containing pattern P is called the FT-support of P, denoted as* $sup^{FT}(P)$*.*

Let B(P) be the set of transactions FT-containing pattern P. Given a frequent item-support threshold min_sup^{item} *and a FT-support threshold* min_sup^{FT}*, a pattern P is called a frequent FT-pattern if*

1. $sup^{FT}(P) \geq min\_sup^{FT}$*; and*
2. *for each item* $p \in P$, $sup^{item}_{B(P)}(p) \geq min\_sup^{item}$*, where* $sup^{item}_{B(P)}(p)$ *is the number of transactions in B(P) containing item p.*

Table 1. An example TDB

TID	Items
10	a, b, c, f
20	c, d, e, f
30	e, f, g
40	e, f, h
50	a, b, c, d
60	a, b, d, e
70	a, b, d
80	e, f, g, h
90	f, g, h, i, j
100	j, x, y, z

Example 2.1 (FFT). Table 1 shows a transaction database *TDB*. Suppose that the FT-support threshold $min\_sup^{FT} = 4$, the minimal item-support threshold $min\_sup^{item} = 2$, and the FT parameter $\delta = 2$. For pattern $P = abcde$, $B(P)$ includes transaction 10, 20, 50, 60, 70, since they all FT-contain P, we have $sup^{FT}(P) = 5$. Each item of P appears in at least two transactions of $B(P)$. Therefore, pattern P is a frequent FT-pattern and recorded as 5 | 3 4 3 4 2 (FT support | array of item support) or $abcde$ = 5 | 3 4 3 4 2, alternatively.

To avoid rescanning the whole database when data insertion and deletion, except the information of frequent fault tolerance patterns, we also record the information of *potential frequent fault tolerance pattern* (*PFFT*, in short). Following is the definition of *PFFT*.

Definition 2.2 (Potential Frequent Fault Tolerance patterns/PFFT)
Let $\alpha(itm)$ *and* $\alpha(FT)$ $(0<\alpha(itm), \alpha(FT)<1)$ *denote Potential item support and Potential FT support, respectively. Given a potential frequent item-support threshold* $Pmin\_supitem = \alpha(itm) * Pmin\_supitem$ *and potential FT-support threshold* $Pmin\_supFT = \alpha(FT) * Pmin\_supFT$*, a pattern P is called a potential frequent FT-pattern if*

1. $sup^{FT}(P) \geq Pmin\_sup^{FT}$*; and*
2. *for each item* $p \in P$, $sup^{item}_{B(P)}(p) \geq Pmin\_sup^{item}$*, where* $sup^{item}_{B(P)}(p)$ *is the number of transactions in B(P) containing item p.*

In our approach, we use D, d^+ and d^- to indicate the original, inserted and deleted databases, respectively. Moreover, we use *FFT(H)* and *PFFT(H)* to indicate the set of *FFT* and *PFFT* mined out from database H, respectively.

The following lemmas show the relationship between *FFT* and *PFFT*.

Lemma 1. *If a pattern* X' *belongs to* $FFT(d^+)$*, and does not belong to* $FFT(D)$ *or* $PFFT(D)$*,* X' *might belong to* $PFFT(D \cup d^+)$.

Lemma 2. *If a pattern* X' *belongs to* $FFT(D)$*, and does not belong to* $FFT(d^+)$ *or* $PFFT(d^+)$*,* X' *might belong to* $PFFT(D \cup d^+)$.

Lemma 3. *If a pattern* X' *belongs to* $FFT(D)$ *and* $FFT(d^+)$*, it must belong to* $FFT(D \cup d^+)$.

Lemma 4. *If a pattern* X' *belongs to* $PFFT(D)$ *and* $PFFT(d^+)$*, it must belong to* $PFFT(D \cup d^+)$.

3 Dynamic Fault Tolerance Pattern Mining

The whole process of proposed approach can be decomposed into two separate algorithms. The first algorithm, Patterns Generation (Patterns-Gen) Algorithm, is based on the main concept of Frequent Fault Tolerance Pattern mining algorithm (*FFM*) proposed in [6]. The major difference is that we not only mine out the *FFT(D)*, we also record the information of *PFFT(D)* during the mining process. The second algorithm is the Dynamic Fault Tolerance Patterns (*DFT*) Algorithm. In the second algorithm, we will get all the patterns generated from *FFM* algorithm for both *D* and d^+/d^-, combine it, recalculating the supports and get the Final Frequent Fault Tolerance Pattern (Final *FFT*) and Final Potential Frequent Fault Tolerance Pattern (Final *PFFT*). Because of the space limitation, we only explain the second algorithm in detail in this article.

When new transactions are added into the original database; or current transactions being deleted, we will re-execute Pattern-Gen algorithm in order to find the *FFT* and *PFFT* from the modified database. After re-executing, we will have two sets of *FFT* and *PFFT* from *D* and d^+/d^-.In this algorithm, the mining process is decomposed into three parts. The first part is database merging, to merge the *FFT* and *PFFT*'s results from both databases into combined temporary results as candidates' patterns. The second part is checking candidates' pattern. We are going to check and determine the final *FFT* and final *PFFT* based on the minimum supports. The last part is to update the current bitmap and that had being loaded from file into the memory by adding/deleting transactions from incremental database |*d*|. The updated bitmap is then saved back to the file for future FT-pattern mining.

A detailed description of *DFT* (Dynamic Fault Tolerance Algorithm) for mining frequent FT-pattern in adding algorithm and deleting algorithm while avoid re-scanning the original/existing database is described in Figure 1. We explain the further detail of *DFT* algorithm in three sub parts below.

3.1 Merging Databases

In line 2, we read original/latest bitmap, *FFT*, *PFFT* as well as all minimum supports (min_sup^{FT}, min_sup^{item}, $Pmin\_sup^{item}$, $Pmin\_sup^{item}$) from previous generated data-

base. Then in lines 3 ~ 9, we merge all patterns found from each original/existing database D and incremental database $|d|$. The result is then saved to a temporary variable called *tmp*. While merging, we also update the *FT* support of each pattern $sup^{FT}(P)$ and item support for each item in each pattern $sup^{item}B(P)$. After merging, we re-count all minimum supports (min_sup^{FT}, min_sup^{item}, $Pmin\_sup^{item}$, $Pmin\_sup^{item}$) from merging database as described in lines 10 ~ 19.

3.2 Checking Candidates' Pattern

All candidates' pattern in *tmp* are then checked for their $sup^{FT}(P)$ and $sup^{item}{}_{B(P)}(x)$ to determine whether pattern *P*is still a *FFT* or *PFFT*in the modified database. After getting the pattern support, the next step is to check whether the support of pattern is complied with the minimum support. In line 21, by checking the status " if $sup^{FT}(P) < Pmin\_sup^{FT}$", the boolean variable named is Discarded value is set to true and no further checking is required as pattern with support smaller than potential support. For the pattern whose $sup^{FT}(P) \geq min\_sup^{FT}$, we will check whether the pattern is frequent or potential FT-pattern. For each item x in pattern P, we compare item support $sup^{item}B(P)(x)$ with minimum support. If $sup^{item}B(P)(x) \geq min\_sup^{item}$ of each item, the pattern might be a *FFT* and we increase the variable $countSup^{Itm}(P)$ by 1; otherwise if $sup^{item}B(P)(x) \geq Pmin\_sup^{item}$,thenit might be a *PFFT* pattern, and we increase variable $countPSup^{Item}(P)$ by 1 (lines 22 ~ 28). At the end, described in lines 29 ~ 38, we determine whether the added pattern is truly *FFT*.

After getting the merged patterns of Final *FFT* and Final *PFFT*, we check is Discarded variable to determine whether the discarded pattern is *PFFT* pattern. This variable is used to comply with Lemmas 1-4. As describes in lines 39 ~ 49, we only check the patterns that is Discarded. Therefore, only small part of bitmap for pattern *P*is extracted from bitmap(D). For pattern whose $sup^{FT}(P) \geq Pmin\_sup^{FT}$, we will check the possibility that the pattern will be a potential FT-pattern. After that, we compare the support count with the minimum item support. If $sup^{item}{}_{B(P)}(x) \geq Pmin\_sup^{item}$, for each item, the pattern might be a *PFFT* pattern and therefore we increase variable $countPSup^{Itm}(P)$ by 1. Finally line 51 checks whether $countPSup^{Itm}(P) \geq |P|$, therefore we can insert pattern *P* to Final *PFFT*.

3.3 Updating Bitmap

This section will be executed after we got all the appropriate final *FFT* and *PFFT*. Line 49 is performed to merge the bitmaps. During the bitmap updating process, for new coming transactions, we insert the data into the last row in the bitmap. And for new coming item, we create a new bitmap's column and set the entrances to 0 for the original transactions (i.e., the transactions of D). Furthermore, for those transactions which are deleted, we eliminate transactions row in the original database based on the user input.

Algorithm 2. (Dynamic Fault Tolerance Algorithm (DFT))

Input: additional database $|d|$
Minimum item-support threshold: min_sup^{item}
Minimum FT support threshold: min_sup^{FT}
Minimum Potential item-support threshold: $Pmin\_sup^{item}$
Minimum Potential FT support threshold: $Pmin\_sup^{FT}$
FT parameter: δ
TID from and to for transaction rows to be delete from database D

Output: bitmap(D ∪ |d|), Final FFT and Final PFFT

Method:

1. Execute Patterns-Gen Algorithm for additional/delete database
2. //read latest bitmap, latest FFT, PFFT and minimum supports from files
3. tmp← FFT^D∪ $PFFT^D$ ∪$FFT^{|d|}$∪$PFFT^{|d|}$
4. if AddingData {
5. $supFT(P) = supFT(P)$ D +$supFT(P)$ |d| ;
6.$supitemB(P) = supitemB(P)$ D + $supitemB(P)$ |d| ;}
7. if DeletingData {
8. $supFT(P) = supFT(P)$ D − $supFT(P)$ |d| ;
9.$supitemB(P) = supitemB(P)$ D − $supitemB(P)$ |d| ;}
10. if Adding Data {
11. $min\_sup^{item} = min\_sup\ D^{FT} + min\_sup\ |d|^{item}$;
12. $min\_sup^{FT} = min\_sup\ D^{FT} + min\_sup\ |d|^{FT}$;
13. $Pmin\_sup^{item} = Pmin\_sup\ D^{FT} + Pmin\_sup\ |d|^{item}$;
14. $Pmin\_sup^{FT} = Pmin\_sup\ D^{FT} + Pmin\_sup\ |d|^{FT}$;}
15. if Deleting Data {
16. $min\_sup^{item} = min\_sup\ D^{FT} - min\_sup\ |d|^{item}$;
17. $min\_sup^{FT} = min\_sup\ D^{FT} - min\_sup\ |d|^{FT}$;
18. $Pmin\_sup^{item} = Pmin\_sup\ D^{FT} - Pmin\_sup\ |d|^{item}$;
19. $Pmin\_sup^{FT} = Pmin\_sup\ D^{FT} - Pmin\_sup\ |d|^{FT}$;}
20. for each pattern P in tmp {
21. if $supFT(P) < Pmin\_supFT$
23. isDiscarded;
22. else if $supFT(P) \geq min\_supFT$
23. for each item support of x in $supitemB(P)$ {
24. if $supitemB(P)(x) \geq min\_supitem$
25. countSupItm = countSupItm + 1;
26. else if $supitemB(P)(x) \geq Pmin\_supitem$
27. countPSupItm = countPSupItm + 1;
28. }
29. if countSupItm≥|P|; add P to Final $FFTi$;
30. else if (countPSupItm + countSupItm) ≥|P|; add P to Final $PFFTi$;
31. else isDiscarded;
32. else if $supFT(P) \geq Pmin\_supFT$

Fig. 1. DFT Algorithm

```
33.                 for each item support of x in supitemB(P) {
34.                         if supitemB(P)(x) ≥Pmin_ supitem
35.                                 countPSupItm = countPSupItm + 1;
36.                 }
37.                 if countPSupItm≥|P| then add P to Final PFFTi;
38.                 else isDiscarded;
39.     if is Discarded and P ∈ FFT^|d|   and Adding Data
40.         if P ∈ FFT^|d|   and P ∉ FFT^|D|
41.                 Extract bitmap(P) from bitmap(D);
42.           if P ∈ FFT^D   and P ∉ FFT^d
43.                 Extract bitmap(P) from bitmap(d);
44.         get XsupFT(P) and XsupitemB(P) from bitmap(P) ;
45.         supFT(P) = supFT(P) d + XsupFT(P) ;
46.         if supFT(P) ≥Pmin_ supFT
47.             for each item count of    x in P from supitemB(P) d + XsupitemB(P)){
48.                 if (supitemB(P)(x) + XsupitemB(P)(x))    ≥Pmin_ supitem
49.                     countPSupItm = countPSupItm + 1;
50.             }
51.             if countPSupItm≥ |P| then add P to Final PFFTi;
52.   }
53. Merge bitmap from both DB and |d|;
54. Save bitmap, FFT, PFFT and all minimum support to file
```

Fig.1. (*Continued.*)

4 Experimental Results

In this section, a set of simulations were performed to show the benefit of our approach. The test data was generated by using IBM synthetic-data generator. All experiments were performed by an Intel i 5 3.2 GHz computer with 4GB of memory, running Windows 7. Table 2 lists the parameter settings for generating the test datasets and evaluating our approach with prior works. Moreover, we split the proposed *DFP* algorithm into two parts, data insertion and data deletion, to make an easier comparison.

Figures 2 and 3 show the effects of number of distinct items in the database. Refer to figure 2, the execution times increases as the number of distinct items in the database increases for both approaches. The reason is that the larger the number of items in the database, the larger the number of patterns will be generated during the mining process which result in longer execution time. Moreover, our approach outperforms the *FFM* algorithm due to the reason that we need not to rescan the whole database. Figure 3 shows the memory usages for both approaches. Because we only need to check the patterns belong to "is Discarded" type, only a small part of bitmap is extracted from bitmap(*D*). Therefore, our approach outperforms the previous approach.

Table 2. Parameter Settings for FFM and DFP algorithm

Notation	Meaning	Default	Range
TLEN	Average length of a transaction	10	
I	Number of distinct items	15	10 ~ 30
N	Total number of transactions	100000	-
n	Total number of added/deleted transactions	10000	1000 ~ 200000
δ	FT parameter	1	-
min_sup^{item}	Minimum item support threshold	0.075	
min_sup^{FT}	Minimum FT support threshold	0.1	
$Pmin\_sup^{item}$	Potential minimum item support threshold	0.05625	
$Pmin\_sup^{FT}$	Potential Minimum FT support threshold	0.075	

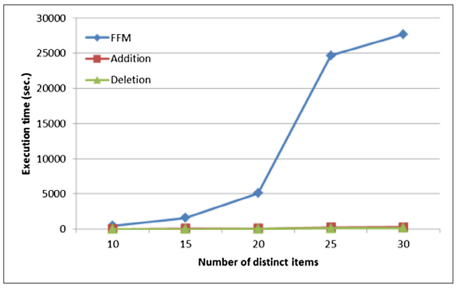

Fig. 2. Execution times

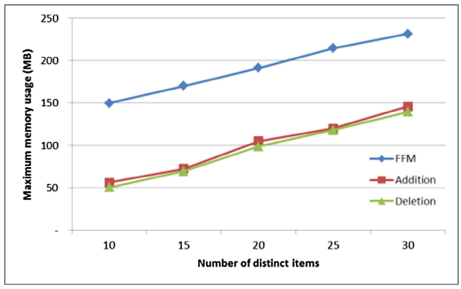

Fig. 3. Maximum memory usages

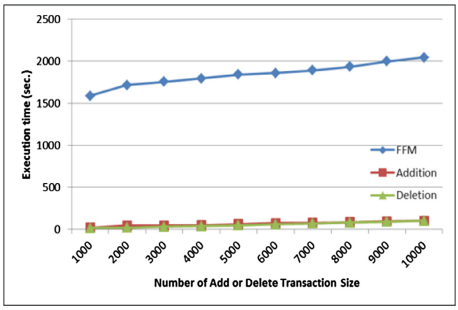

Fig. 4. Execution times

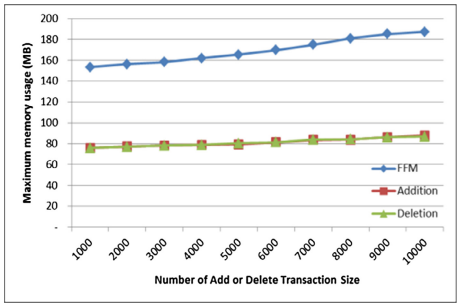

Fig. 5. Maximum memory usagesvs. the modification database sizevs. the modification database size.

Figures 4 and 5 show the effects of the modification database sizes. As shown in the results, the execution times and memory usages increase as the database modification sizes increase for both approaches. Obviously, our approach outperforms the previous one due to that we avoid rescanning the whole database and only load part of the bitmap during the mining process.

5 Conclusion and Future Works

In this paper, a framework is proposed to handle database updates while keeping the performance on hand. To avoid rescanning the whole database, in our approach, we generate the frequent fault tolerance patterns and potential frequent fault tolerance patterns in the original databases and re-using it by merging the patterns with the updated patterns when database is modified. The whole process of the approach can be decomposed into two separate algorithms. The first algorithm, Patterns Generation (Patterns-Gen) Algorithm, is based on the main concept of Frequent Fault Tolerance Pattern mining algorithm(*FFM*). And, the second algorithm is the Dynamic Fault Tolerance Patterns (*DFT*) Algorithm. In the second algorithm, we will get the information of the patterns generated from *FFM* algorithm for both D and d^+/d^-, combine it, recalculating the supports and get the Final Frequent Fault Tolerance Patterns and Final Potential Frequent Fault Tolerance Patterns. Experimental results show that by re-using the existing pattern that had been generated, the proposed algorithms are highly efficient in terms of execution time and maximum memory usage for mining fault-tolerance frequent pattern in dynamic database.

Acknowledgements. The authors would like to thank the anonymous referees for their helpful suggestions. This research was partially supported by the Ministry of Science and Technology of the Republic of China under Contract No. MOST 103-2221-E-259-022.

References

1. Koh, J.-L., Yo, P.-W.: 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. 568–575. Springer, Heidelberg (2005)
2. Lee, G., Lin, Y.-T.: A study on proportional fault-tolerant data mining. In: Proc. of Int. Conf. Innovation in Information Technology, Dubai, pp. 1–5, November 2006
3. Lee, G., Peng, S.-L., Lin, Y.-T.: Proportional Fault-tolerant Data Mining with Applications to Bioinformatics. Special Issue of Knowledge Discovery and Management in Biomedical Information Systems with the journal of Information Systems Frontiers (2009 to appear)
4. Pei, J., Tung, A.K.H., Han, J.: Fault-tolerant frequent pattern mining: problems and challenges. In: DMKD 2001, Santa Barbara, CA, May 2001
5. Yang, C., Fayyad, U., Bradley, P.S.: Efficient discovery of error-tolerant frequent item sets in high dimensions. In: Proceedings of the Seventh ACM SIGKDD International Conference On Knowledge Discovery And Data Mining, August 2001
6. Zeng, J.-J., Lee, G., Lee, C.-C.: Mining fault-tolerant frequent patterns efficiently with powerful pruning. In: Proc. of ACM, pp. 927–931 (2008)
7. Tseng, C.-C., Lee, G.: A depth-first search approach for mining proportional fault-tolerant frequent pat-terns efficiently in large database. In: International Conference on Information Management, February–March 2015