

**N x N FREQUENCY MATRIX FOR FREQUENT  
ITEMSETS DISCOVERY**

*R-06388*



*Developed By:*

**Awais Usman Butt**

**291-FAS/MSCS/F06**

*Supervised by:*

**Muhammad Imran Saeed**

**International Islamic University Islamabad**

**Faculty of Basic and Applied Sciences**

**Department of Computer Science**

**2009**

MS

005.369

BUN

Computer programming

2/11/2010 Sg



Accession No. TH-6388

*In The Name of*

**ALLAH ALMIGHTY**

*The Most Merciful, The Most Beneficent*

**Department of Computer Science**

**International Islamic University, Islamabad**

**Final Approval**


It is certified that we have read the project titled "N x N Frequency Matrix" submitted by **Awais Usman Butt**; Reg. #: 291-FAS/MSCS/F06. It is our judgment that this project is of sufficient standard to warrant its acceptance by **International Islamic University, Islamabad** for the degree **MS in Computer Science**.

**COMMITTEE**

**External Examiner:**

**PROF. DR. NAZIR AHMED SANGHI**

(Allama Iqbal Open University, Islamabad)

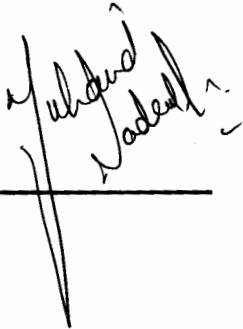


---

**Internal Examiner:**

**ASST. PROF. MUHAMMAD NADEEM**

(International Islamic University, Islamabad)

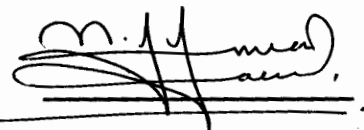


---

**Supervisor:**

**ASST. PROF. IMRAN SAEED**

(International Islamic University, Islamabad)



---

**Dated:** 29-07-2009

**A dissertation submitted to the  
Department of Computer Science,  
Faculty of Basic and Applied Sciences,  
International Islamic University, Islamabad, Pakistan,  
as a partial fulfillment of the requirement for the award of the degree of  
MS Computer Science**

**To**  
***My Parents***

**The continuous support, encouragement and prayers of my parents  
have always been the key to my success.**

***My Honorable Teachers***

**The guidance and motivation from my parents motivated me to  
accomplish this task.**

## **Declaration**

I, hereby declare that “N x N Frequency Matrix” neither as a whole nor as a part thereof has been copied out from any source. I have developed this project and the accompanied report entirely on the basis of my personal efforts made under the sincere guidance of my supervisor. No portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institution of learning.

**Dated:** \_\_\_\_\_

**Awais Usman Butt**

**291-FAS/MSCS/F06**

## **Acknowledgement**

All acclamation to Allah Almighty Who has empowered and enabled me to accomplish the task successfully.

First of all I would like to thank Allah Almighty who helped me out in every problem during the research. I would like to express my serious and humble gratitude to Almighty whose blessings, help and guidance has been a real source of all achievements in life. I would like to admit that the completion of my thesis is due to my loving parents.

I would like thank my supervisor ASST. PROF. Muhammad Imran Saeed for his sincere efforts to guide me throughout this project.

I would like to thank my friend for their cooperation and encouragement.

**Awais Usman Butt**

**291-FAS/MSCS/F06**



## **PROJECT IN BRIEF**

<b>Project Title</b>	N x N Frequency Matrix
<b>Undertaken By</b>	Awais Usman Butt
<b>Supervised By</b>	ASST. PROF. Imran Saeed
<b>Starting Date</b>	September 2008
<b>Ending Date</b>	December 2008
<b>Software Used</b>	Visual C# 2005, SQL Server 2005
<b>Operating System</b>	Microsoft Window XP
<b>System Used</b>	Pentium IV

## ABSTRACT

Nowadays data grows rapidly and the rapid growth of data brings along the problems of data storage, data usage and data analysis. Data mining is used to analyze the data. In the field of data mining “Association rule discovery” is a very important research topic. “Frequent Itemset Discovery” is the key process towards mining association rules. Numerous efforts had been made to address and to provide solutions for frequent itemset discovery. The solutions faced problems like repeated I/O and inefficient CPU resource utilization. A new method is proposed in this report that minimizes the I/O scans and CPU resource utilization and supports large database. The new algorithm adopts an “ $N \times N$  Frequency matrix” to map the complete database in a 2 dimensional structure. Single DB scan is required to build the  $N \times N$  Frequency Matrix.

## Table of Content

<i>Introduction</i>	<b>4</b>
<b>1. INTRODUCTION</b>	<b>5</b>
1.1. OBJECTIVES	5
1.2. ORGANIZATION OF STUDY	6
<i>Frequent Item-set Discovery</i>	<b>7</b>
<b>2. FREQUENT ITEM-SET DISCOVERY</b>	<b>8</b>
2.1. DEFINITION AND TERMINOLOGY	8
2.2. APPROACHES USED TO FIND FREQUENT ITEM-SETS	8
2.2.1. CANDIDATE GENERATION AND TESTING	9
2.2.1.1. APRIORI ALGO	9
2.2.1.2. DIRECT HASHING AND PRUNING (DHP)	10
2.2.1.3. DYNAMIC ITEM-SET COUNTING (DIC)	10
2.2.2. PREFIX TREE STRUCTURE	10
2.2.2.1. FP GROWTH	11
2.2.3. MATRIX BASED APPROACH	11
2.2.3.1. HORIZONTAL DESIGN	11
2.2.3.2. VERTICAL DESIGN	12
2.2.3.3. ALGO BASED ON BOOLEAN MATRIX	12
2.2.3.4. INVERTED MATRIX	12
2.2.4. STRUCTURE-BASED APPROACH	13
2.2.4.1. CUBIC STRUCTURE-BASED APPROACH	13
<i>Literature Review</i>	<b>14</b>
<b>3. LITERATURE REVIEW</b>	<b>15</b>
3.1. APRIORI ALGO	15
3.1.1. APRIORI CANDIDATE GENERATION	16
3.1.2. APRIOR SUBSET FUNCTION	16
3.2. DIRECT HASHING AND PRUNING (DHP)	17
3.2.1. EFFICIENT GENERATION FOR LARGE ITEM-SETS	17
3.3. DIC	21
3.3.1. DIC ALGO STEPS	22
3.4. FP GROWTH	23
3.5. HORIZONTAL DESIGN	27
3.6. VERTICAL DESIGN	27
3.7. ABBM	28
3.7.1. ALGO DETAILS	29
3.7.1.1. DATASET CONVERSION	29
3.7.1.2. RECURRENT 1-ITEM-SET PRODUCTION	29
3.7.1.3. PRUNING THE BOOLEAN MATRIX	30
3.7.1.4. PRODUCTION OF RECURRENT K-ITEM-SETS	30
3.7.1.5. EXAMPLE	31
3.8. INVERTED MATRIX	33
<b>4. PROBLEM STATEMENT</b>	<b>37</b>
<i>Proposed Solution</i>	<b>38</b>
<b>5. PROPOSED ALGORITHM</b>	<b>39</b>

---

<b>6. SCOPE</b>	<b>43</b>
<i>Methodology</i>	<b>44</b>
<b>7. MODULES</b>	<b>45</b>
7.1. GENERATION OF N X N FREQUENCY MATRIX	45
7.2. DISCOVERY OF FREQUENT ITEM-SETS	47
7.3. Working of Recursive Method	50
<i>Design and Implementation</i>	<b>51</b>
<b>8. DESIGN AND IMPLEMENTATION</b>	<b>52</b>
8.1. FREQUENCY MATRIX GENERATION	52
8.1.1. SetValuesInHashTable ()	52
8.1.2. CreateMatrix ()	53
8.1.3. SetFrequencies ()	54
8.1.4. CopyMatrix ()	55
8.2. DISCOVERY OF FREQUENT ITEM-SETS	56
8.2.1. GetItemsIndex ()	56
8.2.2. GetFrequentItem-sets ()	57
8.2.3. FormatHashTable ()	58
8.2.4. GetItem-set ()	59
8.2.5. GetRemainingItems ()	59
8.3. Software and Hardware Requirements	60
<b>9. CONCLUSIONS</b>	<b>61</b>
9.1. Example	61
9.2. Problem Area	62
<b>10. REFERENCES</b>	<b>63</b>
<b>11. GLOSSARY OF ACRONYMS</b>	<b>65</b>

## Table of Figures

Figure 1 : Algo for Large Item-set Generation .....	18
Figure 2 : DHPAlgo .....	19
Figure 3 : Sub procedures for algo DHP .....	20
Figure 4 : An Item-set Lattice .....	21
Figure 5 : Construction of compact data structure for FP growth .....	25
Figure 6 : FP Tree created from Compact data structure .....	26
Figure 7 : Horizontal Layout .....	27
Figure 8 : Vertical Layout .....	28
Figure 9 : Algo to generate Frequent Item-set .....	31
Figure 10 : Transactional dataset and its Boolean Matrix .....	32
Figure 11 : Frequency of Items .....	33
Figure 12 : Inverted Matrix .....	34
Figure 13 : Algo to create Inverted Matrix .....	35
Figure 14 : Algo to create Set of Frequent Patterns .....	36
Figure 15 : Frequency Matrix after one transaction .....	39
Figure 16 : Frequency Matrix after two transactions .....	40
Figure 17 : Frequency Matrix after three transactions .....	40
Figure 18 : Frequency Matrix after four Transactions .....	41
Figure 19 : Frequency Matrix for 10 transactions .....	42
Figure 20 : Frequency Matrix Creation Steps .....	45
Figure 21 : Discovery of Frequency Itemsets .....	47
Figure 22 : Intermediate Results for Frequency Itemsets .....	49
Figure 23 : Itemsets from NxN Frequency Matrix .....	49
Figure 24 : Simulator view of Frequency Matrix .....	60
Figure 25 : Selecting Problem Area .....	61
Figure 26 : Final Problem Area .....	62

# **Introduction**

# 1. INTRODUCTION

Present era is the era of technology and technology has its impact on all fields of science, business, medicine, military, etc. With the introduction to technology there comes a problem of generating a huge amount of data. The same rate of growth in processing power of evaluating and analyzing the data did not follow this massive growth. Due to this phenomenon, a great volume of data is still kept without being considered.

Data mining, a research field that tries to simplify this problem, proposes some solutions for the mining of noteworthy and potentially useful patterns from these large collections of data. One of the canonical tasks in data mining is the discovery of association rules. Discovering association rules, measured as one of the most important tasks, have been the focal point of many studies in last few years. Finding of frequent item-set presents a break through for mining association rules [1]. Finding frequent item-sets and then mining association rules on the basis of discovered item-sets was first brought up by (Agrawal, Imienlinski and Swami in 1993) [1].

In this literature a new method for discovering frequent item-sets is proposed, which used  $N \times N$  frequency matrix.  $N \times N$  frequency matrix is a summarized form of the transaction dataset (Market basket data, or dataset containing information about transactions, transaction composed of items). This method is build to minimize the repeated database scans and to provide a platform for repeatedly applying mining algo for difference user input and requirements.

## 1.1. OBJECTIVES

Many techniques have been projected using a chronological or parallel representation. However, the on hand algos depend a lot on enormous computation that might results high relevancies on the memory size or recurring I/O scans for the data sets. Algorithms

at present projected in the literature to mine association rules are not adequate for particularly large datasets and new solutions that are independent of recurring I/O scans and less reliant on memory size, still have to be found. Objectives of this algo is to find out a solution that doesn't rescans the database again and again and has minimum utilization of memory and also takes less CPU resources and computation time.

## **1.2. ORGANIZATION OF STUDY**

Topic is discussed in section 2, Literature review of the topic is discussed in section 3, proposed solution is discussed in section 5, Scope is discussed in section 6, Modules of the proposed solution is discussed in section 7, Design and Implementation of the proposed solution is discussed in section 8. Terms and abbreviation used in this document is given in glossary of acronyms section 11.



# Frequent Item-set Discovery

## 2. FREQUENT ITEM-SET DISCOVERY

Frequent item-set discovery is mostly related to Market Basket Data. Market Basket Data is the data related to super store and super market etc. i.e. dataset made up of transactions and the transactions are made up of items. (Continue from here)

### 2.1. DEFINITION AND TERMINOLOGY

Problem can be defined as “In a database ‘D’ there are ‘T’ transactions. Each transaction ‘Ti’ ( $T_i \in T$ ) has ‘TI’ items where  $I > 0$ . ‘TI’ is a subset of I (items, products or parts etc.) such that  $I = TI_1, TI_2, TI_3, \dots, TI_n$  where  $n > 0$ . FI is an item or a set of items such that FI is a subset of I. FI is said to be frequent if and only if FI is a subset of Ti where ‘i’ is greater or equal to Min support ‘min\_supp’. Min support is a checkpoint set by analyst such that the number of occurrences of FI for database D should be greater or equal to numeric value. Min support and minimum confidence is the check point for discovering of association rules from the database ‘D’. Confidence is the ratio of support of antecedent determining consequent to the support of antecedent. Min support and minimum confidence is the check point for mining association rules”.

### 2.2. APPROACHES USED TO FIND FREQUENT ITEM-SETS

There are several methods to find the frequent item-sets. These methods are groups on the basis of the approach used. These approaches are given below.

## 2.2.1. CANDIDATE GENERATION AND TESTING

This type of approach generates candidate item-sets from the database, and then compares each of the candidate item-set recursively with the database records (transactions). Size of candidate item-set  $K$  (in terms of number of items in a set) depends upon occurrences of the candidate item-set  $K-1$ .

### 2.2.1.1. APRIORI ALGO

The main scheme of the algo is based on the a priori theory, namely, an item-set can only be recurrent if all its subsets are also recurrent. I.E., if an item-set is not recurrent, no superset of it can be recurrent. Taking advantage of this knowledge makes possible to decrease the search space efficiently when finding out the recurrent item-sets, because the number of candidate sets can be reduced by taking advantage of this knowledge. The Apriori algo is an iterative technique, which demonstrates that it determines the  $k$ -item-sets during the  $k$ th dataset scan [4].

The first go by of the Apriori algo basically calculates item occurrences to find out the large 1-item-sets. A succeeding pass, say pass  $k$ , consists of two phases. First, the large item-sets  $L_{k-1}$  found in  $(k-1)$ th pass are used to generate the candidate item-sets  $C_k$ . Next, the database is scanned and the support of candidates in  $C_k$  is counted. For fast counting, it is needed to efficiently determine the candidates in  $C_k$  that are contained in a given transaction [1].

### **2.2.1.2. DIRECT HASHING AND PRUNING (DHP)**

The direct hashing and pruning (DHP) makes use of a hash method for entrant item-set creation throughout the groundwork iterations and enforce pruning techniques to gradually trim down the operational database size [2].

### **2.2.1.3. DYNAMIC ITEM-SET COUNTING (DIC)**

DIC trims down the number of go by through the data while maintaining the count of item-sets which are calculated in any go by comparatively low as in contrast to methods which relay on sampling. “The insight following DIC is that it works like a train running over the data with stops at intervals  $M$  transactions apart. When the train reached the end of the transaction file, it has made on pass over the data and it starts over the beginning for the next pass. The “passengers” on the train are item-sets. When an item-set is on the train, we count its occurrence in transactions that are read” [3].

## **2.2.2. PREFIX TREE STRUCTURE**

In computer science, a prefix tree is a prearranged tree data organization that is used to accumulate an associative array where the keys are typically ordinary text. Contrasting a binary search tree, no joint in tree supplies the key associated with that joint; in its position, its position in tree shows what key it is connected with. Each of the children of a joint have a same prefix of the series associated with that node, and the source is related to the blank string. Values are typically not related with each node, only with leaves and some inner nodes that correspond to keys of significance.

### **2.2.2.1. FP GROWTH**

A well-liked "preprocessing" tree formation is the FP tree proposed by Han et al. (2000). The FP tree stores a solitary item (attribute) at each node, and includes supplementary relations to assist processing. These links start from a header table and link together all nodes in FP tree which store the same "label", i.e. item.

The algo works as follows. Throughout the first dataset scan the recurrence of every item is calculated and the occasional items are eliminated. Then in descending order the recurrent items are sorted. The FP-tree is populated during the second dataset scan by reading the transactions and the recurrent items are mapped into the FP tree. In this way the dataset is pruned and is packed together into main memory. Purpose of FP tree is to keep transactions in main memory in such a way that the process of discovering association rules is an efficient proces. [4].

### **2.2.3. MATRIX BASED APPROACH**

A matrix is a two dimensional array of data, letters, or other facts placed in rows and columns.

#### **2.2.3.1. HORIZONTAL DESIGN**

Horizontal design is frequently used; it relates and integrates every item of the common transition with respect to transaction id. In this technique the primary key of the target dataset is the id of the transaction. The most important property of the horizontal design is the integration of all items of the same transaction; this design also experience some restrictions like unnecessary processing as items don't have index created upon them [5].

### **2.2.3.2. VERTICAL DESIGN**

In case of vertical design, transaction are integrated and combined with respect to the item. In this technique the id of each trace is the item. In this technique the primary key of the table is the item, and each item's record contains the ids of the transactions in which this item occurs. One of the main advantages of this technique is, on items its acts as an index and decreases the probability of rescanning the complete dataset again and again. On the other hand, vertical design still requires the costly candidacy generation part. Mining exceptionally huge datasets with this design becomes nearly impractical because of candidate generation and the additional processes attached with this approach. [5].

### **2.2.3.3. ALGO BASED ON BOOLEAN MATRIX**

Algo based of Boolean Matrix (ABBM) used the horizontal design. On x axis items are mapped and on y axis transaction are mapped. Each cubicle of the ABBM represents a Boolean value the existing of an item in a particular transaction [6].

### **2.2.3.4. INVERTED MATRIX**

The Inverted Matrix is the hybrid of horizontal and vertical approaches; it uses the advantages of both the techniques and tries to eliminate the disadvantages and limitations of both the techniques up to the maximum. The basis idea behind the technique is to integrate every item with all of its corresponding transactions, i.e the transactions containing the items, and then relates each transaction with its related items. Alike to the vertical design, in this design key of record is the item. This technique is different from the vertical design in a sense that each cell of the matrix is not a single value, but it is data structure made up of transaction id and the pointer, the pointer points to the position of the subsequent item on same transaction. [5].

## **2.2.4. STRUCTURE-BASED APPROACH**

Structure based approaches is the emerging approach in which databases are mapped into multidimensional structure like cube.

### **2.2.4.1. CUBIC STRUCTURE-BASED APPROACH**

The Cubic algo is a new method to discover the recurrent 4-item-sets swiftly. It find outs the 4-item-sets in only two complete dataset searches. An “Upper triangular matrix”  $M$  is used to calculate the support of recurrent 1 and recurrent 2 itemsets. If there are  $n$  transaction in the dataset, then  $n(n+1)/2$  will be the size of the matrix . Counters for the entrant item are held up in the diagonal of the matrix, and the additional cubicles are counters for the item pairs. A direct indexing technique is used to count the support of the items by means of the matrix and in an efficient manner [4].

An additional dataset check is made to calculate the three and four recurrent item-sets. An index table based on a cubic structure is used for skillful counting of the support of the entrant itemsets. This hash table is developed when passing through the matrix  $M$ . The rows of the matrix, which have minimum support greater then the threshold are used to create a cube. A cube is used to store 3 to 4 entrant itemsets that belongs to recurrent 2-itemsets and starting with the same item. In this way the first item of applicant chooses the suitable cube and the additional items deals with the cubicles in cube [4].

# Literature Review



### 3. LITERATURE REVIEW

A number of efficient association rule mining algos, techniques and methods have been proposed in last few years.

#### 3.1. APRIORI ALGO

In the first step method calculates the recurrence of each item and find out large 1-item-sets. A successive iteration, say iteration  $i$ , is made up of two stages. First, the candidate item-sets  $X_i$  are produced, with the help of large item-sets  $G_{i-1}$  calculated in  $(i-1)$ th iteration of Apr-Generation method. After that, the dataset is examined and the support of entrants in  $X_i$  is calculated. To count efficiently, we need to proficiently conclude on the candidates in  $X_i$  that are contained in a given transaction  $y$ . [1].

- $G_1 = \{\text{Recurrent 1-Item-sets}\};$
- **for**(  $j = 2; G_{j-1} \neq \emptyset; j++$ )
- {
  - $N_j = \text{Apr-Generation}(G_{j-1});$
  - **ForAll** (transactions  $y \in S$ )
  - {
    - $N_t = \text{subset}(N_j, y);$
    - for candidates  $c \in N_t$
    - do
      - $c.\text{increment}++;$
  - }
  - $L_j = \{c \in N_j \mid c.\text{increment} \geq \text{minimum\_support}\}$
- }
- $\text{Result} = \bigcup_j L_j$

*(Above algo and details are referred from [1])*

### 3.1.1. APRIORI CANDIDATE GENERATION

The Apr-Generation function takes as argument  $X_{j-1}$ , the set of all recurrent  $(j-1)$ -item-sets. It gives out a powerset of the set of all recurrent  $j$ -item-sets. The execution steps of the method are as follows. First, in make up step, we join  $X_{j-1}$  with  $X_{j-1}$ . [1]

- Add up in  $N_j$
- Pick  $z.itm1, z.itm2, \dots, z.itmj-1, y.itmj-1$
- from  $X_{j-1}$   $z, X_{j-1}$   $y$
- whenever  $z.itm1$  equals  $y.itm1, z.itmj-2$  equal  $y.itmj-2, z.itmj-1$  less then  $y.itmj-1$ ;

Next, in prune step, we delete all item-sets  $c \in C_j$  such that some  $(j-1)$ -subset of  $c$  is not in  $L_{j-1}$ ; [1]

- **forall** item-sets  $c \in N_j$
- **do**
  - **forall**  $(j-1)$  subsets  $s$  of  $c$
  - **do**
    - **if** ( $s \notin L_{j-1}$ )
    - **then**
      - **delete**  $c$  from  $N_j$

*(Above algo and details are referred from [1])*

### 3.1.2. APRIOR SUBSET FUNCTION

A hash table is used to keep the entrant itemsets  $C_j$ . A collection of itemsets or a complete hash table can be stored at the nodes of the hash tree. Within hash table, every container of the hash table spots to further nodes. If at depth  $l$  is the root of the hast tree

and at depth  $d$  is an internal node, then internal node will be pointing to node at depth  $d+1$ . Initially all node are created as leaf and itemsets are stored in leaves [1].

Starting from the root node, the subset method discovers every one of the applicant set enclosed in a transaction  $t$ . "If we are at leaf, we find which of the item-sets in leaf are contained in  $t$  and add references to them to the answer set. If we are at an interior node and we have reached it by hashing the item  $I$ , we hash on each item that comes after  $I$  in  $t$  and recursively apply this procedure to the node in corresponding bucket. For the root node, we hash on every item in  $t$ " [1].

*(Above algos and details are referred from [1])*

## **3.2. DIRECT HASHING AND PRUNNING (DHP)**

DHP is an useful hash-based algo for the contestant set generation. Explicitly, the quantity of contestant 2-item-sets produced by the proposed algo is, in orders of magnitude, lesser than that by earlier techniques, thus improving the efficiency factor. Note that the production of smaller contestant sets enables us to efficiently cut down the transaction dataset size at a much former phase of the iterations, thereby reducing the computational cost for later on iterations considerably. DHP has two main characteristics: First is the proficient production of large item-sets and the second is efficient trimming of the cardinality of the dataset [2].

### **3.2.1. EFFICIENT GENERATION FOR LARGE ITEM-SETS**

In each pass the set of large item-sets,  $K_i$ , to outline the set of contestant large item-sets  $X_{i+1}$  by combining  $K_i$  with  $K_i$  on  $(i-1)$  (indicated by  $K_i * K_i$ ) familiar items for the next

iteration. To determine  $K_{i+1}$ , dataset is scanned again and support of each item-set in  $X_{i+1}$  is counted [2].

```

/* Part 1 */
s = a minimum support;
set all the buckets of  $H_2$  to zero; /* hash table */
forall transaction  $t \in D$  do begin
    insert and count 1-items occurrences in a hash tree;
    forall 2-subsets  $x$  of  $t$  do
         $H_2[h_2(x)] ++$ ;
end
 $L_1 = \{c | c.count \geq s, c \text{ is in a leaf node of the hash tree}\}$ 

/* Part 2 */
k = 2;
 $D_k = D$ ; /* database for large k-itemsets */
while ( $|\{x | H_k[x] \geq s\}| \geq LARGE$ ) {
    /* make a hash table */
    gen_candidate( $L_{k-1}, H_k, C_k$ );
    set all the buckets of  $H_{k+1}$  to zero;
     $D_{k+1} = \phi$ ;
    forall transactions  $t \in D_k$  do begin
        count_support( $t, C_k, k, \hat{t}$ ); /*  $\hat{t} \subseteq t$  */
        if ( $|\hat{t}| > k$ ) then do begin
            make_hasht( $\hat{t}, H_k, k, H_{k+1}, \hat{t}$ );
            if ( $|\hat{t}| > k$ ) then  $D_{k+1} = D_{k+1} \cup \{\hat{t}\}$ ;
        end
    end
    end
     $L_k = \{c \in C_k | c.count \geq s\}$ ;
    k ++;
}

```

**Figure 1 : Algo for Large Item-set Generation [2]**

(Above algo and details are referred from [2])

(Above algos and details are referred from [2])

```

/* Part 3 */
gen_candidate( $L_{k-1}, H_k, C_k$ );
while ( $|C_k| > 0$ ) {
     $D_{k+1} = \phi$ ;
    forall transactions  $t \in D_k$  do begin
        count_support( $t, C_k, k, \hat{t}$ );      /*  $\hat{t} \subseteq t$  */
        if ( $|\hat{t}| > k$ ) then  $D_{k+1} = D_{k+1} \cup \{\hat{t}\}$ ;
    end
     $L_k = \{c \in C_k | c.count \geq s\}$ ;
    if ( $|D_{k+1}| = 0$ ) then break;
     $C_{k+1} = \text{apriori\_gen}(L_k)$ ;      /* refer to [5] */
     $k++$ ;
}

```

**Figure 2 : DHP Algo [2]**

(Above algo and details are referred from [2])

Part 1 obtains a set of large 1-item-sets and builds a hash table (i.e.,  $J_2$ ) for 2-item-sets. Part 2 produces a list of contestant item-sets  $X_k$  based on the hash table (i.e.,  $J_k$ ) produced in previous iteration, find out the set of large  $k$ -item-sets  $K_k$ , decreases the cardinality of dataset for the up coming large item-sets, and build a hash table for contestant large  $(k+1)$ -item-sets. Part 3 is on the whole similar as part 2 apart from that it does not utilize a hash table. Note that DHP is particularly influential to find out large item-sets in near the beginning stages, thus improving the performance bottleneck. The cardinality of  $X_k$  reduces considerably in later stages, thus rendering minute explanation of its additional filtering. [2].

```

Procedure gen_candidate( $L_{k-1}, H_k, C_k$ )
   $C_k = \phi$ ;
  forall  $c = c_p[1] \cdot \dots \cdot c_p[k-2] \cdot c_p[k-1] \cdot c_q[k-1]$ ,
   $c_p, c_q \in L_{k-1}, |c_p \cap c_q| = k-2$  do
    if ( $H_k[h_k(c)] \geq s$ ) then
       $C_k = C_k \cup \{c\}$ ; /* insert  $c$  into a hash tree */
  end Procedure

Procedure count_support( $t, C_k, k, \hat{t}$ )
  /* explained in Section 3.2 */
  forall  $c$  such that  $c \in C_k$  and  $c (= t_{i_1} \dots t_{i_k}) \in t$  do
    begin
       $c.count++$ ;
      for ( $j = 1; j \leq k; j++$ )  $a[i_j]++$ ;
    end
    for ( $i = 0, j = 0; i < |t|; i++$ )
      if ( $a[i] \geq k$ ) then do begin  $\hat{t}_j = t_{i_j}; j++$ ; end
  end Procedure

Procedure make_hasht( $\hat{t}, H_k, k, H_{k+1}, \hat{t}$ )
  forall ( $k+1$ )-subsets  $x (= \hat{t}_{i_1} \dots \hat{t}_{i_{k+1}})$  of  $\hat{t}$  do
    if (for all  $k$ -subsets  $y$  of  $x, H_k[h_k(y)] \geq s$ ) then do
      begin
         $H_{k+1}[h_{k+1}(x)]++$ ;
        for ( $j = 1; j \leq k+1; j++$ )  $a[i_j]++$ ;
      end
      for ( $i = 0, j = 0; i < |\hat{t}|; i++$ )
        if ( $a[i] > 0$ ) then do begin  $\hat{t}_j = \hat{t}_{i_j}; j++$ ; end
  end Procedure

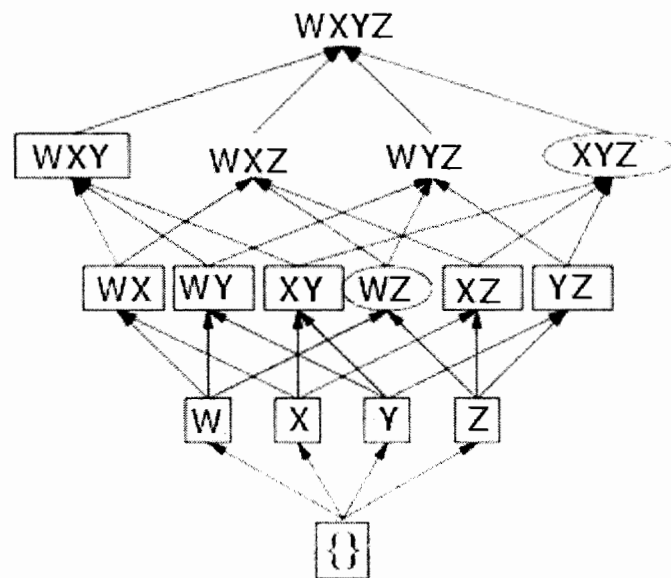
```

**Figure 3 : Sub procedures for algo DHP [2]**

(Above algo and details are referred from [2])

### 3.3. DIC

In DIC item-sets creates a large web with the vacant item-set at the base and the set of all items are the peak. Item-sets are recurrent if their count is large. However, it is impracticable to calculate all of the small item-sets. Luckily, it is adequate to calculate just the smallest ones (the item-sets that do not include any other small item-sets) since if an item-set is not recurrent, all of its supersets are small too. An algo which counts all the recurrent item-sets must uncover and calculate all of the large item-sets and the minimal small item-sets. [3]



**Figure 4 : An Item-set Lattice [3]**

(Above algo and details are referred from [3])

DIC algo, marks item-sets in four different possible ways:

1. Solid box – definite large item-set – an item-set we have completed counting that goes beyond that support threshold.
2. Solid circle – definite minute item-set – an item-set we have ended counting that is underneath the support threshold.

3. Dashed box – assumed large item-set – an item-set we are still including that outshine the support threshold.
4. Dashed circle – assumed small item-set – an item-set we are still including that is underneath the support threshold.

### **3.3.1. DIC ALGO STEPS**

1. The blank item-set is associated with a solid box. Every one of the 1-item-sets is associated with dashed circles. Every one of other item-sets is unassociated.
2. Examine X transaction. For each transaction, increase the particular counters for the item-sets associated with dashes.
3. If an associated circle has a count that goes beyond the support threshold, transforms it into a dashed square. If any direct superset of it has all of its subsets as solid or dashed squares, include a new counter for it and construct into a dashed circle.
4. If a dashed item-set has been calculated through all the transactions, convert into solid and stop calculating it.
5. If we reach at the end of the transaction file, wind back to the beginning.
6. If any of the dashed item-sets stay behind, jump to step 2.

DIC uses this technique to set up calculating just the 1-item-sets and then rapidly attaches counters 2, 3, 4,..., k-item-sets. Subsequent to a small number of pass over the data (typically fewer than two for small values of X) if comes to an end counting all the item-sets. In an ideal world, X should be as minimum as possible so counting of item-sets will set up very early in step 3. Though, step 3 and 4 gains substantial overhead so X should be greater or equal to 100. [3]

*(Above algos and details are referred from [3])*



### 3.4. FP GROWTH

First, a compressed data structure, called frequent-pattern tree is created, which is comprehensive prefix-tree formation accumulating significant, quantitative details about FPs (frequent pattern). To make certain that tree formation is packed together and instructive, nodes of the trees will be occupied by recurrent 1-item-set, and the order of nodes is such that probability of node is greater in more recurrently occurring nodes than that of less recurrently occurring nodes. Fp trees set the plate form for mining process and there is no need to rescan the dataset again and again. [4]

An FP tree-based model-fragment development mining technique is built up, which initializes from a recurrent length-1 model (as an early suffix outline), inspects just its “*conditional-pattern base*” (a “sub-dataset” which is made up of recurrent items co-occurring with the suffix outline), creates its (conditional) FP tree, and apply mining process repeatedly on suffix tree. The model development is accomplished when the suffix mode is concatenated with the recent ones produced from a conditional FP tree. Because the recurrent item-set in any transaction has always been prearranged in corresponding path of the recurrent-model trees, model development guarantees the comprehensiveness of the result. The most important actions of mining are count buildup and prefix path count fine-tuning, which are generally much less expensive than contestant production and outline matching procedures performed in most Apriori-like algos.[4].

The search method engaged in mining is a *partitioning-based, divide-and conquers method* other than Apriori-like level-wise production of the grouping of recurrent item-sets. This noticeably decreases the magnitude of conditional-model base produced at the succeeding stage of exploration in addition to the size of its matching provisional FP tree. Furthermore, it changes the problem of finding long recurrent model to looking for

shorter ones and then concatenates the suffix. It makes use of the least recurrent items as suffix, which offers good selectivity. All of these method contribute to substantial decrease of search costs.[4].

### Characteristics of FP Growth

- After thoroughly studying the characteristics of FP trees, it is figured out that FP tree may not at all times be minimal.
- To accelerate the growth of FP growth some optimizations are projected, for example, a method to control particular path FP tree has been additionally be proposed for performance enhancements.
- A dataset projection scheme has also been introduced to deal with the circumstances when it is difficult to detain FP tree in main memory—in large datasets this case may happens.

### Recurrent Pattern tree construction:

Frequent-pattern tree can be constructed as follows.

- In the first step of recurrent pattern tree construction, the target dataset is completely scanned and recurrence of each item is calculated and a list is created  $[[z - 5],[y - 5],[x - 4],[w - 4],[v - 4],[u - 4]]$ . The number after “-“ represents the recurrence of the item in the dataset. Items in the list are ordered in descending order with respect to items recurrence.
- In next step frequent pattern tree is constructed by scanning the dataset second time. First a root in the tree is added and is marked with null.
  - First branch of the frequent pattern tree is created by scanning the first transaction of the database. Items in the transactions are ordered with respect to the recurrence of items in the recurrence list  $\{(z: 1), (y: 1), (x: 1), (v: 1), (u: 1)\}$ .

- In the next step second transaction is scanned, as item in the transaction is ordered therefore items are checked with the first branch of the frequent pattern tree. The frequency of the items sharing the common prefix (z, y, x) is incremented by one and the nodes that don't share the prefix (u, v) are added as new nodes in the tree.
- As only one item of the next transaction share the common prefix (z), its frequency is incremented by 1 and a new node (w) is added to the tree and is linked with z.
- A new branch is added to the tree when fourth transaction of the dataset is scanned as its item doesn't share any common prefix. (y, w, u).
- As for the last transaction all the items share the common prefix, frequency of all the items sharing the common prefix is incremented by 1.

TID	Items bought	(Ordered) frequent items
100	Z, X, y, d, g, i, v, u	Z, y, X, v, u
200	x, w, y, Z, l, v, o	Z, y, X, w, v
300	w, Z, h, j, o	Z, v
400	w, y, k, s, u	y, v, u
500	x, Z, y, e, l, u, v, n	Z, y, X, v, u

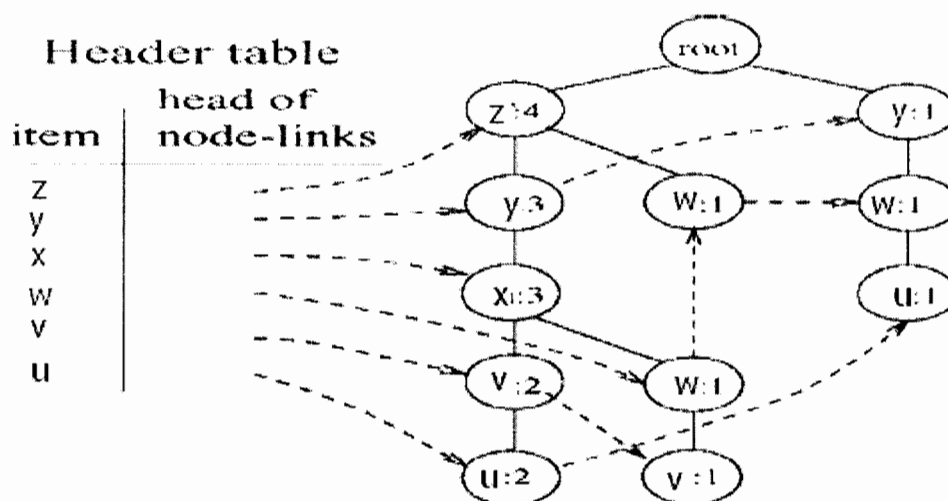
**Figure 5 : Construction of compact data structure for FP growth [4]**

(Above algo and details are referred from [4])

With the help of this example we can develop a recurrence tree by the following method..

**Definition 1 (FP tree).** A frequent-pattern tree (or FP tree in short) is a tree arrangement defined below.

- It consists of single root tagged as “null”, a collection of item-prefix sub-structures as the offspring of the origin, and a frequent-item-header table.
- every node in item-prefix sub-tree is made up of three fields: item-name, calculation, and node-link, where item-name records which item this node stands for, calculation records the total count of transactions corresponding to the segment of the path approaching this node, and node-link relates to the next node in FP tree having the same item-name, or null if there is not any.



**Figure 6 : FP Tree created from Compact data structure [4]**

*(Above algo and details are referred from [4])*

- every entry in frequent-item-header table is made up of two fields, (i) item name and (ii) head of node-link (a pointer directing to the first node in FP tree carrying the item-name).

*(Above details and algos are referred from [4])*

### 3.5. HORIZONTAL DESIGN

Horizontal design is frequently used, it relates and integrates every item of the common transition with respect to transaction id. In this technique the primary key of the target dataset is the id of the transaction. The most important property of the horizontal design is the integration of all items of the same transaction; this design also experience some restrictions like unnecessary processing as items don't have index created upon them. [5].

X#	Items				
X1	Z	Y	X	W	V
X2	Z	V	X	S	T
X3	Y	X	W	Z	V
X4	U	Z	S	T	J
X5	Z	Y	X	V	I
X6	K	Z	V	I	X
X7	Z	S	V	T	I
X8	K	L	M	N	O
X9	L	R	Q	Z	O
X10	P	N	Y	Z	M

X#	Items				
X1	Z	Y	X	W	V
X2	Z	V	X	S	T
X3	Y	X	W	Z	V
X4	U	Z	S	T	J
X5	Z	Y	X	V	I
X6	K	Z	V	I	X
X7	Z	S	V	T	I
X8	K	L	M	N	O
X9	L	R	Q	Z	O
X10	P	N	Y	Z	M

X#	Items		
X1	Z	X	V
X2	Z	V	X
X3	X	Z	V
X4	Z		
X5	Z	X	V
X6	Z	V	X
X7	Z	V	
X9	Z		
X10	Z		

**Figure 7 : Horizontal Layout [5]**

(Above algo and details are referred from [5])

### 3.6. VERTICAL DESIGN

In case of vertical design, transaction are integrated and combined with respect to the item. In this technique the id of each trace is the item. In this technique the primary key of the table is the item, and each item's record contains the ids of the transactions in which this item occurs. One of the main advantages of this technique is, on items its acts as an index and decreases the probability of rescanning the complete dataset again and again. On the other hand, vertical design still requires the costly candidacy generation part. Also calculating the recurrences of item-sets turns into the tiresome job of interconnecting records of unrelated items of the nominee patterns. Mining exceptionally

huge datasets with this design becomes nearly impractical because of candidate generation and the additional processes attached with this approach. [5].

Item	Transactions									
	i	ii	iii	iv	v	vi	vii	ix	x	
Z	i	ii	iii	iv	v	vi	vii	ix	x	
Y	i	iii	v	x						
W	i	ii	iii	v	vi					
X	i	iii								
V	i	ii	iii	v	vi	vii				
U	iv									
T	ii	iv	vii							
S	ii	iv	vii							
I	v	vi	vii							
J	iv									
K	vi	viii								
L	viii	ix								
M	viii	x								
N	viii	x								
O	viii	ix								
P	x									
Q	ix									
R	ix									

**Figure 8 : Vertical Layout [5]**

(Above details and algos are referred from [5])

### 3.7. ABBM

This algo converts the transactional dataset into a Boolean matrix. As name of matrix explains, the Boolean matrix is of bit type and each cell of the matrix requires 1 bit to store its value in main memory. All transactions of the database containing a large number of items are mapped into the Boolean matrix. A very simple method is adopted to discover the frequent item-sets from the Boolean matrix. [6].

### 3.7.1. ALGO DETAILS

There are four modules of the ABBM and are given below. These modules are presented step by step:

- Create a Boolean matrix from the dataset.
- Recurrent 1-itemset creation.
- Matrix pruning.
- Producing a list of recurrent  $j$ -item-sets  $L_j(j>1)$ .

#### 3.7.1.1. DATASET CONVERSION

There are 'i' items in each transaction, and there are 'j' transactions in the dataset 'k'. Suppose the set of items is  $i = \{i_1, i_2, i_3, \dots, i_n\}$  and the set of transactions is  $j = \{j_1, j_2, j_3, \dots, j_n\}$ . A boolean matrix having  $x$  rows and  $y$  columns is created, the matrix is represented as  $A_{ij}$ . Now the dataset is scanned, while reading the transaction  $J_n$ , if the item  $I_m$  (where  $1 \leq n \leq i$ ,  $1 \leq m \leq j$ ) then value of the cell  $A_{ij}$  is '1,' else value of the cell  $A_{ij}$  is '0.' This process continues until all the transactions are mapped into the matrix.

#### 3.7.1.2. RECURRENT 1-ITEM-SET PRODUCTION

The Boolean matrix  $A_i^*j$  is examined and recurrence count of every item is calculated. The recurrence count of  $I_j$ .supth of item  $I_j$  is the count of '1s' in  $j$ th column of the Boolean matrix  $A_i^*j$ . If  $I_j$ .supth is lesser than the min recurrence count  $\text{minsupth}$ , item-set  $\{I_j\}$  is not a recurrent 1-item-set and the  $j$ th column of the Boolean matrix  $A_i^*j$  will be eliminated from  $A_i^*j$ . Otherwise item-set  $\{I_j\}$  is the recurrent 1-item-set and is added to the list of recurrent 1-item-set  $L_1$ . The sum of the element values of each row is

recomputed, and according to Proposition 1, the rows whose sum of element values is smaller than 2 are deleted from this matrix.

### 3.7.1.3. PRUNNING THE BOOLEAN MATRIX

Pruning the Boolean matrix means removing several columns and rows from it. First, the column of the Boolean matrix is pruned according to Proposition 2. This is described in detail as: Let  $I'$  be the set of all items in frequent set  $L_{k-1}$ , where  $k > 2$ . Compute all  $|L_{k-1}(j)|$  where  $j \in I'$ , and delete the column of correspondence item  $j$  if  $|L_{k-1}(j)|$  is smaller than  $k-1$ . Second, recompute the sum of the element values in each row in Boolean matrix. According to Proposition 1, those rows of eliminated from the Boolean matrix whose sum column count is smaller than  $k$ .

### 3.7.1.4. PRODUCTION OF RECURRENT K-ITEM-SETS

Recurrent  $k$ -item-sets are discovered only by “and” relational calculus, which is carried out for the  $k$ -vectors combination. If the Boolean matrix  $A_{p \times q}$  has  $q$  columns where  $2 < q \leq n$  and  $\minsup_{th} \leq p \leq n$ ,  $k \leq q \leq c$ , combinations of  $k$ -vectors will be produced. The ‘and’ relational calculus is for each combination of  $k$ -vectors. If the sum of element values in “and” calculation result is not smaller than the min support number  $\minsup_{th}$ , the  $k$ -item-sets corresponding to this combination of  $k$ -vectors are the frequent  $k$ -item-sets and are added to the set of frequent  $k$ -item-sets  $L_k$ .



Input: The transaction database D, the minimum support number minsupth  
Output: the set of frequent itemsets L

1. Transform the transaction database D into the Boolean matrix A;
2. For each column  $A_i$  of A
3.   If  $\text{sum}(A_i) \geq \text{minsupth}$  //  $\text{sum}(A_i)$  is the sum of the element value of  $A_i$
4.      $L_1 \leftarrow A_i$ ;
5.   Else delete  $A_i$  from A;
6. For each row  $A_m$  of A
7.   If  $\text{sum}(A_m) < 2$
8.     Delete  $A_m$  from A;
9. For ( $k=2; |L_{k-1}| > k-1; k++$ )
10. {
11.   Produce k-vectors combination for all columns of A;
12.   For each k-vectors combination  $\{A_{i_1}, A_{i_2}, \dots, A_{i_k}\}$
13.   {
14.      $B = A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_k}$ ;
15.     If  $\text{sum}(B) \geq \text{minsupth}$
16.        $L_k \leftarrow \{I_{i_1}, I_{i_2}, \dots, I_{i_k}\}$ ; //  $\{I_{i_1}, I_{i_2}, \dots, I_{i_k}\}$  is the itemsets according to  $\{A_{i_1}, A_{i_2}, \dots, A_{i_k}\}$
17.   }
18. For each item  $I_i$  in  $L_k$
19.   If  $|L_k(I_i)| < k$
20.     Delete the column  $A_i$  according to item  $I_i$  from A;
21. For each row  $A_m$  of A
22.   If  $\text{sum}(A_m) < k+1$
23.     Delete  $A_m$  from A;
24.    $k=k+1$
25. };
26. Return  $L=L_1 \cup L_2 \cup \dots \cup L_k$ ;

**Figure 9 : Algo to generate Frequent Item-set [6]**

(Above algo and details are referred from [6])

### 3.7.1.5. EXAMPLE

This section describes a sample execution of the ABBM algo. The sample transaction data of the transaction database D is shown in Table 1; the min support is 0.4;  $i=5$  is the total count of items, and  $j=5$  is the cardinality of the transactional dataset. Therefore, the min support number minsupth=2. The transaction database D is converted into the Boolean matrix  $A_{5 \times 5}$ :

TID	Itemsets
T1	I1,I4
T2	I2,I3,I5
T3	I1,I2,I3,I5
T4	I2,I5
T5	I1,I2,I3

	I1	I2	I3	I4	I5
T1	1	0	0	1	0
T2	0	1	1	0	1
T3	1	1	1	0	1
T4	0	1	0	0	1
T5	1	1	1	0	0

**Figure 10 : Transactional dataset and its Boolean Matrix [6]**

(Above algo and details are referred from [6])

We compute the sum of the element values of each column in Boolean matrix  $A_{5 \times 5}$  and the set of frequent 1-item-set is:

$$L1 = \{\{I1\}, \{I2\}, \{I3\}, \{I4\}\}$$

The fourth column of the Boolean matrix  $A_{5 \times 5}$  is removed because the support number of item I4 is lesser than the min support number 2. We then calculate the sum of the element values of each row in Boolean matrix and removed all rows where the sum of the element values is smaller than 2. Finally, the Boolean matrix  $A_{4 \times 4}$  is produced.

The process of 2-supports is executed for the all columns of the Boolean matrix  $A_{4 \times 4}$  and the set of recurrent 2-item-set is:

$$L2 = \{\{I1,I2\}, \{I1,I3\}, \{I2,I3\}, \{I2,I5\}, \{I3,I5\}\}$$

In pruning the Boolean matrix  $A_{4 \times 4}$  by the set of frequent 2-item-sets  $L_2$ , the third row of the Boolean matrix  $A_{4 \times 4}$  is deleted because sum of its element values is smaller than 3. Finally, the Boolean matrix  $A_{3 \times 4}$  is generated.

The process of 3-supports is executed for all columns of the Boolean matrix  $A_{3 \times 4}$  and the set of recurrent 3-item-set is:

$$L_3 = \{\{I_1, I_2, I_3\}, \{I_2, I_3, I_5\}\}$$

According to Proposition 3, the execution of ABBM algo is exited because there are two recurrent 3-item-sets in set of recurrent 3-item-set  $L_3$ .

(Above algo and details are referred from [6])

### 3.8. INVERTED MATRIX

The Inverted Matrix is the hybrid of horizontal and vertical approach; it uses the advantages of both the techniques and tries to eliminate the disadvantages and limitations of both the techniques up to the maximum. [5].

Item	Frequency	Item	Frequency	Item	Frequency
P	1	F	1	Q	1
R	1	J	1	O	2
D	2	K	2	L	2
M	2	N	2	I	3
G	3	H	3	B	4
C	5	E	6	A	9

Figure 11 : Frequency of Items [5]

(Above algo and details are referred from [5])

loc	Index	Transactional Array								
		1	2	3	4	5	6	7	8	9
1	(P,1)	(10,2)								
2	(F,1)	(5,1)								
3	(Q,1)	(4,1)								
4	(R,1)	(6,2)								
5	(J,1)	(13,2)								
6	(O,2)	(8,2)	(9,2)							
7	(D,2)	(15,1)	(15,2)							
8	(K,2)	(12,2)	(9,1)							
9	(L,2)	(10,1)	(18,7)							
10	(M,2)	(11,1)	(11,2)							
11	(N,2)	( $\phi,\phi$ )	(15,4)							
12	(I,3)	(15,3)	(16,5)	(13,3)						
13	(G,3)	(14,1)	(14,2)	(14,3)						
14	(H,3)	(16,2)	(17,4)	(17,6)						
15	(B,4)	(16,1)	(16,3)	(16,4)	(18,9)					
16	(C,5)	(17,1)	(17,2)	(17,3)	(17,4)	(17,5)				
17	(E,6)	(18,1)	(18,2)	(18,3)	(18,5)	(18,6)	(18,7)			
18	(A,9)	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )	( $\phi,\phi$ )

**Figure 12 : Inverted Matrix [5]**

(Above algo and details are referred from [5])

The Inverted Matrix is the hybrid of horizontal and vertical approaches; it uses the advantages of both the techniques and tries to eliminate the disadvantages and limitations of both the techniques up to the maximum. The basis idea behind this approach is to integrate each item with all the transactions in which it occurs and to link each transaction with its related items. Alike to the vertical design, the item is the key of each record in this design. This technique is different from the vertical design in a sense that each cell of the matrix is not a single value, but it is data structure made up of transaction id and the pointer, the pointer points to the position of the subsequent item on the same transaction. [5].

**Algorithm 1: Inverted Matrix (IM) Construction**  
**Input :** Transactional Database ( $\mathcal{D}$ )  
**Output :** Disk Based Inverted Matrix

**Method :**

**Pass I**

1. Scan  $\mathcal{D}$  to identify unique items with their frequencies.
2. Sort the items in ascending order of their frequency.
3. Create the index part of the IM using the sorted list.

**Pass II**

1. While there is a transaction  $T$  in the database ( $\mathcal{D}$ ) do
  - 1.1 Sort the items in the transaction  $T$  into ascending order according to their frequency
  - 1.2 while there are items  $s_i$  in the transaction do
    - 1.2.1 Add an entry in its corresponding transactional array row with 2- parameters
      - (A) Location in index part of the IM of the next item  $s_{i+1}$  in  $T$  *null* if  $s_{i+1}$  does not exist.
      - (B) Location of the next empty slot in the transactional array row of  $s_{i+1}$ , *null* if  $s_{i+1}$  does not exist.
  - 1.3 Goto 1.2
2. Goto 1

**Figure 13 : Algo to create Inverted Matrix [5]**

*(Above algo and details are referred from [5])*

Inverted Matrix is developed in two steps, in first step dataset is scanned completely and recurrence of each item is calculated and then in ascending order items are ordered. In second step dataset is scanned again and ascending order items of each transaction are ordered and then the matrix is filled.

Developing the Inverted Matrix is understood to be preprocessing of the transactional dataset. For a given transactional dataset, it is constructed one time and for all.

**Algorithm 2: Creating and Mining COFI-Trees**  
**Input:** Inverted Matrix (IM) and a minimum support threshold  $\sigma$   
**Output:** Full set of frequent patterns

**Method:**

1. Frequency\_Location = Apply binary search on the index part of the IM to find the Location of the first frequent item based on  $\sigma$ .
2. While (Frequency\_Location < IM.Size) do
  - 2.1 A = Frequent item at location (Frequency\_Location)
  - 2.2 A\_Transactional = The Transactional array of item A
  - 2.3 Create a root node for the (A)-COFI-Tree with *frequency-count* and *participation-count* = 0
  - 2.4 Index\_Of\_TransactionalArray = 0
  - 2.5 While (Index\_Of\_TransactionalArray < Frequency of item A)
    - 2.5.1 B = item from Transactional array at location (Index\_Of\_TransactionalArray)
    - 2.5.2 Follow the chain of item B to produce sub-transaction C
    - 2.5.3 Items on C form a prefix of the (A)-COFI-Tree.
    - 2.5.4 If the prefix is new then
      - 2.5.4.1 Set *frequency-count* = 1 and *participation-count* = 0 for all nodes in the path
      - Else
      - 2.5.4.2 Adjust the *frequency-count* of the already exist part of the path.
    - 2.5.5 Adjust the pointers of the *Header list* if needed
    - 2.5.6 Increment Index\_Of\_TransactionalArray
    - 2.5.7 Goto 2.5
  - 2.6 MineCOFI-Tree (A)
  - 2.7 Release (A) COFI-Tree
  - 2.8 Increment Frequency\_Location //to build the next COFI-Tree
3. Goto 2

**Function: MineCOFI-Tree (A)**

1. nodeA = select\_next\_node //Selection of nodes will start with the node of most frequent item and following its chain, then the next less frequent item with its chain, until we reach the least frequent item in the *Header list* of the (A)-COFI-Tree
2. while there are still nodes do
  - 2.1 D = set of nodes from nodeA to the root
  - 2.2 F = *frequency-count-participation-count* of nodeA
  - 2.3 Generate all Candidate patterns X from items in D. Patterns that do not have A will be discarded
  - 2.4 Patterns in X that do not exist in the A-Candidate List will be added to it with frequency = F otherwise just increment their frequency with F
  - 2.5 Increment the value of *participation-count* by F for all items in D
  - 2.6 nodeA = select\_next\_node
  - 2.7 Goto 2
3. Based on support threshold  $\sigma$  remove non-frequent patterns from A Candidate List.

**Figure 14 : Algo to create Set of Frequent Patterns [5]**

(Above algo and details are referred from [5])

## **4. PROBLEM STATEMENT**

After introduction of Apriori algo [1], there have been continuous efforts to address the problems of repeated database scanning and large number of candidate item-sets generation. Apriori-like algos suffer from the above mentioned problems when database size (in terms of cardinality) is larger, and when there are so many items.

# **Proposed Solution**



## 5. PROPOSED ALGORITHM

The  $n \times n$  frequency matrix is built in a single database scan. Method iterates through the database starting from the first transaction to the last transaction. If there are three items in starting transaction then three rows and three columns are added to the empty matrix. Each item represents a row and a column. E.g. for the transactions given below:

### 1. Plums Lettuce, Tomatoes

- System read the transaction and it found three items in transaction. For each of the item a row and a column is added to the empty matrix.
- For plums row; frequencies of lettuce and tomatoes are incremented by 1.
- For lettuce row; frequencies of plums and tomatoes are incremented by 1.
- For tomatoes row; frequencies of plums and lettuce are incremented by 1.
- Frequencies of same items are not updated i.e. for plums frequency for plums will not be updated and is represented by X.
- For a transaction number of rows updated is equal to the number of items in transaction.

	Plums	Lettuce	Tomatoes
Plums	X	1	1
Lettuce	1	X	1
Tomatoes	1	1	X

**Figure 15 : Frequency Matrix after one transaction**

### 2. Celery, Confectionery

- System read the next transaction and it found two new items in transaction. For each of the new item a row and a column is added to the existing matrix.
- For celery row; frequency of confectionery is incremented by 1.
- For confectionery row; frequency of celery is incremented by 1.

- Frequencies of same row and column items are not updated.

	Plums	Lettuce	Tomatoes	Celery	Confectionery
Plums	X	1	1		
Lettuce	1	X	1		
Tomatoes	1	1	X		
Celery				X	1
Confectionery				1	X

**Figure 16 : Frequency Matrix after two transactions**

### 3. Apples, Carrots, Tomatoes, Potatoes, Confectionery

(Same process as above for transaction # 2)

	Plums	Lettuce	Toma...	Celery	Confe...	Apples	Carrots	Potat...
Plums	X	1	1					
Lettuce	1	X	1					
Toma...	1	1	X					
Celery				X	1			
Conf...			1	1	X	1	1	1
Apples			1		1	X	1	1
Carrots			1		1	1	X	1
Potat...			1		1	1	1	X

**Figure 17 : Frequency Matrix after three transactions**

**4. Apples, Oranges, Lettuce, Tomatoes, Confectionery**

- System read the next transaction and it found one new items in transaction. For the new item a row and a column is added to the existing matrix.
- Frequencies of items are incremented by the same process described for transaction # 2.

	PL	LE	TO	CE	CO	AP	CR	PO	OR
PL	X	1	1						
LE	1	X	2		1	1			1
TO	1	2	X		2	2	1	1	1
CE				X	1				
CO		1	2	1	X	2	1	1	1
AP		1	2		2	X	1	1	1
CR			1		1	1	X	1	
PO			1		1	1	1	X	
OR		1	1		1	1			X

**Figure 18 : Frequency Matrix after four Transactions****5. Confectionery****6. Peach, Oranges, Celery, Potatoes****7. Beans, Lettuce, Tomatoes****8. Oranges, Lettuce, Carrots, Tomatoes, Confectionery****9. Apples, Bananas, Plums, Carrots, Tomatoes, Onion, Confectionery**

(5, 6, 7, 8, 9 are the transactions)

## 10. Apples, Potatoes

	PL	LE	TO	CE	CO	AP	CR	PO	OR	PE	BE	BN	ON
PL	X	1	2		1	1	1					1	1
LE	1	X	4		2	1	1		2		1		
TO	2	4	X		4	3	3	1	2		1	1	1
CE				X	1			1	1	1			
CO	1	2	4	1	X	3	3	1	2			1	1
AP	1	1	3		3	X	2	2	1			1	1
CR	1	1	2		3	2	X	1	1			1	1
PO			1		1	2	1	X					
OR		2	2	1	2	1	1	1	X	1			
PE				1				1	1	X			
BE		1	1								X		
BN	1		1			1	1					X	1
ON	1		1		1	1	1					1	X

Figure 19 : Frequency Matrix for 10 transactions

A hash table will contain information about each item and its index in ' $N \times N$  Frequency Matrix'.

Association rules mining using  $n \times n$  matrix is an efficient process. Intersection point of each item in matrix can be used to keep information like its support. Each cell of matrix presents a 2-item-set association. N-item-set association can be mining with the help of a simple mining algo

## **6. SCOPE**

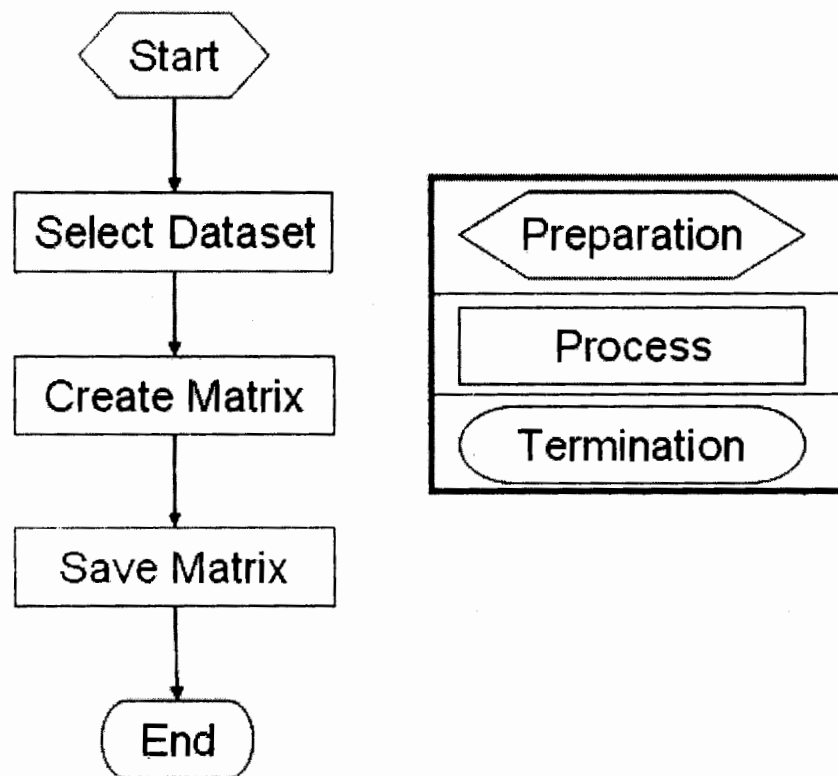
Algo is designed for maximum of 15000 items on a machine having 1 GB ram and provides support for millions of records (transactions). As matrix is of 32-bit Integer type it supports a database of 2,147,483,647 transactions and for maximum of 15000 items. Data entry operations for the system and data management are not included in this solution. Algo will only read existing data for analysis and will only update matrix database.

# **Methodology**

## 7. MODULES

There are two main modules of the proposed solution. First module is related to the generation of  $N \times N$  frequency matrix. Second module is related to the discovery of frequent item-sets

### 7.1. GENERATION OF $N \times N$ FREQUENCY MATRIX



**Figure 20 : Frequency Matrix Creation Steps**

In this module each transaction of the database is read one by one and items in each transaction is used to create or append the frequency matrix. At the start of this module, there are no rows and no columns in matrix. When the first transaction is read, the items in transaction are added one by one on rows and columns of the frequency matrix. During reading the second transaction matrix is updated, if items in first transaction and the second transaction are same then the frequency of the items are updated in matrix otherwise a new row and column for the new items is added to the matrix. This process continues until the complete database is read.

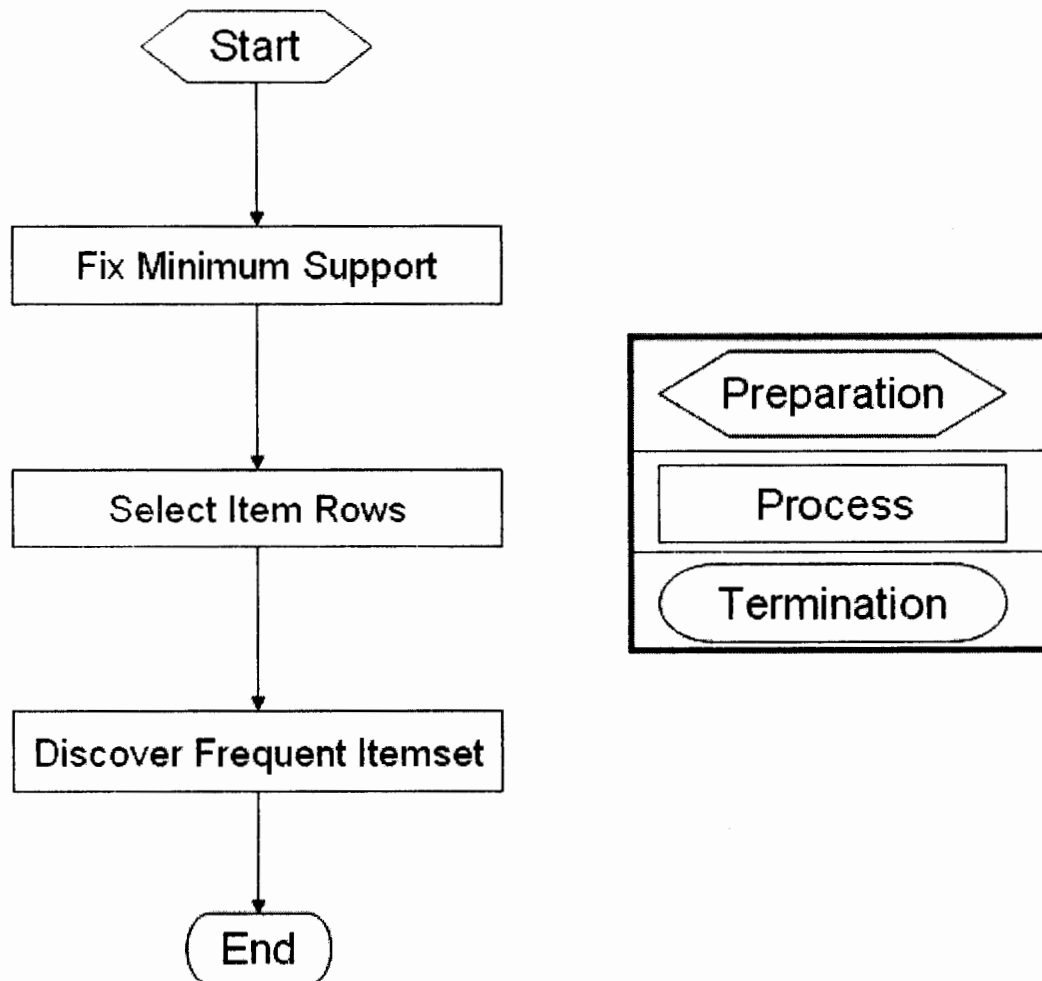
```

// Summary
// Read Transaction one by one,
// Create or append Frequency Matrix
// Summary
private void GenerateMatrix()
{
    try
    {
        Matrix.CreateMatrix.iGlobalMatrix = null;
        objOld = new CreateMatrix();
        String[] strItems = null;
        String strQuery = "Select ItemsNos from Details";
        IDataReader dataReader = db.ExecuteReader(strQuery);
        while (dataReader.Read())
        {
            Object obj = dataReader["ItemsNos"];
            if (obj is DBNull)
            {
                continue;
            }
            else if (obj.ToString().Trim() == String.Empty)
            {
                continue;
            }
            else
            {
                //Split items in transactions.
                strItems = obj.ToString().Split(
                    new char[] { ',' },
                    StringSplitOptions.RemoveEmptyEntries);
                SetValuesInHashTable(strItems);
                CreateMatrix(htItemIndex, strItems);
            }
        }
        dataReader.Close();
        objNew.PrintMatrix(
            ref objNew, ref dgvMatrix, ref htItemIndex);
    }
    catch (Exception ex)
    { throw new Exception(ex.Message); }
}

```



## 7.2. DISCOVERY OF FREQUENT ITEM-SETS



**Figure 21 : Discovery of Frequency Itemsets**

Frequent item-sets are discovered by exploring the frequency matrix. As described above items are mapped on the rows and columns of the frequency matrix and each cell determines the frequency of the items. A hashtable is used to store the maximum frequency of each item. Rows for items are fetched from the frequency matrix for the

items that satisfy the min support (User Input). Then a recursive method is used to discover the frequent item-sets from the fetched rows.

```

// Summary
// This method is used to Generate Frequent Item-sets
// Summary
private void GenerateFrequentItem-sets()
{
    try
    {
        ClearHashTable();
        Hashtable htItemsIndex = new Hashtable();
        htItemsIndex = objFrequentItem-
sets.GetItemsIndex(htItemsIndex);
        htFrequentItems = objFrequentItem-sets.GetFrequentItem-sets(
            htFrequentItems, htItemsIndex, txtMinSupport.Text);
        dtFrequentItems = objFrequentItem-sets.FormatHashtable(
            htFrequentItems, out htFreqItemsTemp, out alTemp);

        //Print Table

        DataTable dt = new DataTable();
        dt.Columns.Add("Item-sets #");
        dt.Columns.Add("Item-set Details");

        for (int i = 0; i < alTemp.Count; i++)
        {
            if (htFreqItemsTemp.ContainsKey(alTemp[i].ToString()))
            {
                str = String.Empty;
                DataRow dataRow = dt.NewRow();
                dataRow[0] = alTemp[i];
                str = alTemp[i].ToString()+",";
                String strValues =
                htFreqItemsTemp[alTemp[i].ToString()].ToString();
                String[] strArrValues = strValues.Split(
                new Char[] { ',' },
                StringSplitOptions.RemoveEmptyEntries);
                str += strArrValues[0] +",";
                strValues = strValues.Replace(strArrValues[0], "");
                if (strArrValues.Length > 1)
                {
                    GetItem-set(strArrValues[0], strValues);
                    dataRow[1] = str.Remove(str.Length - 1, 1);
                    dt.Rows.Add(dataRow);
                    continue;
                }
            }
            else
            {
                dataRow[1] = alTemp[i]+",";
                htFreqItemsTemp[alTemp[i].ToString()].ToString();
                dt.Rows.Add(dataRow);
            }
        }
    }
}

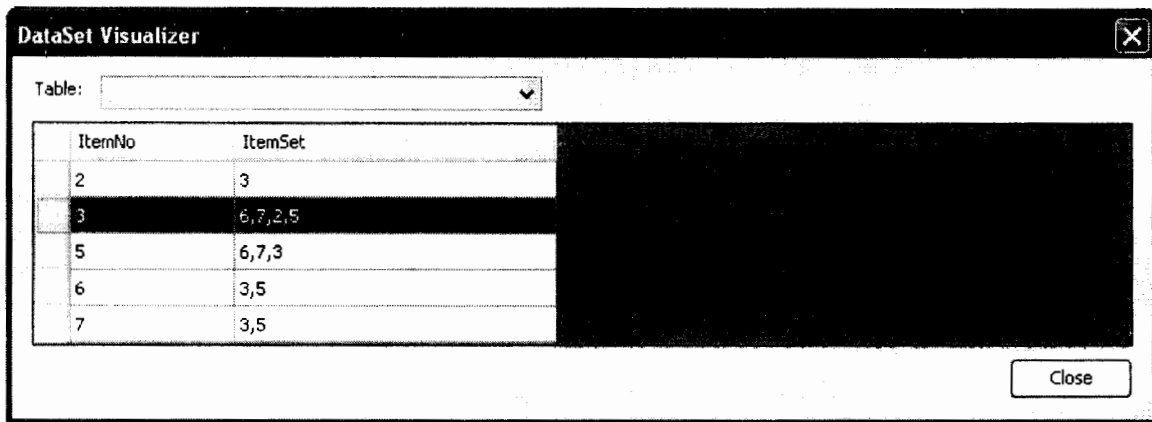
```

```

    }

    //Sort items in each Item-set
    dt = SortItems(dt);
    //Remove duplicate entries
    if (!bGridResults)
        dt = RemoveDuplicateEntries(dt);
    //Get name of items
    dt = objFrequentItem-sets.GetItemNames(dt);
    dgvFrequentItems.DataSource = dt.DefaultView;
}
catch (Exception ex)
{
    throw new Exception(ex.Message);
}
}

```



**Figure 22 : Intermediate Results for Frequency Itemsets**

Item	Item-set
Lettuce	Tomatoes
Tomatoes	Lettuce, Confectionery, Apples, Carrots
Confectionery	Tomatoes, Apples, Carrots
Apples	Tomatoes, Confectionery
Carrots	Tomatoes, Confectionery

**Figure 23 : Itemsets from NxN Frequency Matrix**

### 7.3. Working of Recursive Method

```
GetItemSet("TO",["LE","CO","AP","CR"])
```

For Tomatoes (TO) all the items (LE, CO, AP, CR) occurs in frequency matrix. TO will be then added to the list.

Method is called recursively for the LE and [CO, AP, CR].

```
GetItemSet("LE",["CO","AP","CR"])
```

In frequency matrix given items (CO, AP, CR) don't occurs for Lettuce (LE). LE will be discarded from the list.

Method will be called recursively for CO and the remaining items [AP, CR].

```
GetItemSet("CO",["AP","CR"])
```

In frequency matrix given items (AP, CR) occurs for Confectionery CO. CO will be added to the list

Method will be called for AP and the remaining item(s) [CR].

```
GetItemSet("AP",["CR"])
```

In frequency matrix given items (CR) don't occurs for Apples AP. CR will be discarded to the list and AP will be added to the list as the last item.

Recursive method will end here.

TO	CO	AP
List generated for frequent Item		

# **Design and Implementation**

## 8. DESIGN AND IMPLEMENTATION

This chapter discusses in detail the design and methodology adopted for the application development. Further hardware and software requirements are also presented in this chapter.

### 8.1. FREQUENCY MATRIX GENERATION

In this part the methods used to create frequency matrix generation are discussed in detail along with the code written in Visual C# 2005.

#### 8.1.1. SetValuesInHashTable ()

```
/// <summary>
/// This method take an array of string type,
/// Each item of array is an ItemNo,
/// If ItemNo dont exists in hash table it will be added to hash table,
/// position in hash table will be the index of item.
/// This method also add items to a Hashtable which contains
/// frequencies of each item.
/// </summary>
/// <param name="strItems"></param>
private void SetValuesInHashTable(string[] strItems)
{
    try
    {
        foreach (String strItemNo in strItems)
        {
            if (htItemIndex.ContainsKey(strItemNo))
            {
                continue;
            }
            else
            {
                htItemIndex.Add(strItemNo, htItemIndex.Count);
                Matrix.CreateMatrix.htItemsFrequency.Add(strItemNo, 0);
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}
```

### 8.1.2. CreateMatrix ()

```

/// <summary>
/// This method is used to create and update Matrix for each
/// transaction.
/// A hashtable containing items and items indexes is passed,
/// String[] contains items of the transaction.
/// </summary>
/// <param name="htItemIndex"></param>
/// <param name="strItemNos"></param>
private void CreateMatrix(Hashtable htItemIndex, String[] strItemNos)
{
    try
    {
        //Execute only for 1st transaction.
        if (objOld.iMatrix.GetLength(0) == 0)
        {
            objOld = null;
            objOld = new CreateMatrix(htItemIndex.Count);
            objOld.SetFrequencies(htItemIndex, strItemNos, ref objOld);
            Matrix.CreateMatrix.iGlobalMatrix =
            Matrix.CreateMatrix.InitializeGlobalMatrix(
                htItemIndex.Count);
            Matrix.CreateMatrix.iGlobalMatrix = objOld.CopyMatrix(
                ref objOld, Matrix.CreateMatrix.iGlobalMatrix);
        }
        else if (objOld.iMatrix.GetLength(0) < 10000)
        {
            objNew = new CreateMatrix(htItemIndex.Count);
            objNew = objNew.CopyMatrix(
                Matrix.CreateMatrix.iGlobalMatrix, objNew);
            objNew.SetFrequencies(htItemIndex, strItemNos, ref objNew);
            Matrix.CreateMatrix.iGlobalMatrix =
            Matrix.CreateMatrix.InitializeGlobalMatrix(
                htItemIndex.Count);
            Matrix.CreateMatrix.iGlobalMatrix = objNew.CopyMatrix(
                ref objNew, Matrix.CreateMatrix.iGlobalMatrix);
        }
        else
        {
            //Create XML
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}

```

### 8.1.3. SetFrequencies ()

```

/// <summary>
/// This method is used to SetFrequencies in Matrix.
/// Hashtable is used to find index of items in strItemNos array.
/// objCreateMatrix is the object that contains the Matrix
/// </summary>
/// <param name="htItemIndex"></param>
/// <param name="strItemNos"></param>
/// <param name="objCreateMatrix"></param>
public void SetFrequencies(Hashtable htItemIndex, string[] strItemNos,
ref CreateMatrix objCreateMatrix)
{
    try
    {
        Boolean bIncremented;
        Int16[] iIndexes = new Int16[strItemNos.Length];
        Int16 i, j;
        i = j = 0;
        for (i = 0; i < strItemNos.Length; i++)
        {
            iIndexes[i] = Convert.ToInt16(htItemIndex[strItemNos[i]]);
            Matrix.CreateMatrix.htItemsFrequency[strItemNos[i]] =
            Convert.ToInt32(
            Matrix.CreateMatrix.htItemsFrequency[strItemNos[i]] + 1;
            //Set Frequency of each item in transaction.
        }
        for (i = 0; i < iIndexes.Length; i++)
        {
            bIncremented = false;
            for (j = 0; j < iIndexes.Length; j++)
            {
                if (iIndexes[i] == iIndexes[j])
                {
                    if (!bIncremented)
                    {
                        objCreateMatrix.iMatrix[iIndexes[i], iIndexes[j]]++;
                        bIncremented = true;
                    }
                }
                else
                {
                    objCreateMatrix.iMatrix[iIndexes[i], iIndexes[j]] += 1;
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}

```



### 8.1.4. CopyMatrix ()

```
/// <summary>
/// This method copies an Int32[,] array to a CreateMatrix
/// object. It then returns the CreateMatrix object in
/// which Int32[,] array is copied.
/// </summary>
/// <param name="objCopyFrom"></param>
/// <param name="objCopyTo"></param>
/// <returns></returns>
public CreateMatrix CopyMatrix(Int32[,] objCopyFrom,
CreateMatrix objCopyTo)
{
    try
    {
        for (int i = 0; i < objCopyFrom.GetLength(0); i++)
            for (int j = 0; j < objCopyFrom.GetLength(1); j++)
                objCopyTo.iMatrix[i, j] = objCopyFrom[i, j];
        return objCopyTo;
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
        return objCopyTo;
    }
}
```

## 8.2. DISCOVERY OF FREQUENT ITEM-SETS

In this part the methods which are used to discover frequent item-set from the frequency matrix are discussed in detail along with the code written in Visual C# 2005.

### 8.2.1. GetItemsIndex ()

```
/// <summary>
/// This method is used to get items and index from database
/// </summary>
/// <returns></returns>
public Hashtable GetItemsIndex(Hashtable htItemsIndex)
{
    try
    {
        dsTemp = db.GetDataSet(strItemsQuery);
        if (dsTemp.Tables[0].Rows.Count > 0)
        {
            for (int i = 0; i < dsTemp.Tables[0].Rows.Count; i++)
            {
                DataRow dr = dsTemp.Tables[0].Rows[i];
                if (htItemsIndex.Contains(dr[0]))
                    continue;
                else
                    htItemsIndex.Add(dr[0], dr[1]);
            }
        }
    }
    catch (Exception ex)
    { throw new Exception(ex.Message); }
    return htItemsIndex;
}
```

### 8.2.2. GetFrequentItem-sets ()

```

/// <summary>
/// To get a hash table of frequent item-sets
/// </summary>
/// <param name="iMinSupp"></param>
/// <returns></returns>
public Hashtable GetFrequentItem-sets(
    Hashtable htFrequentItem-sets,
    Hashtable htItemsIndex,
    String strMinSupport)
{
    try
    {
        Int32 iMinSupp = 0;

        if (!String.IsNullOrEmpty(strMinSupport))
            iMinSupp = Convert.ToInt32(
                ConfigurationSettings.AppSettings["MinSupp"]);
        else
            iMinSupp = Convert.ToInt32(strMinSupport);

        String strTempQuery = strQuery.Replace(
            "****", ""+iMinSupp+"");

        dsTemp = db.GetDataSet(strTempQuery);
        strTempQuery = "";

        for (int i = 0; i < dsTemp.Tables[0].Rows.Count; i++)
        {
            DataRow dr = dsTemp.Tables[0].Rows[i];
            if (
                htFrequentItem-sets.Contains(htItemsIndex[dr[0]]))
            {
                htFrequentItem-sets[htItemsIndex[dr[0]]] =
htFrequentItem-sets[htItemsIndex[dr[0]]].ToString() + dr[4]+", ";
            }
            else
            {
                htFrequentItem-sets.Add(
                    htItemsIndex[dr[0]], dr[4]+", ");
                aItems.Add(htItemsIndex[dr[0]]);
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
    return htFrequentItem-sets;
}

```

### 8.2.3. FormatHashTable ()

```

/// <summary>
/// Method used to create a table that contains frequent items
/// </summary>
/// <param name="htFrequentItems"></param>
/// <returns></returns>
public DataTable FormatHashtable(Hashtable htFrequentItems,
    out Hashtable htFreqItemsTemp, out ArrayList alTemp)
{
    try
    {
        htFreqItemsTemp = new Hashtable();

        if (!dtFrequentItems.Columns.Contains("ItemNo"))
            dtFrequentItems.Columns.Add("ItemNo");
        if (!dtFrequentItems.Columns.Contains("Item-set"))
            dtFrequentItems.Columns.Add("Item-set");

        for (int i = 0; i < aItems.Count; i++)
        {
            DataRow dr = dtFrequentItems.NewRow();
            dr[0] = aItems[i];
            String strTemp = "";
            strTemp = htFrequentItems[aItems[i]].ToString();
            strTemp = strTemp.Trim();
            strTemp = strTemp.Replace(aItems[i] + ",", "");
            if (strTemp.Length > 0)
            {
                strTemp = strTemp.Remove(
                    strTemp.Length - 1, 1);
                dr[1] = strTemp;
                dtFrequentItems.Rows.Add(dr);
                if (!htFreqItemsTemp.Contains(dr[0]))
                    htFreqItemsTemp.Add(dr[0], dr[1]);
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
    alTemp = aItems;
    return dtFrequentItems;
}

```

**8.2.4. GetItem-set ()**

```

/// <summary>
/// Recursive method to generate Item-set
/// </summary>
/// <param name="ItemNo"></param>
/// <param name="strItemsValue"></param>
private void GetItem-set(String strItemNo, String
strItemsValue)
{
    try
    {
        //if (strItemsValue.Trim().Equals(""))
        //    bContinue = false;
        String strValues = "";
        strValues = htFreqItemsTemp[strItemNo].ToString();
        String[] strArrTemp1 = strItemsValue.Split(
            new Char[] { ',' },
            StringSplitOptions.RemoveEmptyEntries);
        String[] strArrTemp2 = strValues.Split(
            new Char[] { ',' },
            StringSplitOptions.RemoveEmptyEntries);
        for (int i = 0; i < strArrTemp1.Length; i++)
        {
            for (int j = 0; j < strArrTemp2.Length; j++)
            {
                if (strArrTemp1[i].Equals(strArrTemp2[j]))
                {
                    str += strArrTemp1[i] + ",";
                    GetItem-set(strArrTemp1[i],
                        GetRemainingItems(strArrTemp1, i + 1)
                    );
                    return;
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}

```

**8.2.5. GetRemainingItems ()**

```

private string GetRemainingItems(string[] strArrTemp1, int i)
{
    String strTemp = String.Empty;
    try
    {
        for (int k = i; k < strArrTemp1.Length; k++)
            strTemp += strArrTemp1[k] + ",";
    }
    catch (Exception ex)
    { throw new Exception(ex.Message); }
    return strTemp;
}

```

### 8.3. Software and Hardware Requirements

This application is implemented on a Pentium IV machine having 1GB ram running Windows XP operating system.

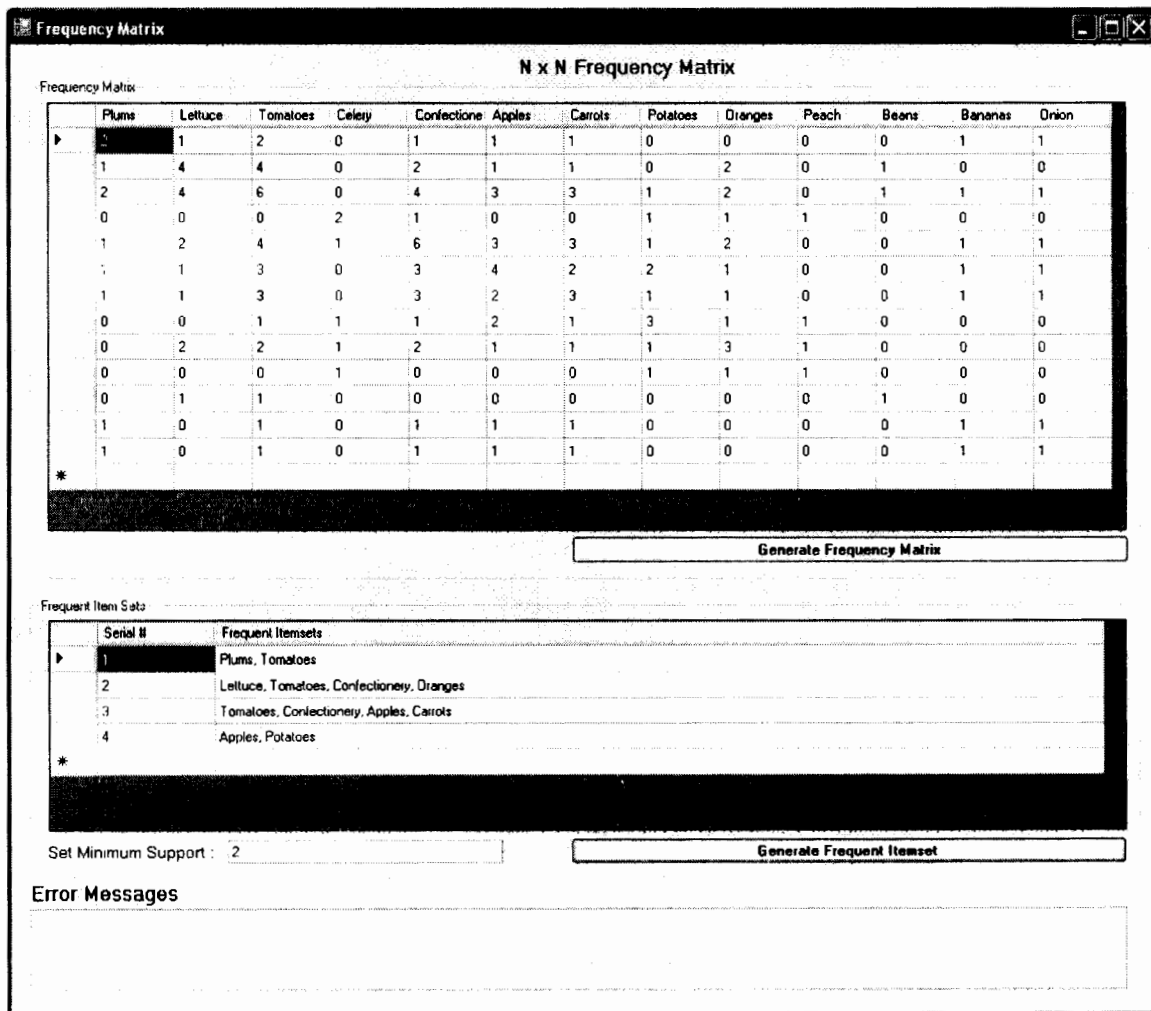


Figure 24 : Simulator view of Frequency Matrix

## 9. CONCLUSIONS

Algo proposed provides a significant decrease in processing and I/O cost. I/O cost is decreased because database is read once to create a frequency matrix (once the NxN frequency matrix is created it is stored in database for future usage and reference) and processing cost is decreased because the frequent item-set discovery method is directly proportional to the number of items that fulfills the min support criteria.

### 9.1. Example

	PL	LE	TO	CE	CO	AP	CR	PO	OR	PE	BE	BN	ON
PL	X	1	2		1	1	1					1	1
LE	1	X	4		2	1	1		2		1		
TO	2	4	X		4	3	3	1	2		1	1	1
CE				X	1			1	1	1			
CO	1	2	4	1	X	3	3	1	2			1	1
AP	1	1	3		3	X	2	2	1			1	1
CR	1	1	2		3	2	X	1	1			1	1
PO			1		1	2	1	X					
OR		2	2	1	2	1	1	1	X	1			
PE				1				1	1	X			
BE		1	1								X		
BN	1		1			1	1					X	1
ON	1		1		1	1	1					1	X

Figure 25 : Selecting Problem Area

## 9.2. Problem Area

	LE	TO	CO	AP	CR
LE	X	4	2	1	1
TO	4	X	4	3	3
CO	2	4	X	3	3
AP	1	3	3	X	2
CR	1	2	3	2	X

**Figure 26 : Final Problem Area**

### *Rows and columns picked from NxN Frequency Matrix*

Each item represents a row in  $N \times N$  frequency matrix and the each column is also represented by an item. Items discovery method only picks those items (rows) which fulfill the minimum criteria and also it picks only those columns of the rows which fulfill the minimum criteria. As it is described that each cell of the matrix represents frequency. In this way, our problem area gets small and processing cost and I/O cost decreases. Consider the following example for min support 3.



## 10. REFERENCES

- [1] Agrawal, R., Imienlinski, T., & Swami, A. (1993). Mining association between sets of items in massive database. In *Proceeding of the 1993ACM-SINMOD International Conference on Management of Data* (pp. 207-216). New York: ACM Press
- [2] Park, J. S., Chen, M. -S., Yu, P. S. (1995). An effective hash based algorithm for mining association rules. In *Proceeding of the ACM SIGMOD International Conference on Management of Data* (pp. 175-186). New York: ACM Press.
- [3] Brin, S., Motwami, R., Ullman, J. D., & Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. In *Proceeding of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1997)* (pp. 255-264). AZ: New York: ACM Press.
- [4] **Fast Discovery of Frequent Item-sets: a Cubic Structure-Based Approach** (Renata Ivancsy, Istvan Vajk 2004)
- [5] **Inverted Matrix: Efficient Discovery of Frequent Items in Large Dataset in Context of Interactive Mining** (Mohammad El-Hajj, Osmar R. Zaiane 2003)
- [6] **Hanbing Liu, Bashing Wang** An Association Rule Mining Algorithm Based on a Boolean Matrix *Data Science Journal, Volume 6, Supplement, 9 September 2007*
- [7] Pei, J., Han, J., & Mao, R. (2000). CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proceedings of the 2000 ACM-SIGMOD International Workshop on Data Mining and Knowledge Discovery (DMKD '00)* (pp. 21-30). New York: ACM Press
- [8] Agrawal, R., & Srikant, R (1994). Fast algorithm for mining association rules. In *Proceedings of the 20<sup>th</sup> International Conference on Very Large Database* (pp. 487-499). San Francisco: Morgan Kaufman.
- [9] **J. Han and M. Kamber. Data Mining: Concepts and Techniques.** Morgan Kaufman, San Francisco, CA, 2001.
- [10] **H. Huang, X. Wu, and R. Relue.** Association analysis with one scan of databases. In *IEEE International Conference on Data Mining* 9pp. 629-636). December 2002.

- [11] **J. Han, J. Pei, and Y. Yin.** Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.
  
- [12] **M. El-Hajj and O. R. Zaïane (2003)** Non Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations, *Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, Prague, Czech Republic.

## 11. GLOSSARY OF ACRONYMS

<b>Plums</b>	Plums, PL
<b>Lettuce</b>	Lettuce, LE
<b>Tomatoes</b>	Tomatoes, Toma., TO
<b>Celery</b>	Celery, CE
<b>Confectionery</b>	Confectionery, Conf., CO
<b>Apples</b>	Apples, AP
<b>Carrots</b>	Carrots, CR
<b>Potatoes</b>	Potat., PO
<b>Orange</b>	Orange, OR
<b>Peach</b>	Peach, PE
<b>Beans</b>	Beans, BE
<b>Bananas</b>	Bananas, BN
<b>Onion</b>	Onion, ON
<b>Algo</b>	Algorithm
<b>Algos</b>	Algorithms



