COMPARISON OF DYNAMIC RULE MINING ALGORITHMS

A Thesis submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Karunda SHARIFF

August 2012
İZMİR

We approve the thesis of **Karunda SHARIFF**

Examining Committee Members:

Assist. Prof. Dr. Belgin ERGENÇ
Department of Computer Engineering
İzmir Institute of Technology

Prof. Dr. Oğuz DİKENELLİ
Department of Computer Engineering
Ege University

Inst. Dr. Selma TEKİR
Department of Computer Engineering
İzmir Institute of Technology

29 August 2012

Assist. Prof. Dr. Belgin ERGENÇ
Supervisor, Department of Computer Engineering
İzmir Institute of Technology

Prof. Dr. İ. Sıtkı AYTAÇ
Head of the Department of
Computer Engneering

Prof. Dr. R. Tuğrul SENGER
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGMENTS

Many thanks to the Turkish government, that sponsored my education, financed my wellbeing including medical expenses during my stay in Turkey. I say life would have been almost impossible without the Turkish government. Then I cannot thank enough my cherished supervisor Assist. Prof. Dr. Belgin ERGENÇ who was always available to me all the time for her very much appreciated advice, guidance and council during my study and writing of this thesis. It was such a great honor working with her. Many thanks to the professors from whom i took classes, the knowledge I gained will be of great value in my career. I also acknowledge the support and love from my fellow students especially Damla Oguz and Beakal Assefa who guided me in one way or another during my study period and writing of my research.

ABSTRACT

COMPARISON OF DYNAMIC RULE MINING ALGORITHMS

In real life, new data is constantly added to databases while the existing one is modified or deleted. The new challenge of association rule mining is the need to always maintain meaningful association rules whenever the databases are updated. Many dynamic algorithms that use different techniques have been proposed in the past to deal with this challenge. However less work has been done in comparing their performance. In this study comparison of two dynamic rule mining algorithms; Dynamic Matrix Apriori and Fast Update 2, which have not been compared in the past, is done. The algorithms are tested on three different datasets to determine their execution time with updates of: additions, deletions and different support thresholds. Our findings reveal that DMA performs better with two dataset and so is FUP2 with the other dataset. The difference in performance of the two algorithms is mainly caused by the nature of the datasets.

ÖZET

DEVİNGEN KURAL MADENCİLİĞİ ALGORİTMALARININ KARŞILAŞTIRILMASI

Gerçek hayatta, bir yandan veritabanlarında duran veri güncellenmekte ya da silinmekte iken bir yandan da yeni veri akışı devam etmektedir. Sürekli değişen veri tabanlarından anlamlı ilişki kurallarını bulmak yeni bir zorluk olarak karşımıza çıkmaktadır. Bu zorlukla başetmek için önemli miktarda devingen ilişki kuralları madenciliği algoritmaları geliştirilmiştir. Buna rağmen bu algoritmaların başarımlarını karşılaştıran çalışmalara aynı oranda rastlanmamaktadır. Bu tezde iki devingen ilişki kuralı madenciliği algoritması "Dynamic Matrix Apriori (DMA)" ve "Fast Update 2 (FUP2)" karşılaştırılmaktadır. Bu algoritmaların başarımları gelen farklı miktardaki ekleme ve silme taleplerinde ve değişen destek eşiklerinde, üç farklı veri seti ile ölçülmüştür. Bulgularımıza göre DMA iki set ile daha iyi başarım sergilerken FUP2 diğerinde daha etkili olmuştur. Veri setlerinin özelliklerindeki değişiklikler bu başarım farkına neden olmaktadır.

DEDICATION

I dedicate this work to my beloved Mother Mrs. Nakabubi Mariat. From the time I was a little child, she taught me the value of education and she struggled all her life to ensure that I achieved success. I thank her so much and may the Almighty Allah reward her with Paradise for her struggle towards my success.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Recently intensive research focused on association rule mining, which is one the main topic in data mining. Association Rule Mining has received greater focus because of its applicability in decision support, share market, layout of shelves in supermarkets, web log analysis, text mining, understanding customer behavior, telecommunication alarm diagnosis, and prediction. [1]. Association rule was first introduced by Agarwal et al. [13] and is defined as, X% of the customers who buy item A also buy item B" (formulated as A⟹B). Therefore association rules are meant to find the impact of one set of items on another set of items.

1.1. Mining of Association Rules

Association rule mining has two problems: (1) finding frequent itemsets/patterns, (2) generating association rules [14]. Of the two, the first is the most challenging and hence the focus of many studies. Many powerful algorithms have been proposed to find the frequent itemsets from massive databases. The most classical one is the Apriori algorithm [14] that uses candidate generation and testing approach to discover frequent itemsets. Later on other algorithms using Apriori-like technique were introduced in [2, 3, 4, 5, 6, 7, 8, 9, and 10]. Because of the drawback of candidate generation and multiple hits on the database in Apriori-like algorithms, algorithms that do not depend on candidate generation were introduced, for example FP-growth and Matrix Apriori. In [12] FP-growth uses a tree data structure while Matrix Apriori [11] depended on the matrix to store candidates.

1.2. Dynamic Association Rule Mining

One main assumption in all the above-mentioned algorithms is that database is static, but in real life, databases are constantly updated with new data, and old data may as well be deleted or modified. This implies, the originally discovered association rule may no longer be valid and yet new interesting rules may emerge as a result of an update on the database. The most straightforward way to update the rules would be to repeat the entire mining process from scratch however this is very expensive in terms of execution time and memory allocation. Therefore many efficient algorithms both dependent on candidate generation and non- candidate generation techniques have been introduced in [15, 16, 17, 19, 27, 28, and 29]. The algorithms perform faster and use less system resources than repeating the entire mining processes.

1.3. Aim of the Thesis

Many dynamic rule mining algorithms have been introduced and their performances have been compared with their parent algorithms. Comparisons have also been made for between candidate generation algorithms and non-candidate generation using a tree data structure.

The aim of this thesis is to make a comparison between Apriori-like algorithm and non-candidate generation algorithm that uses a matrix data structure. Since no work in past was done on comparing matrix based algorithm and Apriori based ones, the objective of the present research is to determine the performances of the algorithms in terms of execution time when;

- Transactions of different sizes are added to the database
- Transactions of different sizes are deleted from the original database
- The algorithms are executed for varying minimum supports

1.4. Organization of the Thesis

The rest of the thesis is organized as follows.

- Chapter 2 is dedicated to related work. This chapter makes a study on the past static association rule mining algorithms, the existing dynamic association rule mining algorithms are also reviewed.
- In chapter 3, a detailed discussion including giving examples to two dynamic algorithms (FUP2 and DMA) which are the focus of the study is done.
- Chapter 4 is dedicated to the performance studies of the algorithms. Illustrations showing the behavior of FUP2 and DMA with three different datasets when; 1) new increments arrive, 2) different transaction sizes are deleted from the database and 3) the minimum support threshold is changed are presented.
- Chapter 5 is the conclusion of the thesis and future work.

CHAPTER 2

RELATED WORK

In this section literature related to association rule mining is reviewed. In order to address the challenges of association rule mining, which is finding frequent itemsets many algorithms have been proposed recently. Some used candidate generation techniques while others used techniques that eliminate candidate generation procedures. To some extent comparisons were made to determine which algorithms performed better than others [26]. Despite of the methods used the algorithms assumed databases to be static, which is not the case in real world scenarios.

In real life, databases are updated and therefore the integrity of association rules needs to be maintained. The most straightforward method is to rerun the entire mining process from scratch. This is time consuming for even a very small update to the database. As a solution, static algorithms were improved to handle cases with database updates and this led to the evolution of dynamic/incremental rule mining algorithms. Some of dynamic association rule mining algorithms use candidate generation techniques while others eliminated candidate generation by creating the signature of the databases in trees, matrices and vectors.

In the remainder of the chapter, to be self-contained a discussion reviewing static association rule mining algorithms that form the core of dynamic rule mining algorithms is given in the first section. In the second section the categorized dynamic algorithms are discussed.

2.1. Static Algorithms

Apriori [14] is a very influential algorithm which has been used for finding association rules in large transaction databases. Figure 2.1 shows the working principle of the algorithm.

```
1) $L_1$ = {large 1-itemsets};
2)  for ( k = 2; $L_{k-1} \neq \emptyset$; k++ ) do begin
3)     $C_k$ = apriori-gen($L_{k-1}$ ); // New candidates
4)     for all transactions t $\in$ D do begin
5)     $C_t$ = subset($C_k$ , t);// Candidates  in t
6)       for all candidates c $\in$ $C_t$ do
7)         c.count++;
8)     end
9) $L_k$ = {c $\in$ $C_k$ |c.count $\geq$ min_support}
10)  end
11)  Return = $L_k$;
```

Figure 2.1. Apriori Algorithm [14]

In the first pass, the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, for example pass k, consists of two phases; 1), the large itemsets L_{k-1} found in the (k-1)th pass are used to generate the candidate itemsets C_k, using the Apriori candidate generation function (apriori-gen); and 2) Next, the database is scanned and the support of candidates in C_k is counted to determine frequent itemsets. At each path the algorithm determines new k-frequent itemsets. The main bottleneck hindering the performance of Apriori algorithm is candidate generation and repeated scans to the database.

FP-growth Algorithm [12], as a solution to the problem of *candidate generation and test* face by Apriori-like methods, FP-growth was developed to discover frequent itemset by eliminating candidate itemset generation. It uses a compact data structure known as Frequent Pattern tree (FP-tree) and based on the tree frequent patterns are generated. In Figure 2.2 the construction of the FP-tree is displayed.

```
Input: Database DB, Minimum support s.
Output: FP-tree, Frequent-pattern tree of DB.

1. Scan DB once. Collect F, the set of frequent items,
   and their support
2. Sort F in support-descending order to form FList.
3. Create the root of FP-tree, T, and label it "null".
4. For each transaction Trans in DB do
5.   Sort Trans according to FList [p | P],
          where p = first element
                P = the remaining list.

6.   Call insert tree ([p | P], T).
7. Return Frequent-pattern tree
```

Figure 2.2. FP-tree Construction Process

FP-growth algorithm involves two main processes (Figure 2.2). Firstly, FP-tree is built. This requires exactly 2 passes over the database. In the first pass, it finds the support for each item in the database eliminating infrequent ones and then sorts frequent items in decreasing order based on their support. In the second pass, FP-Tree is constructed. All frequent items in each transaction are represented in the tree. A sample tree is shown in Figure 2.3.

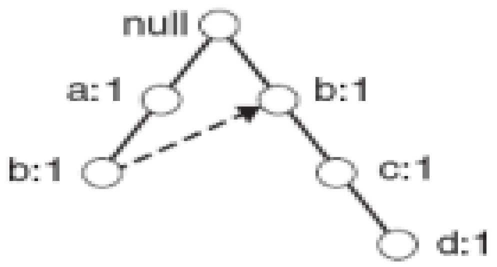


Figure 2.3. FP-tree for 2 Transactions

Secondly, frequent itemsets are extracted directly from the FP-tree. It uses a bottom-up strategy from the leaves towards the root. Then a prefix path sub-trees ending in each itemset is extracted. Next, each prefix path sub-tree is processed recursively to get the other frequent itemsets ending in a particular itemset which requires building an addition tree called a co*nditional FP-tree* (which is basically an FP-Tree that would be built if only transactions containing a particular itemset are considered). The above process is applied to every frequent itemset and then all subsequent frequent items are determined.

Lastly it is noted that FP-tree makes only two scans to the database and a performance study in [12] shows that it performs better than Apriori. However, it's expensive in terms of time required to build the FP-tree and conditional FP-tree. Moreover the trees may not fit in the memory especially for large databases with unique transaction patterns.

Matrix Apriori [11] is another algorithm that eliminates candidate generation techniques of apriori and different from FP-growth algorithm. It uses a matrix structure and a vector to determine frequent itemsets. It makes two passes to the database. In the fast pass it extracts itemsets and their support, gets frequents itemsets and sorts them in ascending order according to their support. In the second pass to the database, a matrix of frequent itemsets called MFI and a vector called STE are constructed as illustrated in Figure 2.4.

```
Input: Database DB, Minimum support s.
Output: DMA, STE, Frequent-pattern FP.

   1. Scan DB once. Collect FList, the set of frequent
      items, and their support
   2. Sort FList by ascending support
   3. FOR each item in FList
   4. Add a column to MFI
   5. END FOR
   6. FOR each transaction in DB
   7.  IF the transaction exists in MFI
   8.  Increment its support in STE by 1
   9.  ELSE
   10.       Add a new line to MFI
   11.       Set the STE of the new line to 1
   12.END FOR
   13.Modify MFI// speedup search for FP
   14.Generate FP from MFI
   15.Return MFI, STE, F
```

Figure 2.4. Frequent Pattern Generation Process for Matrix Apriori

For all the transactions in the database (example is given in Figure 3.5), the frequent items are identified and sorted by support value. And each resulting set is a candidate set which is then inserted into MFI starting from the second row. The columns of MFI represent items while the rows represent candidate sets. If an item is available in the candidate set, its column cell is set to 1, otherwise it is set to 0. The support count of the candidate set is set to 1 in STE. And if the candidate set is already presented in MFI then its count in STE is Incremented by 1. After the construction of MFI and STE is completed, the next step is to modify MFI to support indexing. This modification helps in speeding up the search for frequent patterns and is done as follows; first of all starting with the first row of MFI that was left empty; each cell in first row is updated with the row number to which the value of 1 appears for the first time in a particular column. Next for the remaining rows except the last one with a column value of 1 are replaced with the next row number with a column value equal to 1.

Lastly frequent itemsets are generated from MFI and STE. This involves combining a conditional pattern item with the frequent items found on its left hand side in MFI, and then calculating its corresponding support, then it checks whether the analyzed combination is a frequent pattern. The process executes recursively until all frequent itemsets are obtained. Constructing MFI and STE are less expensive that constructing FP-tree in FP-growth. And studies show that MFI has a better performance over FP-growth.

2.2. Dynamic Algorithms

The dynamic/incremental rule mining algorithms are categorized in two: 1) apriori-like algorithms which depend on candidate generation and 2) algorithms which eliminate candidate generation.

2.2.1. Dynamic Algorithms with Candidate Generation

FUP (Fast Update) [16] has a framework similar to that of apriori and is also the pioneer incremental association rule mining algorithm in large database. FUP handles only a special case of transaction addition. To maintain association rules when new transactions are added to the database, FUP follows the following procedures:

Firstly, during the first iteration the support count of size-1 frequent itemset is updated against the newly added transactions. In so doing, some originally frequent itemsets may no longer be frequent thus becoming losers. Losers are determined by scanning only the newly added transactions. Then in the same iteration, a set of size-1 candidate itemsets is generated from the incremental transactions, which is a set of items that were not frequent in the old database. Next since the candidate sets where not frequent in the old database means that they must be frequent in the incremental transactions before they can be updated against the old database to determine if they are frequent in the updated database. Lastly at the end of the iteration a set of size-1 frequent items from which new candidate itemsets are generated is obtained.

In a subsequent iteration, FUP performs filtering two times to remove losers; 1) on size$-k$ frequent itemsets using losers from previous size $(k-1)$ frequent itemsets and 2) other on generated candidates. However, the algorithm also suffers from the bottleneck of apriori-like algorithms, which is candidate generation especially in cases when the size of the increment database is large.

FUP2 (Fast Update 2) [17] is an extension of FUP. Both algorithms use the same method to update the database when new transactions are added to the database. Whereas FUP handles only the incremental cases, the FUP2 algorithm is applicable to both incremental and deletion cases. It is also important to note that FUP2 like FUP assumes a constant minimum support threshold and are not applicable to situations where minimum support change is required. It also uses Apriori [14] procedures to determine new frequent items when a section of transactions are deleted from the databases. FUP2 is discussed in details in Chapter 3.

The Borders algorithm [15] is the first dynamic algorithm with the capability to handle all cases of: transaction addition, transaction addition and deletion and change in minimum support threshold. While incremental algorithms [16, 17] prior to border addressed transactions incremental and deletion, they never considered situations where there is need to change the minimum support. Borders algorithm uses the concept of border sets [18].

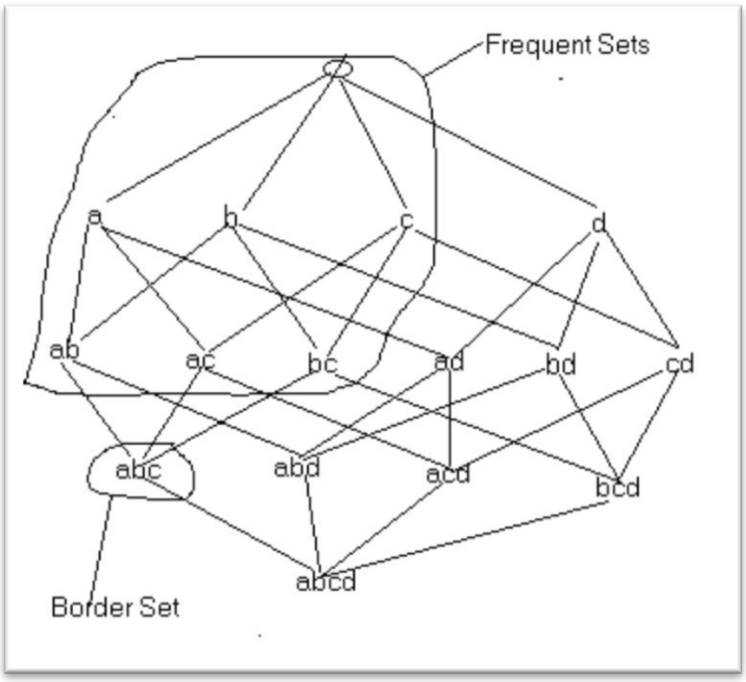


Figure 2.5. Border Set Example

An itemset A is considered a border set, if A is not frequent but all its proper subsets are frequent as shown in Figure 1. When the database is updated, the new frequent itemsets are expected to occur only if some border itemsets have reached the minimum support threshold and hence becomes frequent itemsets called *promoted border sets*. Borders algorithm further maintains that scanning the old database is only performed if some border sets have reached minimum support and become promoted border sets. Like in apriori algorithm , same apriori concepts is employed in the borders algorithm to generate candidates, Several scan to the old database when new candidates sets arise in addition to the cost of updating and maintaining the border set. However performance comparisons [15] reveal that border algorithm performs better than FUP.

IARMUPFI (Incremental Association Rule Mining Using Promising Frequent Itemset Algorithm) [19] is another incremental algorithm based on the frame work of apriori capable of handling transaction addition cases. To achieve greater performance IARMUPFI uses a new approach of promising frequent itemsets.

During the first execution of the algorithm before new transactions arrive, IARMUPFI executes similarly to Apriori. However, in each of the iteration, additional set of promising frequent size-k itemsets is stored. These promising frequent item sets have their minimum support below the set minimum support threshold, equal or above a certain support threshold(Min_{pl}),which is defined as in equation 1:

$$Min_{pl} = min\_supportDB - \left(\frac{Max_{support}}{Total_{size}}\right) * Inc_{size} < \min\_ supportDB \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \quad (2.1)$$

Where $Max_{support}$ is the maximum frequent size-k frequent itemset, $Total_{size}$ is the total number of transactions in the old database, and Inc_{size} is the maximum expected incremental size of the transactions

When new transactions arrive, min_{pl} must be calculated again since the size of the increment is known. Equation 2

$$Min_{pl(update)} = min\_sup(update) - \left(\frac{Max_{support}}{Total_{size}}\right) * Inc_{size} \dots\dots.. \quad (2.2)$$

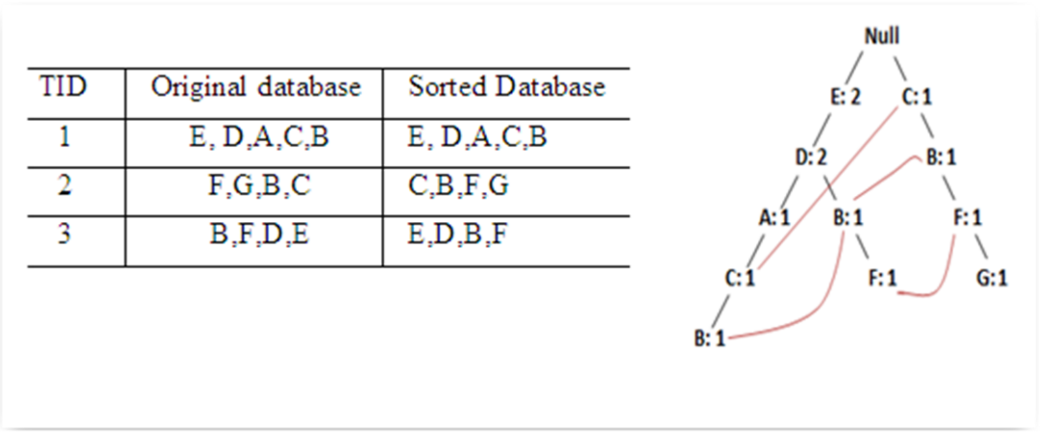
Then, candidate 1-itemsets of an incremental database are extracted along with their support count which is updated against the support count of 1-itemset candidates of the original database to determine their new support count in the updated database. If any item meets the specified minimum support, $min\_sup(update)$, this item is moved to Large 1-itemset of an updated database. Otherwise, if any item has support count less than $min\_sup(update)$ but greater or equal to $Min_{pl(update)}$, this item is moved to promising frequent 1-itemset of an updated database.

Furthermore, if any item is a *new* frequent item or a *new* promising frequent item, this item will be added to another set called Temp1. This set is joined and pruned the same way like in apriori [14] to obtain new candidates named $Temp\_newCk$. A k-itemset of $Temp\_newCk$ can become a frequent itemset or promising frequent itemset in an updated database only if it is a frequent itemset in the incremental database (db), which means that an itemset is moved to estimated frequent k-itemset, if it has support count greater than or equal to $min_{sup(db)}$. Likewise, the $k-$itemset is moved to an estimated promising frequent k-itemset it has support count less than $min\_sup(db)$ but greater or equal to $min\_pl(update)$ or $min\_PL(DB)$. Lastly, both the estimated promising frequent $k-$itemsets and the estimated frequent $k-$itemsets are scanned against DB to determine frequent and promising k-itemsets in the updated database.

To obtain new candidates for the next iterations, $k-$frequent itemsets and $k-$promising frequent itemsets are joined. The process is repeated until no more candidates can be generated, which is when $L_K = \emptyset$.

2.2.2. Dynamic Algorithms without Candidate Generation

TIARM (Tree-based Incremental Association Rule Mining) [20**],** is an extension of FP-growth algorithm and mines frequent pattern without candidate generation with different supports. TIARM is capable of handling transaction insertions as well as deletions, and achieves this by using a new data structure called INC-Tree (INCremental Tree), which is mainly intended to improve storage compression of FP-tree.(Figure 2.6), Initially, INC-Tree is empty with a null root node. Then, transactions except the first one are preprocessed by sorting its items according to the item's appearance order in the database and also based on previously inserted transactions. In the next step, when transactions are inserted, the support count of the items is also updated.



TID	Original database	Sorted Database
1	E, D,A,C,B	E, D,A,C,B
2	F,G,B,C	C,B,F,G
3	B,F,D,E	E,D,B,F

Figure 2.6. INC-Tree Construction

After building INC-Tree, new transactions can be added to and deleted from it. Lastly, mining of frequent patterns is done as in FP-growth, with even different support values.

IULFP (Incremental Updating algorithm based on LFP-tree) [21] uses a different data structure known as LFP-tree (Layered Frequent Pattern tree) to maintain frequent itemsets whenever the database is updated. As shown in Figure 2.7, LFP-tree is built by scanning the database and getting 1-itemsets and their frequency, which forms the first level of the tree. Every itemset in each level is represented as 3-tuple < a, v ,t >, where “a” denotes the itemset, “v” denotes its frequency and “t” is a Boolean value which is either 1 or 0 to indicate whether the itemset is frequent or not, respectively. For the case of frequent 2-itemsets, they are generated from level 1 itemsets and linked to level 2 and the remaining frequent k -itemsets are linked to level k

.

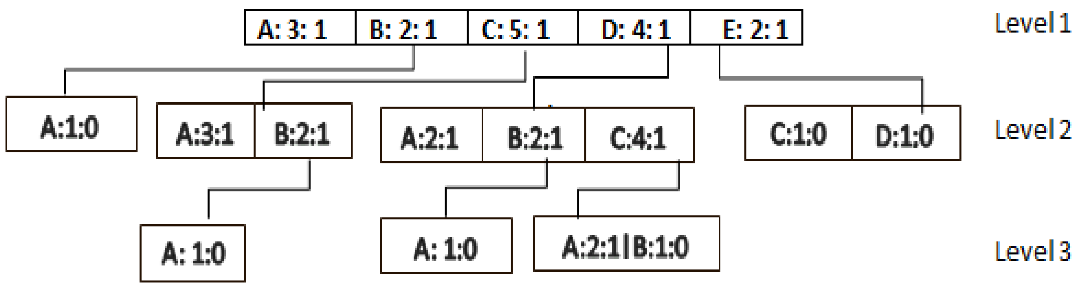


Figure 2.7. LFP- Tree Construction

When transactions arrive, they are scanned once to get the support count of the itemsets. The itemset values for support in level 1 are updated and then determined if they are frequent or not. For the remainder of the levels, potential k-frequent itemsets are determined from the previous level, and later updated in the LFP-tree. Lastly, all frequent patterns are generated by inspecting each conditional pattern in the tree. From the comparison tests in [21], **IULFP** has a mining time of only 69% of FUP

DMA (Dynamic Matrix Apriori) [22, 23] Unlike other dynamic rule mining algorithms that eliminate candidate generation, by representing the signature of database in a tree data structure, DMA uses a matrix and vector as its data structures. From the very beginning, MDA is but built with the dynamic aspect in mind. This involves storing all the itemsets irrespective of their support in the matrix. Since DMA is one of the case study algorithms, a more detailed study is provided in chapter 3.

FUFP-tree (Fast Updated FP-tree) algorithm [24] is constructed in the same way as FP-tree algorithm [12] except that the links between the parent nodes and their child nodes are linked in a bi-directional way to increase the process of item deletion in the maintenance process. Also, the counts of the sorted frequent items are kept in the Header-Table to determine quickly the new frequent items when new transactions arrive. When new transactions arrive, FUFP-tree algorithm processes the items in the transaction into four partitions. That is depending on whether they are large or small in the original database and also in the new increment. By processing each part independently, the Header-Table and the FUFP-tree are continuously updated whenever necessary. The experimental results comparing FUFP-tree algorithm and rerunning FP-growth algorithm from scratch reveals that the FUFP-tree has a better performance in terms of execution time with different incremental sizes than rerunning the FP-growth algorithm from scratch.

Alternative Incremental FP-tree [25] is another algorithm built on the principles of FP-growth algorithm [12]. However, this algorithm ensures that all items, frequent and infrequent ones are map on the tree by assuming a support threshold count of 1. This means that the algorithm is applicable to changing support thresholds. Besides it also handles transaction increments as well as deletions. Experimental finding with different dataset and increment sizes revealed that incremental FP-tree performs better than FP-growth algorithm when rerun from scratch.

The dynamic rule mining algorithms in the literature reviewed are summarized below. Table 2.1 shows their capabilities in terms of increments, deletions, support changes and the methods employed in achieving efficiency.

Table 2.1. Comparison of Dynamic Rule Mining Algorithms

Dynamic Algorithms	**Additions**	**Deletions**	**Support Change**	**Candidate Generation**	**Eliminate candidates by use of:**	
					Tree	**Matrix**
FUP [16]	✓			✓		
FUP2 [17]	✓	✓		✓		
Border [15]	✓	✓	✓	✓		
IARMUPFI [19]	✓			✓		
TIARM [20]	✓	✓	✓		✓	
IULFP [21]	✓				✓	
DMA [23]	✓	✓	✓			✓
FUFP-tree [24]	✓	✓			✓	
Alternative Incremental FP-tree [25]	✓	✓	✓		✓	

In Table 2.1 it is observed many studies have focused on transaction additions. Candidate generations of apriori-like techniques and candidate elimination by tree data structures have dominated the research. And thus more research on other techniques needs further attention.

In the next chapter, a discussion in details of the two dynamic rule mining algorithms that are the focus of the present research is given.

CHAPTER 3

TWO DYNAMIC RULE MINING ALGORITHMS

Related literature on association rule mining has focused on the solution to the problem of finding frequent patterns in massive datasets. The algorithms use different techniques to discover frequent itemsets. However, they all possess a common drawback by assuming the databases are static. This revealed that the obtained rules are only valid for a very short period and requires the algorithms to repeat the entire mining process when the databases are updated.

Furthermore it was argued that in real world scenarios, new data are constantly added to databases, the existing one are modified or deleted as well, which leads to a new challenge of association rule mining process needed to maintain association rules whenever the databases are updated. Dynamic algorithms that have been introduced to deal with this challenge of dynamic rule mining and the algorithms are primarily based on the idea of their parent static algorithms but later modified to work with dynamic rule mining as well.

Research work comparing the performance of static algorithms as well as dynamic/incremental algorithms has been reviewed. Static algorithms compared the performance of the algorithms in terms of execution time with different minimum support thresholds [26]. Dynamic/incremental algorithms, on the other hand, were compared with rerunning the static algorithms for different deletion/addition sizes and different support thresholds, which is basically comparing the dynamic version of the algorithm with rerunning its parent algorithm when databases are updated [22]. Additionally, few comparisons have been made between dynamic algorithms [15]. In cases where comparison is made between algorithms that use candidate generation and those that eliminate the need for candidate generation, non-candidate generation algorithms depended on tree data structures. However, algorithms that use other data structures like a matrix and vectors do exist and yet no comparison work has been done on them.

In this present research therefore, a comparison study based on two dynamic association rule mining algorithms; Dynamic Matrix Apriori (DMA) and Fast Update 2

(FUP2) is presented. The two algorithms use different methods in order to generate frequent itemsets. DMA is a new algorithm that does not depend on candidate generation like FUP2 and also does not use tree data structures, but uses a matrix and a vector to store candidate patterns and their support count respectively. The reason DMA was chosen is that, it is a new algorithm [22] and no performance comparisons have been done yet on it with other dynamic algorithms. Therefore it was deemed necessary to access its strength and weakness with others (FUP2). FUP is the pioneer dynamic algorithm and the most compared algorithm with newly developed ones [15, 19, 21] which is also the reason why it was selected for performance evaluation,

3.1. FUP2 (Fast Update 2)

FUP2 algorithm [17] is an extended version of FUP [16] which is the first incremental rule mining algorithm. In its extension, unlike FUP which handles increments only, FUP2 is capable of maintaining strong association rules when either increments or deletions are made to the database. The algorithm uses a “generate and test” approach of apriori which is basically generating candidate itemsets and testing them if they are frequent.

3.1.1. Additions

For addition cases, the algorithm possesses the same framework as apriori though it combines four key features which distinguish it from apriori thereby achieving greater performance than rerunning apriori algorithm when new transactions arrive. To explain FUP2 for additions, the terms in Table 3.1 are used

Table 3.1. Key Terms used for Addition and Deletion

Term	Meaning
DB	Original database
Db	Increment database
D	Size of **db**
UD	Updated database
L	Frequent itemsets in **DB**
T	Transaction
P	Optimization
L'	Frequent itemsets in **UD**
W	Winners
C	Candidate set
$\mathbf{d}^{-}$	Deleted transactions.
S	Minimum support
$\mathbf{Support}_{\mathbf{d-}}$	Support in deleted dataset
$\mathbf{Support}_{\mathbf{d}}$	Support in increment dataset
$\mathbf{Support}_{\mathbf{UD}}$	Support in updated database
$\mathbf{Support}_{\mathbf{DB}}$	Support in Original database
K= {1, 2, 3…}	Integer value

When new transactions arrive, new frequent itemsets are determined according to Figure 3.1.

```
INPUT:DB, $L_k$, d, s
      OUTPUT:L'
      For 1st iteration;
1.  W=$L_1$; C= $\emptyset$; $L'_1$= $\emptyset$; P= $\emptyset$;
2.  For each T $\in$ db do
3.    For each 1-itemset X $\subseteq$ T do
4.     If X $\in$ W then X.$support_d$ ++;
5.     Else
6.     If X $\notin$ C then  C= C {X}; X.$support_d$ =0;
7.     X.$support_d$ ++;
8.         End do
9.  End do
10.   For each X $\in$ W do
11. If X.$support_{UD}$ $\geq$ (D+d) //Put winners in $L'_1$
12. End do
13.   For each X $\in$ C //prune candidate sets in C
14. If X.$support_d$ < s*d
15.   Then  C= C- {X}; P= P $\cup${X};
16.    For each T $\in$ DB do// scan DB
17.   For each 1-itemset X $\subseteq$ T do
18.     If X $\in$ C then X.$support_D$ ++;
19.     If X $\in$ P then remove X from T
20.   End do
21. End do
22. For each X $\in$ C do
23. If X.$support_{UD}$ $\geq$ S*(D+d)// put winners in $L'_1$
24. End do
25. Return $L'_1$
    For Kth iteration
26. If ($L'_k$ $\neq$ $\emptyset$)do
27. K++;
28.  W =$L_k$ ; $L'_k$ = $\emptyset$;
29.  C = apriori-gen($L'_{k-1}$)-$L_k$;
30.  For each k-itemset X $\in$ W do//prune  W
31.   For each(k-1)-itemset Y $\in$ $L_{k-1}$ -$L'_{k-1}$ do
32.    If Y $\subseteq$ X then { W= W - {x}; beak;}
33. For each T $\in$ db do{//scan db
34.   For each X $\in$ Subset(W,T)do
35.     X.$support_d$ ++;
36.   For each X $\in$ Subset(C,T)do
37.    X.$support_d$ ++;
38.    Reduce_db (T);
39. For all X $\in$ W do
40.   If X.$support_{UD}$ $\geq$ S*(D+d)// put winners in $L'_k$
41. For all X $\in$ C do //prune candidate sets in C
42.  If X.$support_d$ < s*d then C= C -{X};
43. For all T $\in$ DB do// scan DB
44.   For each X $\in$ Subset(C,T)do
45.   X.$support_D$++;
46.   Reduce_DB(T);
47. For each X $\in$ C do
48.    If X.$support_{UD}$ $\geq$ S*(D+d)// put winners in $L'_k$
49.   End do
50. End do
51. Return $L'_k$
```

Figure 3.1. Addition Process for FUP2 Additions

In Figure 3.1 above, the new frequent itemsets are determined in two phases. In the first phase, in each of the iterations, frequent itemsets of size L_k in *DB* are checked against *db* by scanning only *db*. In so doing, the algorithm is capable of filtering out losers (itemsets in L_k whose support has dropped below minimum threshold in *UD*). The ones that remain are added to L'_k.. In the second phase, during the scanning of *db*, a candidate set C_k is extracted together with their $Support_d$. Then $Support_d$ of itemsets in C_k is updated against *DB*. If they turn out to be frequent, then they are also added to L'_k.

It is important to note that although FUP [16] assumes this set to be small, in cases where the incremental databases are large as is the case with the present experiments, this candidate set can be large and hence a performance cost occurs in generating and finding new frequent itemsets. To further understand how FUP2 performs, an example is provided in Figure 3.2.

Original Database (DB)
A, B ,C, D, E, F
A, B, E
A, C, E
C
D, E

Minimum support = 60%

Incremental (d)
A, E, F
A, D, E

$\boldsymbol{Frequent\ itemsets(L)}$	$\boldsymbol{A}$	$\boldsymbol{C}$	$\boldsymbol{E}$	$\boldsymbol{A,E}$
$\boldsymbol{Support}(\%)$	60	60	80	60

<u>In the 1st iteration</u>

1. After scanning $\boldsymbol{d}$,
   - ➢ $\boldsymbol{Support}_{UD}$ of L1={ A,C,E} is updated
   - ➢ Itemset $\boldsymbol{C}$ becomes a loser with $\boldsymbol{Support}_{UD}$ = **42.86**
   - ➢ Itemsets $\boldsymbol{A}$ *and* $\boldsymbol{E}$ become winners with $\boldsymbol{Support}_{UD}$ = $\mathbf{71.43\%}$ *and* $\mathbf{85.71\%}$ respectively.
2. Candidates $\boldsymbol{D}$ *and* $\boldsymbol{F}$ are obtained after scan on $\boldsymbol{d}$.
   - ➢ $\boldsymbol{Support}_{d}$ of both **D** and **F** is **50%**.Therefore they are losers
3. Only itemsets $\boldsymbol{A}$ *and* $\boldsymbol{E}$ are added to $\boldsymbol{L'1}$

<u>In the 2nd iteration</u>

4. Again scan d.

➢ **L2 = {A, E}** $\boldsymbol{Support}_{UD}$ becomes **71.43%.**

5. No need to check candidates from $\boldsymbol{L'1}$ as it has been handled and the algorithm ends.

$\boldsymbol{New\ frequent\ itemsets(L')}$	$\boldsymbol{A}$	$\boldsymbol{E}$	$\boldsymbol{A,E}$
$\boldsymbol{Support}(\%)$	71.43	85.71	71.43

Figure 3.2. Incremental Example for Addition –FUP

3.1.2. Deletions

FUP2 has two cases of deletions. The first deletion, considers a special case of transaction deletion only. The second deals with a general case of transaction deletion and insertion. In the experiments the former was considered as it is related to the comparison of DMA Algorithm for deletion cases (deletion only)

Deletions in FUP2 are also done iteratively and $candidate - gen$ function of apriori [14] is employed for the candidate generation. The same terms displayed in Table3.1 are used to explain the deletion process of the algorithm as well.

When deletion updates are made to the database, FUP2 goes through the processes below to determine new frequent itemsets (Figure 3.3).

```
INPUT: $D$, $L$, $d^-$, $s$, $C_1$
OUTPUT: $L'$
1. Obtain candidate set $C_k$. Halt if $C_k = \emptyset$
2. Partition $C_k$ into $P_k$ and $Q_k$
   $P_k = C_k \cap L_k$;
   $Q_k = C_k - P_k$ ;
3. Scan $d^-$ to find $Support_{d-}$ for each $X \in P_k \cup Q_k$
4. For each $X \in P_k$, Calculate $\sigma'_X$
5. Delete from $Q_k$ those candidates X
   where $Support_{d-} \geq |d^-| * s\%$
6. Scan $D^-$ to find out $Support_{UD}$ of the remaining
   Candidate $X \in Q_k$
7. Add to $L'_k$ those candidates X from $P_k \cup Q_k$
        for which $Support_{UD} \geq |D^-| * s\%$
8. Halt if $|L'_k| < k+1$
```

Figure 3.3. FUP2 Process for Only Deletions

During FUP2 deletion process, first the FUP2 algorithm takes in C_k and L_k then divides the two sets into two partitions, P_k and Q_k. P_k is a set of frequent itemsets in C_k

while Q_{k} is a set of remaining infrequent itemsets in C_{k}. That is, $P_{\mathrm{k}} = C_{\mathrm{k}} \cap L_{\mathrm{k}}$ and $Q_{\mathrm{k}} = C_{\mathrm{k}} - P_{\mathrm{k}}$. Next, d^{-} is scanned to determine $\boldsymbol{Support}_{\mathrm{d-}}$ for itemsets in P_{k} and Q_{k}.

Then, for each itemset in P_{k}, FUP2 get their $Support_{UD}$ by deducting $\boldsymbol{Support}_{\mathrm{d-}}$ from $\boldsymbol{Support}_{\mathrm{DB}}$. If their $Support_{UD} \geq S$, they are added to L' as new frequent itemsets. Afterwards the itemsets in Q_{k} that were non-frequent are checked if they can be potential frequent itemsets, this is done by finding itemsets whose $\boldsymbol{Support}_{\mathrm{d-}} \geq s$ and deleting them from Q_{k} as they can never be frequent in UD . For the remaining itemsets in $Q_{\mathrm{k},}$ they are scanned on the UD, and if they are frequent, then they are added to L' . Lastly, the process is repeated until $C_{\mathrm{k}}\ is\ empty$. It is important to note that a set of candidates in Q_{k} that is small in the deleted section of the database is kept and used as an optimization in later iterations to reduce the size of $Q_{\mathrm{k+1}}$

The example in Figure 3.4 below further explains how FUP2 works for deletion cases only. Transactions are deleted from the same original database DB used in Figure 3.2 above.

Updated Database D^1
A, B, E
A, C, E
D, E

Minimum support = 60%

Candidate set, $C1 = \{ A, B, C, D, E, F \}$

$Large(L)$	A	C	E	A, E
$Support(\%)$	60	60	80	60

Deleted dataset (d^-)
A, B, C, D, E, F
C

- ❖ We have $P1 = \{A, C, E\}$ and $Q1 = \{B, D, F\}$
- ❖ After scanning d^-, support of itemsets in P_1 becomes $66.67\%, 33.33\%$ and 100% respectively. C is filtered out.
- ❖ **For** $\forall X \in Q1$ Have $Support_d = 50\%$. Therefore small in d^- .meaning they could be frequent in the UD.
- ❖ After scanning UD all itemsets in $Q1$ are still small and are filtered out. $New\ L'1 = \{A, E\}$
- ❖ Next candidate set $C2 = \{AE\}$ is generated from $L'1$,
- ❖ Updates support of itemset AE to 66.67%

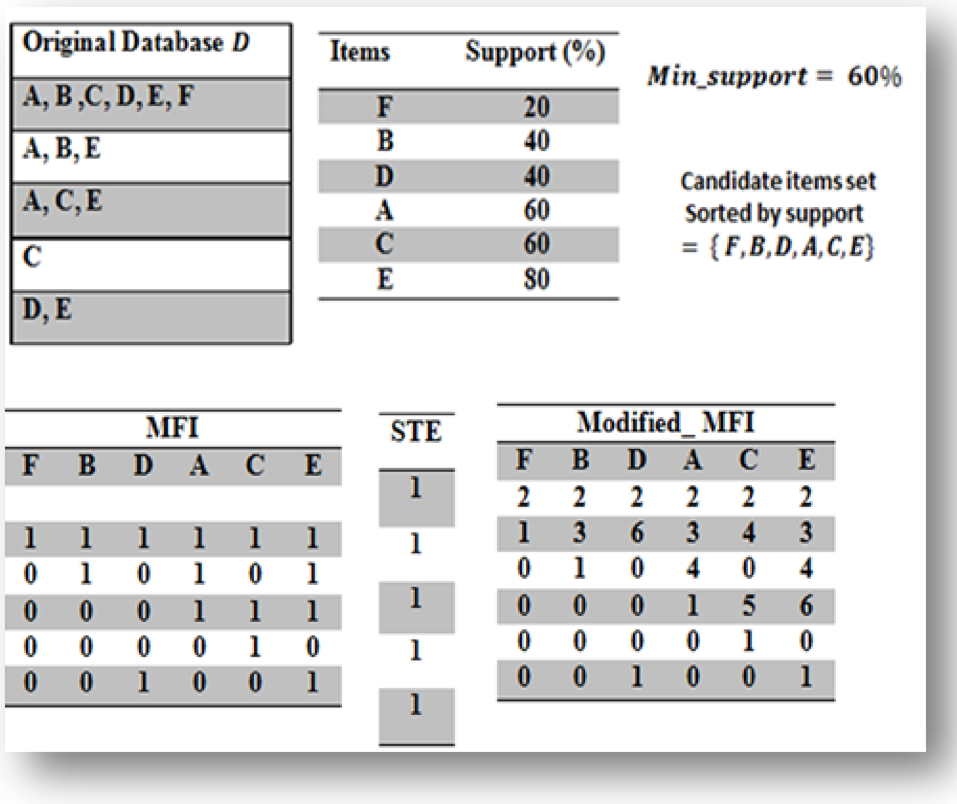
$New\ Large(L')$	A	E	A, E
$Support(\%)$	66.67	100	66.67

Figure 3.4. Deletion Example – FUP2

3.2. DMA (Dynamic Matrix Apriori)

DMA [23] is a current new dynamic rule mining algorithm. Unlike FUP2 which employs the framework of apriori in generating frequent itemsets, DMA uses a matrix structure and a vector to generate frequent patterns. It is also basically built on the framework of the parent algorithm Matrix apriori.

The experiments showed that FUP2 is similar to apriori just before the database is updated. It was observed that FUP2 depends mainly on the frequent itemsets which have already been discovered to speed up the whole dynamic mining process. Unlike FUP2, DMA uses a matrix called MFI (Matrix of frequent Itemsets) to store candidate sets and a vector known as STE to store the support of the candidates. Additionally, MFI in DMA assumes the minimum itemset support count as 1, meaning that all items are stored in MFI as displayed in the example in Figure 3.5.



Original Database *D*
A, B ,C, D, E, F
A, B, E
A, C, E
C
D, E

Items	Support (%)
F	20
B	40
D	40
A	60
C	60
E	80

Min_support = 60%

Candidate items set Sorted by support = {*F*, *B*, *D*, *A*, *C*, *E*}

MFI

F	B	D	A	C	E
1	1	1	1	1	1
0	1	0	1	0	1
0	0	0	1	1	1
0	0	0	0	1	0
0	0	1	0	0	1

STE
1
1
1
1
1

Modified_ MFI

F	B	D	A	C	E
2	2	2	2	2	2
1	3	6	3	4	3
0	1	0	4	0	4
0	0	0	1	5	6
0	0	0	0	1	0
0	0	1	0	0	1

Figure 3.5. Example for DMA before Updates

In Figure 3.5, before updates arrive, D is scanned to determine the support count of all items in the database. Next, the items are sorted in an ascending order in accordance with their support count. Subsequently, in the second scan of D, for each transaction in the database, items in the transaction are identified as sorted by ascending support. This implies that each resulting set is a candidate set. Then, for the transaction being processed its candidate is represented in the MFI and STE as follows.

The columns of MFI represent the items while the rows represent the candidate sets. Now, starting from the second row of MFI, if an item is present in the candidate set its column value is set to 1 and if not present in the column, the value is set to 0. Subsequently the support of the candidate set in STE is set 1. On the other hand, if the candidate set is already represented in the MFI. (Exists in MFI) it is not added as a new row of MFI rather its value in STE is incremented by 1.

After construction of the MFI, the next step is the modification of the MFI to support indexing. This modification helps in speeding up the search for frequent patterns and is done as follows. First, starting with the first row of MFI that was left empty; each cell in the first row is updated with the row number of the cell where the first candidate item of that column appears for the first time. For remaining rows except the last, each of the 1s is replaced with the next row number of the cell where the candidate item is present.

3.2.1. Additions

When new transactions arrive, DMA follows the steps in the pseudo code to generate new frequent patterns. (Figure3.4)

```
INPUT:MFI, STE, increments db, 1-itemset ordered list IS,
      minimum support μ
     OUTPUT:MFI, STE, 1-item ordered list IS_new , Frequent
patterns F
     BEGIN
  1. Scan db
  2. Create IS_new from IS using db
  3. FOR each new item in IS_new
  4. Add a column to MFI
  5. END FOR
  6. FOR each transaction in db
  7.    IF the transaction exists in MFI
  8.       Increment its support in STE by 1
  9.    ELSE
  10.      Add a new line to MFI
  11.      Set the STE of the new line to 1
  12. END FOR
  13. Update and reorder MFI using IS_new
  14. Generate F from MFI
  15. Return IS_new, MFI, STE, F
     END
```

Figure 3.6. Increment Process for DMA

In the first scan when *db* arrives, it is a scanned to identify the items and their support count. Then, $IS_{\mathbf{new}}$ is generated to handle new support changes. Next, it is determined whether there are any new items whose columns should be included in MFI (steps 1 to 5).

In the second scan, each transaction in the increment is checked against MFI. If the transaction exists then its support is incremented by 1 in STE, if not it is added to MFI as a new row (Steps 6 to 12).

Then, the rows are updated to enable indexing and the columns are reordered according to new $IS_{\mathbf{new}}$**.** Lastly, the new frequent patterns are generated from MFI.

For more clarification an example to show the increment process of DMA is provided in Figure 3.5.

Incremental db
A, E, F
A, D, E

Items	Support (%)
B	28.57
F	28.57
C	42.86
D	42.86
A	71.43
E	85.71

$Min\_support = 60\%$

New Candidate items set Sorted by support
$= ~~\{F, B, D, A, C, E\}~~$
$= \{B, F, C, D, A, E,\}$

New Modified_ MFI					
B	F	C	D	A	E
2	2	2	2	2	2
3	7	4	6	3	3
1	0	0	0	4	4
0	0	5	0	7	6
0	0	1	0	0	0
0	0	0	8	0	7
0	1	0	0	8	8
0	0	0	1	1	1

STE
1
1
1
1
1
1
1

Frequent Patterns	Support (%)
E	85.71
A	71.43
A, E	71.43

Figure 3.7. An Example for Addition – DMA

3.2.2. Deletions

The pseudo code in Figure illustrates the deletion process for DMA

```
INPUT:MFI, STE, deletions d-,1-itemset ordered list IS,
minimum support μ
    OUTPUT:MFI, STE, 1-item ordered list $IS_{new}$ , Frequent
patterns F
    BEGIN
  1. Scan d¯
  2. Create $IS_{new}$ from IS using d-
  3. FOR each transaction in d-
  4.       Decrement its support in STE by 1
  5. END FOR
  6. Update and reorder MFI using $IS_{new}$
  7. Generate F from MFI
  8. Return $IS_{new}$, MFI, STE, F
     END
```

Figure 3.8. Deletion Process for DMA

For deletion cases, when transactions are deleted from original database, DMA makes a first scan on $d-$ and identifies the deleted items and their support count. The supports of deleted items in the transactions are then deducted from their previously known supports to form a new $IS_{\mathbf{new}}$ (Step 1 to 2).

In the second scan, each deleted transaction is checked against MFI to locate its position, which helps to identify its location in STE which is decreased by 1 (step 3 to 5). Next, the columns are reordered according to new $IS_{\mathbf{new}}$. Lastly new frequent patterns are generated from MFI.

For more clarification an example to show how deletion process is implemented for DMA is provided in Figure 3.7.

Deleted Dataset
A, B, C, D, E, F
C

Items	Support (%)
F	0.0
B	33.33
C	33.33
D	33.33
A	66.66
E	100.0

$Min\_support = 60\%$

New Candidate items set Sorted by support $= \{F, B, D, A, C, E\}$ $= \{F, B, C, D, A, E, \}$

New Modified_ MFI						
B	F	C	D	A	E	STE
2	2	2	2	2	2	0
3	1	4	6	3	3	1
1	0	0	0	4	4	1
0	0	5	0	1	6	0
0	0	1	0	0	0	1
0	0	0	1	0	1	

Frequent Patterns	Support (%)
E	100
A	66.67
A, E	66.67

Figure 3.9. An Example for Deletion – DMA

In summary, this chapter discussed in details the two dynamic association rule mining algorithms; FUP2 and DMA. By giving pseudo codes and examples showed how FUP2 an apriori-like algorithm handles both increments and deletions done to the database. The same process was conducted for DMA algorithm which eliminated the need for candidate generation by using a matrix data structure.

In the next chapter, the experimental findings of FUP2 and DMA are discussed. The algorithms are tested with three different dataset for their execution times with; increments of different sizes, deletions of different sizes and varying support thresholds. The performance results are displayed graphically.

CHAPTER 4

PERFORMANCE EVALUATION

In this section, the evaluation of the performance of two algorithms, Dynamic Matrix Apriori (DMA) [23] and Fast Update2 (FUP2) [17] is done. FUP2 has been chosen among the incremental algorithms because it possesses the pioneer idea of dynamic association rule mining. It is applicable to both transaction increments and deletions in large databases. DMA, a new algorithm, also possesses FUP2 capabilities and minimum support changes. Therefore, in the evaluation of the two algorithms three scenarios are considered: (1) transaction increments, here new transactions are added to the original databases in increments of different sizes, (2) transaction deletions, again, transactions of different sizes are deleted from the original database, (3) varying minimum support threshold, the increment size of transactions is kept constant then the algorithms are executed for changing support thresholds. All the three scenarios are tested on three different datasets.

4.1. Simulation Environment

To conduct the experiments, both algorithms are coded on a Visual Studio.Net environment in C# (Appendix A) and run on Intel® Core™ i5-2430M CPU @2.40GHZ processor computer with installed memory of 6GB. Then, to determine the accuracy and dependability of the results, experiments are run four times. It is ensured that application processes that use system resources are disabled during the testing of the algorithms.

The data used in the experiments are generated with a synthetic data generator called ARtool [30]. ARtool generator was written in java. It accepts input parameters from the user for example number of transactions, number of items, average size of transactions, etc. it then uses these parameters to generate the datasets. It should be noted that in ARtool, the number of items in generated datasets is not usually equal to the parameter given for items. Instead, ARtool uses random exponential distribution to generate transactions. Therefore, if the generated datasets are not checked, it may affect the interpretation of the results especially if a large number of items size is given in the items parameter and it turns out be small after the generation of the datasets.

In the present study, the parameters given to ARtool are displayed in Table4.1 to generate three different datasets; the first one is Dataset1 which is thought to have longer patterns and low diversity of items. The second is Dataset2 with short patterns and thought to have high diversity of items and then Dataset3 with parameters more than dataset1 for number of items parameter and less for the average size of the transaction patterns.

However it is important to note that, after testing the generated datasets for its parameters, it turns out with real parameters shown in Table 4.2 for Dataset1, Dataset2 and Dataset3, where the diversity of items has changed in all three datasets Henceforth our finding in the study will be based on those new found parameters.

Table 4.1. Parameters given to ARtool

Datasets	**Dataset1**	**Dataset2**	**Dataset3**
Number of transactions	15000	15000	15000
Number of items	10000	30000	22000
Average size of transactions	20	20	20
Average size of the patterns	10	5	7

Table 4.2. Real Parameters of Datasets after Running Tests

Datasets	**Dataset1**	**Dataset2**	**Dataset3**
Number of transactions	15000	15000	15000
Number of items	354	157	251
Average size of transactions	20	20	20
Average size of the patterns	10	5	7

4.2. Additions

Each of the two algorithms, FUP2 and DMA are run thrice with three different datasets. New transactions are added in increments of 5% to 100% of the original dataset sizes. Then, the execution time with respect to different incremental sizes are plotted. It is important to note that the minimum support threshold is kept constant at 10% for the experiments conducted.

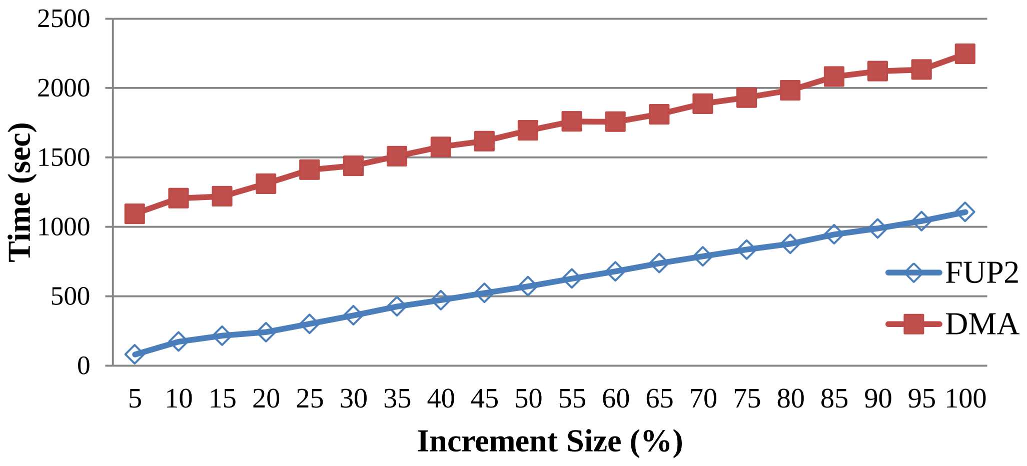


Figure 4.1. Time for Addition – Dataset1

In Figure 4.1 it is displayed that FUP2 performs better than DMA. FUP2 performs approximately 3 times (exact value is 2.817) faster for Dataset1. When the order of frequency of items changes after increments, then, DMA requires an update of the columns of the matrix which is large (354 columns) in addition to searching the matrix whose rows are high due to the pattern sizes of the transaction and hence a performance cost

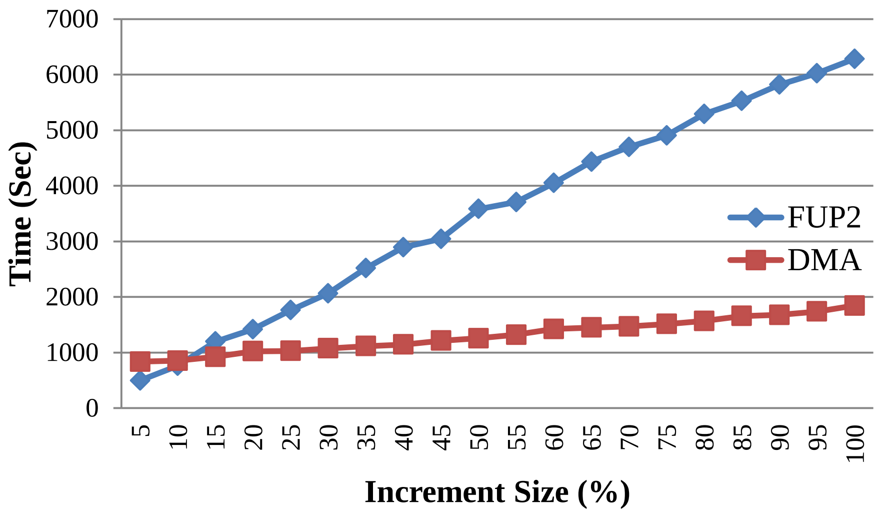


Figure 4.2. Time for Addition – Dataset2

In Figure 4.2 above, two situations are observed. Firstly, for incremental sizes of 10% and below of the original database, FUP2 executes faster than DMA. This is because of the diversity of item which turned out to be low (175 items) for the dataset2. This means that for small increments, the size of items is even reduced more allowing FUP2 to perform faster. Secondly, for increment sizes above 10% DMA generally executes approximately 3 times (exact value 2.83) faster than FUP2. It is observed that there is a steep increase in the execution time of FUP2 which mainly caused by the emergence of new items and hence more time generating candidate sets as opposed to DMA whose matrix for Dataset2 is smaller due to its small transaction patterns and therefore easier to maintain.

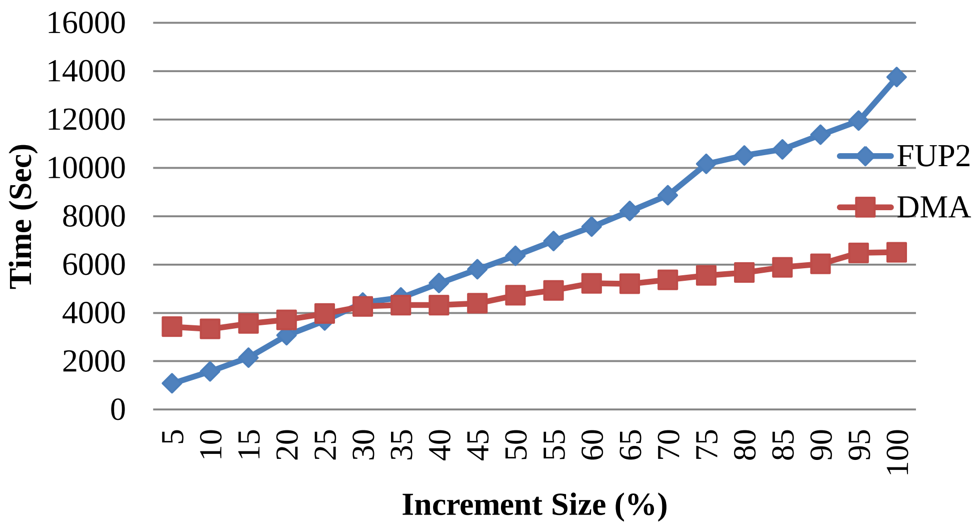


Figure 4.3. Time for Addition – Dataset3

Again, in Figure 4.3 it is observed that FUP2 out performs DMA for increment below 30%. Increasing the items and the pattern of transactions slightly above those of Dataset2 shows a general increase in the execution times for increments.

4.3. Deletions

The algorithms are also tested for deletions scenarios. Once more, transactions are deleted from the Datasets 1, 2 and 3, and their execution time with respect to the deletion sizes of 5% to 50% of the original Database size are plotted in Figure 4.3, Figure 4.4 and Figure 4.5 respectively.

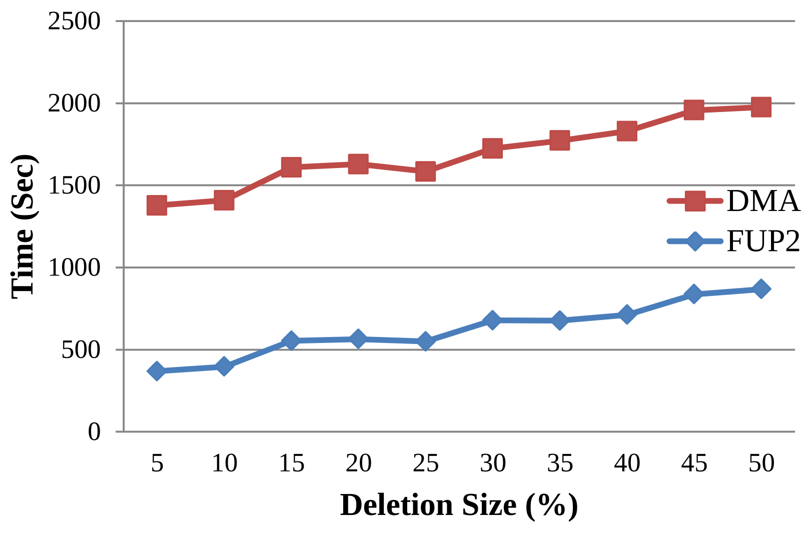


Figure 4.4. Time for Deletion – Dataset1

Deleting from Dataset1, FUP2 performs 1.7 times faster than DMA. With high many transaction patterns in dataset1, after a deletion update FUP2 performs faster than updating the DMA's matrix.

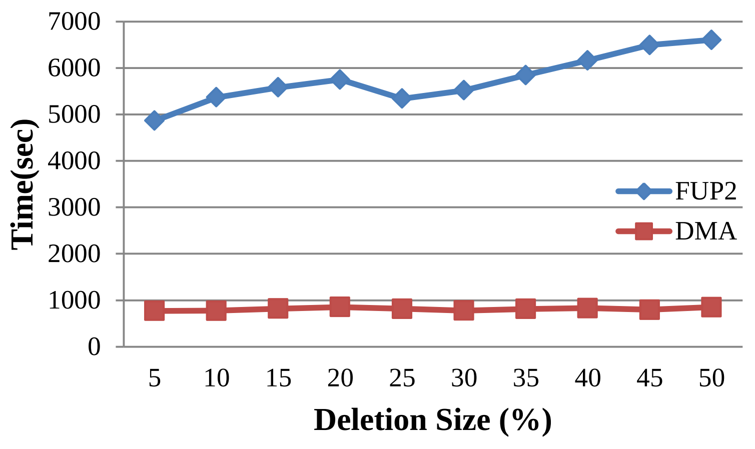


Figure 4.5. Execution Time for Deletion – Dataset2

Deleting from Dataset2 shows an increase in the performance of DMA. Experiments show that DMA performs 7 times better than FUP2. It is observed in Figure 4.5 that DMA has almost a constant execution time, which is due to the size of the matrix that is relatively small due to small transaction patterns.

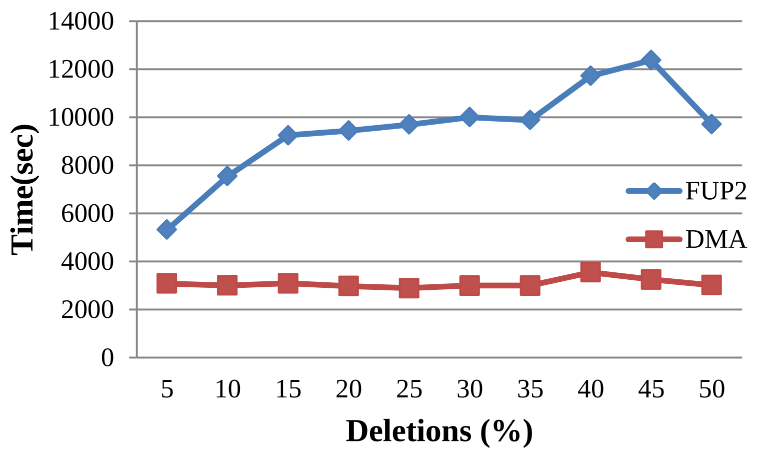


Figure 4.6. Execution Time for Deletion – Dataset3

In Figure 4.6, DMA out Performs FUP2. After increasing the items and the pattern of transactions slightly above those of Dataset2 shows an increase in the execution times for deletions with Dataset3

4.4. Changing Minimum Support

Lastly, the algorithms are compared with varying minimum support threshold. In this case, the incremental dataset size is kept constant at 5% of the original database. Then the minimum support is varies from 2.5% to 20%, that is, in increments of 2.5. Figures 4.7, 4.8 and 4.9 show the execution time for Datasets 1, 2 and 3 respectively.

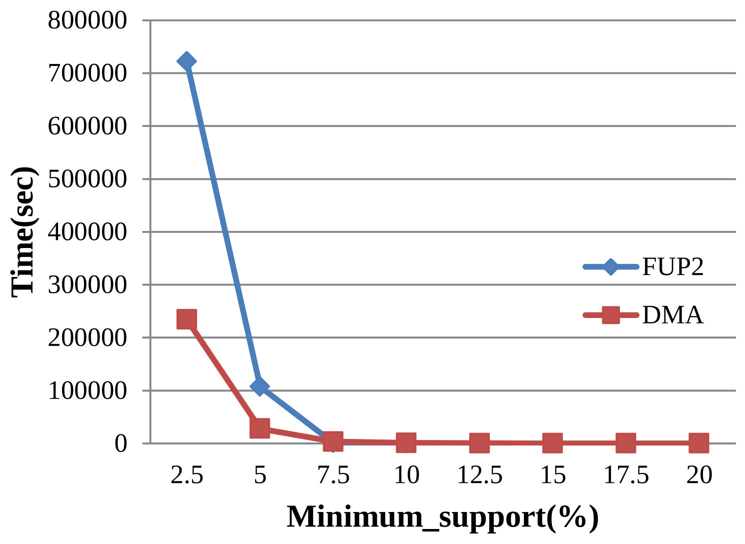


Figure 4.7. Execution Time for Support Changes – Dataset1

Figure 4.5 shows that, DMA has a better performance when the support is kept below 7.5%. With support above the algorithms have a relatively equal execution time. At low minimum support values candidates itemset generated by FUP2 are very many and hence require more execution time.

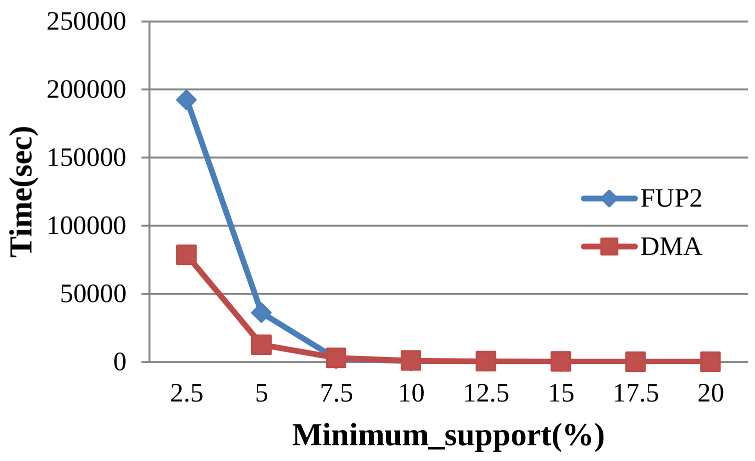


Figure 4.8. Execution Time for Support Change – Dataset2

For Dataset2, we still observe that DMA has a better performance over FUP2 for support values of 7.5 and below. For higher support values, the execution time is linear for both algorithms.

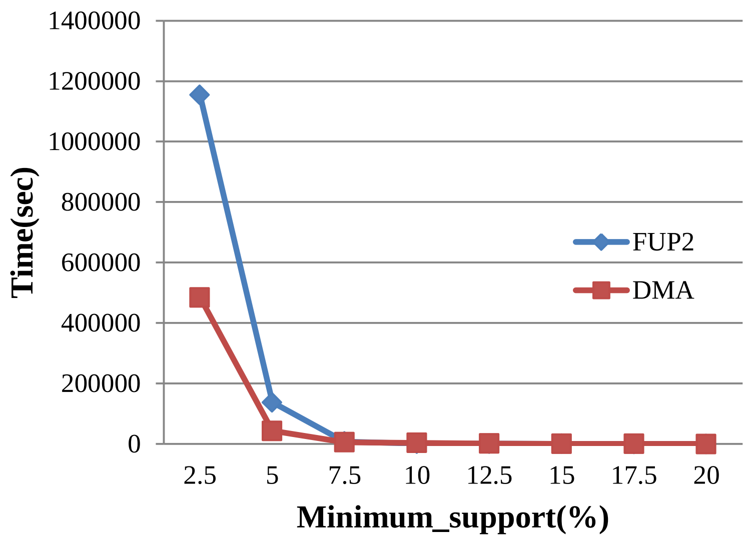


Figure 4.9. Execution Time for Support Change – Dataset3

For Dataset3 in Figure4.9, it is also seen that DMA has a better performance over FUP2 for support values of 7.5 and below. For higher support values, the execution time seem linear for both algorithms with FUP2 doing better at higher support thresholds.

4.5. Discussion on Results

In the research, the performance comparison of two dynamic association rule mining algorithms that use different techniques, namely FUP2 and DMA was compared. The algorithms are tested for their execution time when; 1) new transaction arrive, 2) existing transactions are deleted and 3) minimum support is changed. The three different datasets were used in our experiments.

The findings show that, for additions and deletions the performance of the algorithms is mainly determined by the transaction patterns in the dataset, the number of items and the set minimum support threshold. Another finding is that DMA performs better with datasets 2 and 3, then FUP2 is better with dataset1 when the minimum support is kept constant at 10%/ This is mainly due to the difference in the transaction patterns of the datasets.

Furthermore, when minimum support threshold is varied, DMA performs better than FUP2 for low supports (7.5 and below) for a constant increment of 5% for both datasets 1, 2 and 3. This is due to the main drawback of Apriori-like algorithms (FUP2) is candidate generation especially when support values are small.

Finally in the next chapter, a conclusion of the thesis and suggestions for future are provided

CHAPTER 5

CONCLUSION

Research in data mining has mainly focused on association rule mining due to its wide applicability in many areas. Association rule mining aims to discover strong association among the data stored in databases. Many algorithms such as Apriori, FP-growth, and Matrix apriori have been developed to mine interesting association rules in database. In today's databases, there is a constant modification of the databases with new data and possibly deletion of the existing one, which implicates the need for algorithms to handle databases of this nature. Many dynamic algorithms have been introduced to do mining on constantly updated databases, some of which use candidate-generation techniques while the others eliminate candidate generation by creating the signature of the database on other data structures like trees and matrices.

In this present study, a performance comparison study was made between two the dynamic rule mining algorithms; FUP2 and DMA. FUP2 is a candidate generation algorithm while DMA eliminates candidate generation by storing all candidate patterns in matrix and their support counts in vector. It is significant to note that the two algorithms have not been compared previously and yet they use different techniques to generate frequent patterns, this motivated the current study.

The performances of the two algorithms on increments, deletions, and support changes were experimented on three datasets which have different characteristics.

The findings revealed that for additions and deletions the performance of the algorithms was mainly determined by the transaction patterns sizes of the datasets used number of items and the set minimum support threshold. Another important finding of the study was that generally DMA performed better with datasets 2 and 3 and so was FUP2 with dataset1 when the minimum support is kept constant at 10% with increments. For deletions, DMA performed 7 times better than FUP2 for dataset2 and yet again for dataset1 FUP2 performed 1.7 times better than DMA.

Another significant finding was that when the minimum support threshold is varied, DMA performed better than FUP2 for low supports (7.5 and below) for a constant increment of 5% for both dataset1 and dataset2. This is due to the main drawback of candidate generation especially when support values are small.

Therefore, from this study it can be revealed that in order to get the best out the two algorithms, it is important to first study the nature of the dataset before deciding on the algorithm to use. However, it is important to note that, when the minimum support threshold is kept low DMA will always be a better choice over FUP2. Since FUP2 will take more time generating more candidates as support being low.

Future work should consider investigating the performance of the algorithms on real life datasets. This is partly because synthetically generated dataset have been found to be misleading and may affect analysis of the results if the generated dataset is not independently analyzed.

The present study focused on two dynamic algorithms FUP2 and DMA. However, in order to have a broad understanding of the impact of datasets on dynamic algorithms other types of algorithms should also be investigated, which can be regarded as a suggestion for future work.

REFERENCES

1. Chen. M.S. and Han. J. (1996) “Data mining: An overview from a database perspective. IEEE Transaction on knowledge and Data Engineering” pp. 866-883.

2. Mannila, H., Toivonen, H., and Verkamo, A.I. (1994). Efficient algorithms for discovering association rules. In Proc.AAAI’94 Workshop Knowledge Discovery in Databases (KDD’94), Seattle, WA, pp. 181–192.

3. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A.I. (1996). Fast discovery of association rules. In Advances in Knowledge Discovery and Data Mining, U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), AAAI/MIT Press, pp. 307–328.

4. Savasere, A., Omiecinski, E., and Navathe, S. (1995). An efficient algorithm for mining association rules in large databases. In Proc. 1995 Int. Conf. Very Large Data Bases (VLDB’95), Zurich, Switzerland, pp. 432–443.

5. Park, J.S., Chen, M.S., and Yu, P.S. (1995). An effective hash-based algorithm for mining association rules. In Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD’95), San Jose, CA, pp. 175–186.

6. Lent, B., Swami, A., andWidom, J. (1997). Clustering association rules. In Proc. 1997 Int. Conf. Data Engineering (ICDE’97), Birmingham, England, pp. 220–231.

7. Sarawagi, S., Thomas, S., and Agrawal, R. (1998). Integrating association rule mining with relational database systems: Alternatives and implications. In Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD’98), Seattle, WA, pp. 343–354.

8. Srikant, R., Vu, Q., and Agrawal, R. (1997). Mining association rules with item constraints. In Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD’97), Newport Beach, CA, pp. 67–73.

9. Lakshmanan R., L.V.S., Han, J., and Pang, A. (1998). Exploratory mining and pruning optimizations of constrained associations rules. In Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98), Seattle, WA, pp. 13–24.

10. Grahne, G., Lakshmanan, L., and Wang, X. (2000). Efficient mining of constrained correlated sets. In Proc. 2000 Int. Conf. Data Engineering (ICDE'00), San Diego, CA, pp. 512–521

11. Pav´on, J., S. Viana, and S. G´omez (2006). Matrix apriori: Speeding up the search for frequent patterns. In Proceedings of the 24th IASTED International Conference on Database and Applications, DBA'06, Anaheim, CA, USA, pp. 75–82. ACTA Press.

12. Han, J., J. Pei, and Y. Yin (2000). Mining frequent patterns without candidate generation. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, New York, NY, USA, pp. 1–12. ACM.

13. Agarwal R, Imielinski T and Swami A (1993) "Mining Association Rules between Sets of Items in Large Databases", ACM SIGMOD International Conference on Management of Data, May.

14. Agarwal R and Shrikant R (1994) "Fast Algorithms for Mining Association Rules in Large Databases", Proceedings of 20th International Conference on Very Large Databases, August-September , Santiago, Chile,

15. Feldman R, Aumann Y and Lipshtat O (1999) "Borders: An Efficient Algorithm for Association Generation in Dynamic Databases", Journal of Intelligent Information System, pp; 61–73.

16. Cheung D W, Han J, Ng V T and Wong C Y (1996) "Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique", 12th International Conference on Data Engineering, New Orleans, Louisiana.

17. Cheung D W, Lee D W, and Kao S D (1997). “A General Incremental Technique for Maintaining Discovered Association Rules”, In Proceedings of the fifth International Conference on Database System for Advanced Applications, Melbourne, Australia.

18. Mannila H. and Toivonen H (1997). Level wise Search and Borders of Theories in Knowledge Discovery, Data Mining and Knowledge Discovery, pp. 241–258.

19. Amornchewin R. and Kreesuradej W, (2007) “Incremental association rule mining using promising frequent itemset algorithm”, In Proceeding 6th International Conference on Information, Communications and Signal Processing

20. Pradeepini, G., Jyothi, S. (2010) “Tree-based incremental association rule mining without candidate itemset generation”. In: Trendz in Information Sciences & Computing (TISC), pp. 78–81. IEEE Computer Society.

21. Tongyan Li, Xingming Li. (2010) IULFP: An efficient incremental updating algorithm based on LFP-tree for mining association rules. In Inter. Conference on Computer Application and System Modeling – ICCASM.

22. Oguz, D. and Ergenc B. (2012). Incremental itemset mining based on matrix 45priori algorithm. In 14th International Conference, DaWaK 2012. Springer. Accepted.

23. Oguz, D (2012) Dynamic frequent itemset mining based on matrix 45priori algorithm. Master’s thesis submitted to the Graduate School of Engineering and Sciences of˙Izmir Institute of Technology, izmir-Turkey

24. Hong, T.-P., C.-W. Lin, and Y.-L.Wu (2008). Incrementally fast updated frequent patterntrees. Expert Syst. Appl. pp. 2424–2435.

25. Muhaimenul, R. Alhajj, and K. Barker (2008). Alternative method for incrementally constructing the fp-tree. In P. Chountas, I. Petrounias, and J. Kacprzyk (Eds.), Intelligent Techniques and Tools for Novel System Architectures, Volume 109 of Studies in Computational Intelligence, pp. 361–377. Springer Berlin Heidelberg

26. Yildiz, B. and B. Ergenc (2010). Comparison of two association rule mining algorithms without candidate generation. In Proceedings of the 10th IASTED International Conference on Artificial Intelligence and Applications.

27. Shan, S., Wang, X., Sui, M.: (2010) Mining Association Rules: A Continuous Incremental Updating Technique. In: International Conference on Web Information Systems and Mining, pp. 62–66. IEEE Computer Society

28. Dai, B.R., Lin, P.Y.: iTM: (2009) An Efficient Algorithm for Frequent Pattern Mining in the Incremental Database without Rescanning. In: Chien, B.-C., Hong, T.-P., Chen, S.-M., Ali, M. (eds.) IEA/AIE 2009. LNCS, vol. 5579, pp. 757–766. Springer, Heidelberg

29. Cheung, W., Zaiane, R.: (2003) Incremental Mining of Frequent Patterns without Candidate Generation or Support Constraint. In: Proc. of the Seventh International Database Engineering and Applications Symposium. IEEE Computer Society

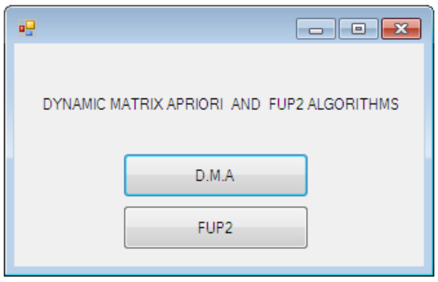
30. Laurentiu Cristofor. ARtool: “a synthetic dataset generator tool http://www.cs.umb.edu/~laur/ARtool.

31. Han, J. and M. Kamber (2006). Data Mining. Concepts and Techniques (2nd ed). Morgan Kaufmann.

.

APPENDIX A

CODES FOR FUP2 AND DMA USED IN THE EXPERIMENTS



Executable Form

DMA –Additions

```
static void Create_MFI(ref StreamReader rad)// creating MFI
{
int element;
string transRow;
int count = 0;
int j;

List<int> gm;
List<int> gm2 = new List<int>();
gm = new List<int>();
for (int z = 0; z <= ItemAndSupport.Count; z++)
{
gm.Add(0);
}
matrix.Add(gm);
matrix.Add(gm);
  while (string.Compare(transRow = rad.ReadLine(), "END_DATA") != 0)
{
int i = 0;
count++;
gm = new List<int>(gm2);
while ((j = transRow.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(transRow.Substring(i, j - i));

int indx = ss.FindIndex(e => e.Equals(element.ToString()));
if (indx != -1)
{
gm[indx] = 1;
}
i = j + 1;
}
if (matrix.Count == 2)
{
matrix.Add(gm);
ST.Add(1);
continue;
}
int exists = matrix.FindIndex(o => o.SequenceEqual(gm));

if (exists != -1)
ST[exists - 1] = ST[exists - 1] + 1;
else
{
matrix.Add(gm);
ST.Add(1);
}
}
}
```

void Modify_MFI()//Modification of MFI

```
{
for (int j = 1; j <= ItemAndSupport.Count; j++)
{
int prev = 1;

for (int i = 1; i <= ST.Count; i++)
{
if (matrix[i][j] > 0)
{
matrix[prev][j] = i;
prev = i;
}
}
}
}
```

void UpdatesItemsOrder(ref StreamReader rd)//First DMA scan

```
{
NewItemAndSupport = new Dictionary<string, double>();
// StreamReader AdditionFile = new StreamReader(fpath2.Text);
Readdata( rd, ref NewtranCount);
 foreach (KeyValuePair<string, double> hh in  ItemAndSupport.ToArray())
{
double sub_sp;
double w1 = ItemAndSupport[hh.Key];
double t3 = w1 * transCount / 100;
if (NewItemAndSupport.ContainsKey(hh.Key))
{
   sub_sp = (t3 + NewItemAndSupport[hh.Key]) * 100 / (NewtranCount + 
                           transCount);
ItemAndSupport[hh.Key] = sub_sp;
NewItemAndSupport.Remove(hh.Key);
if (sub_sp >= min)
{
frequentItems.Add(hh.Key, sub_sp);
}
}
else
{
sub_sp = t3 * 100 / (NewtranCount + transCount);
ItemAndSupport[hh.Key] = sub_sp;
if (sub_sp >= min)
frequentItems.Add(hh.Key, sub_sp);
}
}
              foreach (KeyValuePair<string, double> hh in 
                     NewItemAndSupport.ToArray())
{
double sub_sp = (hh.Value * 100) / (NewtranCount + transCount);
ItemAndSupport.Add(hh.Key, sub_sp);
newitems.Add(hh.Key);
if (sub_sp >= min)
frequentItems.Add(hh.Key, sub_sp);
}

  var newSortedFreqItems = (from h in ItemAndSupport orderby h.Value 
                 ascending select h).ToDictionary
(pair => pair.Key, pair => pair.Value);

    Newss = new List<string>(newSortedFreqItems.Keys);//new sorted 
          frequent 1-itemset where new frequents are expected
Newss.Insert(0, "0");
}
```

static void SecondUpdateScan(StreamReader r)second DMA scan

```
{
```

```
int element;
string transRow;
int j;
int count = 0;
List<int> Newgm = new List<int>();
List<int> gm2 = new List<int>();
for (int z = 0; z < Newss.Count; z++)
{
Newgm.Add(0);
gm2.Add(0);
}
while (string.Compare(transRow = r.ReadLine(), "END_DATA") != 0)
{
int i = 0;
count++;
Newgm = new List<int>(gm2);
while ((j = transRow.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(transRow.Substring(i, j - i));

int indx = Newss.FindIndex(e => e.Equals(element.ToString()));
if (indx != -1)
{
Newgm[indx] = 1;
}
i = j + 1;
}
// adding new transactions
int pos;
bool chkt = false;
for (int a = 2; a < matrix.Count; a++)
{
List<int> x1 = new List<int>(matrix[a]);
for (int h = 1; h < Newgm.Count; h++)
{
chkt = false;
if ((x1[h] >= 1 && Newgm[h] == 1) || (Newgm[h] == x1[h]))
{
chkt = true;
continue;
}
if (chkt == false)
{
chkt = false;
break;
}
}
if (chkt == true)
{
pos = a;
ST[pos - 1] = ST[pos - 1] + 1;
break;
}

}
      if  (chkt  ==  false)//  update  all  rows  of  the  matrix  where  the
      last occurance is 1;//from bottom-up
{
matrix.Add(Newgm);
ST.Add(1);

}
}
```

rowupdate();

```
}
```

static void colupdate()//updating columns of MFI Matrix

```
{
for (int m = 1; m < Newss.Count; m++)
{
int y = ss.IndexOf(Newss[m]);
if (string.Compare(ss[m], Newss[m]) != 0)
{
for (int n = 1; n < matrix.Count; n++)
{
int temp = (matrix[n][m]);
matrix[n][m] = matrix[n][y];
matrix[n][y] = temp;
}
}
int f2 = Newss.IndexOf(ss[m]);
string h = Newss[f2];
ss[m] = ss[y];
ss[y] = h;
}
}
```

static void rowupdate()// updating rows of the matrix

```
for (int j = 1; j < ST.Count; j++)
{
if (matrix[j][i] > 1)
{
```

```
next = matrix[j][i];
continue;
}

if (matrix[j][i] == 1)
{
matrix[next][i] = j;
next = j;
}
}
DMA- DELETIONS

public void Newupdate_matriXColumns(ref StreamReader fil)
{
 // Check if positions of frequent itemsets have changed.and have  mfi
                              modified;

if (ss.SequenceEqual(Newss))
{
NewReaddata2(fil);
}
else
{
NewReaddata2(fil);
colupdate();
}
}
```

FUP2- ADDITIONS

```
// GENERATING CANDIDATE ITEMSETS
while (L.Count != 0)
{

k++;
C.Clear();
C1.Clear();
string hh = null;
for (int x = 0; x < L.Count; x++)
{
string[] subL1 = L[x].ToString().Split(',');

for (int y = x + 1; y < L.Count; y++)
{
string[] subL2 = L[y].ToString().Split(',');
for (int m = 0; m < subL1.Length; m++)
{
```

```
for (int n = 0; n < subL2.Length; n++)
{
if (!((subL1[m] == subL2[n]) || (m < n)))

foreach (string ar in subL2)
{

if (!((ar) == (subL1[m])) && !(L[x].ToString().Contains(ar)))

foreach (string gg in L[x].ToString().Split(','))
{
if (Convert.ToInt32(ar) > Convert.ToInt32(gg))
hh = L[x] + "," + ar;
}

   if (!C1.Contains(hh) && !L.Contains(ar) && Convert.ToInt32(ar) >
                    Convert.ToInt32(subL1.Last()))

C1.Add(hh);
}
}
}
}
}

Console.WriteLine();
L1.Clear();
```

//Generating new frequent itemsets

```
foreach (string icomb in C1)
{
double yyy = 0;
file2 = new StreamReader(fpath.Text);
 while (string.Compare(transRow = file2.ReadLine(), "END_DATA") != 0)
{
int kkk = 0;
foreach (string i2 in icomb.Split(','))
{
int i = 0;

while ((j = transRow.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(transRow.Substring(i, j - i));
if (element.ToString() == i2)
{
kkk++;
break;

}
i = j + 1;

}


}
int y2 = icomb.Split(',').Length;
if (kkk == y2)
{
yyy++;
}

}
yyy = (yyy / transcount) * 100;
if (yyy >= support)
{
L1.Add(icomb);
m_dicFKrequentItems_L.Add(icomb, yyy);
}
}

L = L1;
}
```

FUP-Incremental mining

//Determine L1 count in the incremental db and D

```
  Dictionary<string, double> nfreq = new Dictionary<string, double>();
   Dictionary<string, double> newC = new Dictionary<string, double>();

foreach (KeyValuePair<string, double> nitem in FK.ToArray())
{
double s;
double sub_p = 0;
newfile = new StreamReader(fpath2.Text);
 while (string.Compare(transRow = newfile.ReadLine(), "END_DATA") != 0)
{

int kkk = 0;
foreach (string i2 in nitem.Key.Split(','))
{
int i = 0;
while ((j = transRow.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(transRow.Substring(i, j - i));
if (element.ToString() == i2)
{
kkk++;
break;
}
i = j + 1;
}
}
int y2 = nitem.Key.Split(',').Length;
if (kkk == y2)

sub_p++;

}
if (nitem.Value == 0.0)
{
double p;
p = (sub_p / tcount) * 100;
if (p >= support)

newC.Add(nitem.Key, p);

}
else
{
 s = (((sub_p + ((nitem.Value * transcount) / 100))) * 100) / (tcount +
                               transcount);
if (s >= support)
{
nfreq.Add(nitem.Key, s);
m_dicFKrequentItems_L.Remove(nitem.Key);
}
else
{
losersL1.Add(nitem.Key);
m_dicFKrequentItems_L.Remove(nitem.Key);
}
}
}


foreach (KeyValuePair<string, double> nitem in newC.ToArray())
```

```
{

double sub_supp = 0;
file2 = new StreamReader(fpath.Text);
 while (string.Compare(transRow = file2.ReadLine(), "END_DATA") != 0)
{
int kkk = 0;
foreach (string i2 in nitem.Key.Split(','))
{
int i = 0;

while ((j = transRow.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(transRow.Substring(i, j - i));
if (element.ToString() == i2)
{
kkk++;
break;
}
i = j + 1;
}
}
int y2 = nitem.Key.Split(',').Length;
if (kkk == y2)
{
sub_supp++;
}
}

double newsupp;
   newsupp = (((sub_supp + ((nitem.Value * tcount) / 100))) * 100) /
                              (tcount + transcount);
if (newsupp >= support)
{
nfreq.Add(nitem.Key, newsupp);
m_dicFKrequentItems_L.Remove(nitem.Key);
}
}

int newk = 1;
foreach (KeyValuePair<string, double> vv in nfreq)
{
nL.Add(Convert.ToInt32(vv.Key));
nL.Sort();
}
```
Call_cand_gen ();

```
}
```

FUP2-Deletion

```
//variables to hold data
   Dictionary<string, double> F1 = new Dictionary<string, double>();
Dictionary<string, double> P = new Dictionary<string, double>();
Dictionary<string, double> Q = new Dictionary<string, double>();
// Read new transactions
              foreach (KeyValuePair<string, double> jj in
                   m_dicFKrequentItems_L.ToArray())
{
if (jj.Key.Substring(0).Split(',').Length == 1)
{
F1.Add(jj.Key.ToString(), jj.Value);
}
}
foreach (KeyValuePair<int, double> kk in ITEMS)
{
if (!F1.ContainsKey(kk.Key.ToString()))
{
Q.Add(kk.Key.ToString(), kk.Value);
}
}

// string filepath2 = @"c:\Datasets\For Deletions2\7500.asc";
StreamReader file = new StreamReader(txtpath2.Text);

  while (string.Compare(trasRows = file.ReadLine(), "END_DATA") != 0)
{
tcount++;
}

//Determine L1 count in the incremental db and D
  Dictionary<string, double> allf = new Dictionary<string, double>();
 Dictionary<string, double> nfreq = new Dictionary<string, double>();
// for (int ii = 1; ii < F1.Count; ii++)
 foreach (KeyValuePair<string, double> itemii in F1.ToArray())//get new
                        size-1 freq items here
{
double sub_p = 0.0;
file = new StreamReader(txtpath2.Text);
  while (string.Compare(trasRows = file.ReadLine(), "END_DATA") != 0)
{
int kkk = 0;
foreach (string uu in itemii.Key.Split(','))
{
```

```
int i = 0;

while ((j = trasRows.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(trasRows.Substring(i, j - i));
if (element.ToString() == uu)
{
kkk++;
break;
}
i = j + 1;
}
}
int y2 = itemii.Key.Split(',').Length;
if (kkk == y2)
sub_p++;

}

double ll = F1[itemii.Key];
 double s = ((((ll * transcount) / 100) - sub_p)) * 100 / (transcount -
                                tcount);
if (s >= support)
{
allf.Add(itemii.Key, s);
double p = (sub_p / tcount) * 100;
if (p < support)
{
R.Add(itemii.Key);

}
}



}
foreach (KeyValuePair<string, double> HH in Q.ToArray())
{
double sub_p = 0.0;
file = new StreamReader(txtpath2.Text);
  while (string.Compare(trasRows = file.ReadLine(), "END_DATA") != 0)
{
int kkk = 0;
foreach (string uu in HH.Key.Split(','))
{
int i = 0;
```

```
while ((j = trasRows.IndexOf(" ", i)) != -1)
{
element = Convert.ToInt32(trasRows.Substring(i, j - i));
if (element.ToString() == uu)
{
kkk++;
break;
}
i = j + 1;
}
}
int y2 = HH.Key.Split(',').Length;
if (kkk == y2)
sub_p++;
}
double p = (sub_p / tcount) * 100;
if (p >= support)
{
Q.Remove(HH.Key);
}
else
{
// R.Add(HH.Key);
double ll = Q[HH.Key];
 double s = ((((ll * transcount) / 100) - sub_p)) * 100 / (transcount -
                               tcount);
if (s >= support)
{
allf.Add(HH.Key, s);
R.Add(HH.Key);

}
}


}


foreach (KeyValuePair<string, double> vv in allf)
{
nL.Add(Convert.ToInt32(vv.Key));
nL.Sort();
}
Call cand_gen ();
}
```