PRIVACY PRESERVING SEQUENTIAL PATTERN MINING OVER DISTRIBUTED PROGRESSIVE DATABASES

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

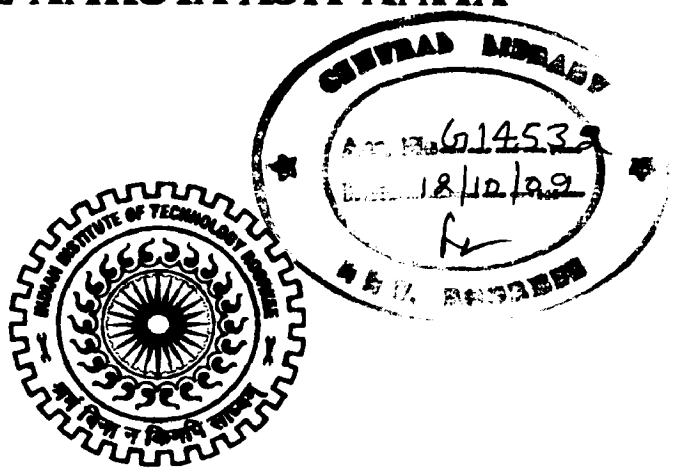
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

MHATRE AMRUTA AJIT ANITA



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)

JUNE, 2009

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled **"Privacy Preserving Sequential Pattern Mining over Distributed Progressive Databases"** towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology** in **Computer Science and Engineering** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from July 2008 to June 2009, under the guidance of **Dr. Durga Toshniwal, Assistant Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Date: 28/6/09
Place: Roorkee

(Mhatre Amruta Ajit Anita)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 28/6/09
Place: Roorkee

Durga Toshniwal
28/6/09.

(Dr Durga Toshniwal)
Assistant Professor
E & CE Dept.
IIT Roorkee

ACKNOWLEDGEMENTS

At the outset I would like to sincerely thank Indian Institute of Technology, Roorkee, for providing me an opportunity of being a part of this prestigious institution. It my proud privilege to express thanks and profound gratitude to my supervisor Dr. Durga Toshniwal, Assistant Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for her able guidance, regular source of encouragement and assistance throughout this dissertation work. I was able to complete this dissertation in time due to her constant motivation and support.

I am also grateful to Mr Nityam Parakh, Mr Tirumalesh C and Mr Khalil Sawant, my friends and colleagues for helping me with my Java related troubles. I also wish to thank all my friends for their valuable suggestions and timely help and support.

Finally, I would like to thank my parents for their love and trust in my abilities which has been a constant source of support and motivation for me.

MHATRE AMRUTA AJIT ANITA

ABSTRACT

Frequent Sequential Pattern Mining, commonly known as Sequential Pattern Mining is a data mining technique used to find interesting patterns in a large collection of data items. A common example of sequential pattern mining from market-basket databases is to track customer-buying patterns between the different items purchased by them. The discovery of such patterns can help retailers develop marketing strategies by gaining insight into frequently purchased items and their trends. The databases used for these purposes are progressive databases, which are a generalized model providing dynamic addition and deletion of data for efficient mining operations.

Sometimes a group of local market players may be interested in mining trends by pooling in their individual data. However the shared data may disclose some information which might be against the privacy policies of these collaborating parties or may be of strategic importance for some party. The need for a privacy preserving mechanism is thus felt to safeguard the sensitive information shared during the mining process.

In our dissertation work, we propose a set of algorithms for finding sequential patterns from distributed databases while preserving privacy. The work aims at maintaining the privacy of the data and patterns mined with minimal effect on accuracy of the results. In this work, the algorithms address all three types of fragmentation (viz. Vertical, Horizontal, Arbitrary).The proposed work of sequential pattern mining is applicable to progressive databases(special cases being static and incremental databases). In this work we use cryptographic and randomization techniques to achieve privacy preservation. The work also proposes an idea to suppress sensitive sequential pattern mining results. This proposition has generally been applied to the various kinds of distributed databases under study.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
Introduction

1.1 Introduction and Motivation

Data Mining, better known as knowledge discovery can be described as obtaining possibly unseen information from large data. Data mining may be better explained as processing data using sophisticated searching capabilities and statistical algorithms, to discover patterns and correlations in large preexisting databases. This process of analyzing data from different perspectives and summarizing it into useful information has a great application in the business world. For example, the mined knowledge can be used to increase revenue, cut costs or make certain marketing decisions. It allows users to analyze data from many different dimensions, categorize it, and summarize the relationships identified. Technically, *data mining can be defined as the process of "mining" knowledge from large amounts of data* [1].

The knowledge discovery process consists of five main stages as shown in Fig.1.1:

- Extract, transform, and load transaction data into the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Analyze the data by data mining algorithm to extract knowledge.
- Present the results in a useful format, such as a graphs or tables.
- Interpret the results.

The type of databases used for storing the collected data, depends upon the application of this data. Sometimes the presence of obsolete data in the data used for mining may result into erroneous results. Progressive databases provide a generalized solution to store all the collected data. These databases allow dynamic addition and deletion of data. This avoids re-mining of the whole data when new data is added. The static and incremental databases are special cases of such databases. As a result

progressive databases hence have a greater scope for application in real world applications.

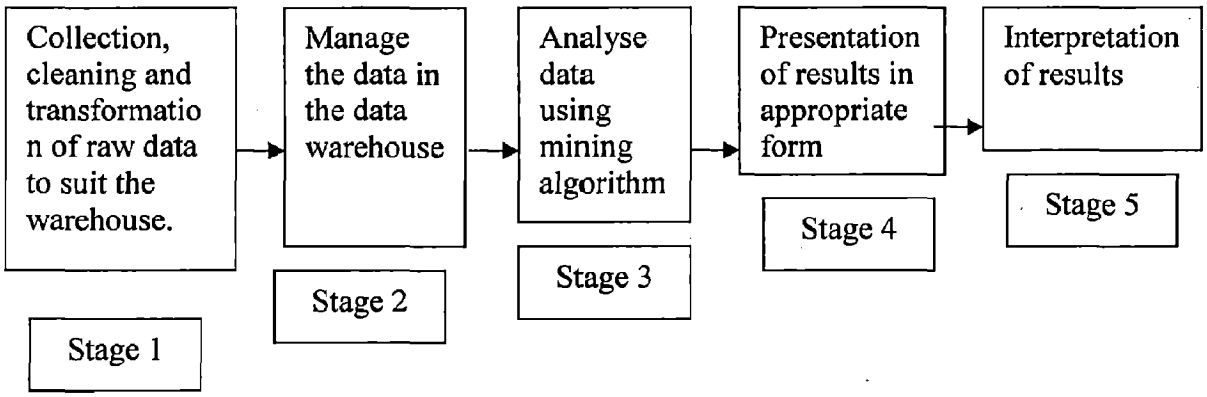


Figure 1.1: Stages in Knowledge Discovery Process

Security and privacy are other important issues for any data collection method .The collected data is shared and is intended to be used for making strategic decisions. Also, when data is mined for applications like customer profiling, medical analysis etc. large amounts of sensitive and private data about individuals needs to be gathered, stored and processed. Such situations make it necessary to maintain the confidentiality of the data in order to prevent its illegal access. Sometimes data mining results may also disclose some new implicit information about individuals which is against privacy policies. For these reasons, privacy preserving data mining is essentially a sought after field of research in data mining.

Algorithms are developed for modifying the original data in some way, so that private data and private knowledge remains private even after the data mining process. The main consideration in privacy preserving data mining is the preservation of sensitive raw data and sensitive knowledge that can be mined from the database with minimal effect on the results. For preserving the privacy, sensitive raw data like identifiers, names, addresses etc. must be modified or masked from the data to be mined, so that the data recipient may not be able to get any sensitive details from the data provider. Also the sensitive knowledge that can be mined from the database must be omitted; as such information can equally compromise the data privacy [2].

The main use of privacy preservation is in the field of distributed data mining, since this area requires extensive data transfer among data sharing parties. However in most of the cases, the sites may not want to disclose their individual data for the purpose of preserving the confidentiality. The data mining algorithms that mine knowledge while preserving privacy have thus been developed. Just as the data used for each mining technique varies from application to application so does the privacy preservation technique.

A typical example in data mining over distributed databases where privacy of data is of importance is in the field of medical research. Consider the case where a number of different hospitals wish to jointly mine their patient data, for the purpose of medical research. Privacy policy and law do not allow these hospitals from pooling their data or revealing it to each other since it could lead to the breach in confidentiality of patient records involved. Although hospitals are allowed to release data as long as the identifiers, such as name, address, and etc., are removed, it is not safe enough because the re-identification attack can link different public databases to relocate the original subjects. In order to pursue mutual gains and relieve the public from the privacy concerns, we need privacy-preserving distributed data mining protocols, which allow distributed data mining to take place while protecting privacy of the underlying distributed data.

Another example of such a scenario is the case of multiple competing supermarkets, each having a large set of data records of its customers' buying behaviors. These supermarkets may have a varied catalogue of thousands of products; and may want to conduct data mining on their joint data for mutual benefit. Since these companies are competitors in the market, they do not want to disclose their customers' information to each other. But they want to share the results obtained from this collaboration since it could bring them an advantage over other competitors.

1.2 Problem Statement

The problem statement for the proposed research work can be stated as: "*To design techniques for preserving the privacy of distributed progressive databases while mining sequential patterns.*"

The following aspects have been considered while designing these techniques:

- The proposed algorithm is designed for boolean data (case data is market-basket data).
- Each data item to be mined is associated with a timestamp, marking the time of its occurrence.
- The various types of fragmentation scenarios (horizontal, vertical, arbitrary) have been considered.

1.3 Organization of the Dissertation

The report is divided into six chapters including this introductory chapter. The rest of this thesis report is organized as follows:

Chapter 2 provides a brief description of literature review on sequential pattern mining. The other topics discussed include the various privacy preserving methods, the possible data fragmentation alternatives etc.

In Chapter 3 we provide a detailed description of proposed algorithms for preserving privacy while mining sequential patterns in distributed databases.

A brief description of the implementation details of the various modules in the proposed work has been discussed in Chapter 4.

Chapter 5 describes the results and includes a discussion on them. It also provides an analysis on important performance parameters of the proposed algorithm.

Chapter 6 concludes the dissertation and gives some suggestions for future work.

CHAPTER 2

Literature Review and Concepts

Sequential pattern mining aims at mining interesting patterns from a large set of data. Preserving privacy while mining data in a distributed scenario, requires data at multiple parties to be mined without compromising the privacy constraints of the data. Our work deals with privacy preserving sequential pattern mining in a distributed database scenario, various ideas, concepts and their related works needed to be studied while arriving at the proposed solution. This chapter presents a brief review of the studied literature. It includes works on sequential pattern mining, privacy preserving mining, various data fragmentation alternatives and so on.

2.1 Progressive Databases

There are two main types of databases: Static and Dynamic. A database that does not change over time is called a static database. Whereas the one in which the data changes with time are called dynamic databases. Dynamic databases can further be classified into incremental databases and progressive databases. Incremental databases assimilate data over time. Hence the size of the database increases with time. A progressive database however, is a kind of dynamic database in which new data can be added to the database and obsolete data can be removed simultaneously. Progressive databases can be called as a generalized model of static, dynamic and incremental databases. Such databases are the most up-to-date databases and have great applicability due to their flexibility of usage [3].

2.2 Distributed Databases

Data required for mining need not always be extracted from a single location. Data is generally distributed across multiple databases in order to harness the advantages of distributed processing. Sometimes databases may also be replicated in order to increase their efficiency. Some of the advantages of distributed databases are as follows:

- Reliability and availability in transaction processing.
- Modularity in operations
- Improved efficiency and flexibility

Splitting a central database in order to generate a distributed database system requires a logical scheme to act as a basis for data distribution. These partitioning schemes are also known as data fragmentation schemes. There are three main schemes for generating partitions from a central database viz.

- Horizontal fragmentation
- Vertical fragmentation
- Arbitrary fragmentation.

Each of these can be briefly described as following:

Horizontal partitioning, partitions a relation R along its tuples. Each horizontal fragment (*HF*) has a distinct subset of the tuples of the relation R [4].

Vertical partitioning of a relation R produces vertical fragments. Each of the vertical fragments (*VF*) contains a subset of R*'s* attributes as well the primary key of R [5].

Concept of arbitrarily partitioned data can be explained as one that generalizes both horizontally and vertically partitioned data. In arbitrarily partitioned data, different attributes for different items can be owned by either party. In arbitrarily partitioned data, there is not necessarily a defined scheme of how data is shared between the parties. Consider two parties A and B. For each tuple, A knows the values for a subset of the attributes, and B knows the values for the remaining attributes. Each tuple d_i is partitioned into disjoint subsets (*except for the primary key*) dA_i and dB_i *which are owned by parties* A and B. It is possible that a given tuple may be "completely owned" by B or by A [6].

2.3 Frequent Sequential Pattern Mining

Frequent sequential pattern mining, commonly known as sequential pattern mining, was first addressed in [7] by R. Agrawal and R. Srikant as the problem: "Given a database of sequences, where each sequence consists of a list of ordered item sets containing a set of different items, and a user defined minimum support threshold, sequential pattern mining finds all subsequences whose occurrence frequencies are no less than the threshold from the set of sequences". The concept of a sequence can be more formally described as:

Definition 1: Let $X = \{x_1, x_2, x_3 \ldots x_n\}$ be a set of different items. An element e, denoted by $< x_1, x_2, \ldots >$, is a subset of items belonging to X which appear at the same time. A sequence s, denoted by $< e_1 ; e_2 ; \ldots ; e_m >$, is an ordered list of elements. A sequence database DB contains a set of sequences, and $| DB |$ represents the number of sequences. in DB. A sequence $\alpha = < a_1 ; a_2 ; \ldots ; a_n >$ is a subsequence of another sequence $\beta =< b_1 ; b_2 ; \ldots ; b_m >$ if there exist a set of integers, $1 \leq i_1 \leq i_2 \leq i_n \leq m$, such that a_1 is a subset of b_{i1}; a_2 is a subset of b_{i2} ; .. . and a_n is a subset of b_{in} [3].

Sequential patterns are useful in businesses for shelf placement, promotions, targeted marketing, customer retention and many other tasks. R. Agrawal et al [7] by proposing two algorithms Aprioriall and Apriorisome, dealing with candidate generation for sequential pattern mining. SPADE [8] illustrated by Zaki, generated patterns by systematically searching the sequence lattice spanned by the subsequence relation. Other proposed algorithm in this area is SPAM [9], which works on searching a lexicographic sequence tree in depth-first manner using a vertical bitmap data layout.

However it was realized that static databases do not cater to many real world scenarios. Most real world scenarios require data to be constantly added, updated and pruned out of the database. This is because the existence of obsolete data in the database may result into sequences that currently may not be frequent. A model to mine sequences without the effects of presence of obsolete data was thus required.

2.4 Progressive Sequential Pattern Mining

Progressive sequential pattern mining extracts sequences over various time intervals. The time interval over which the patterns are mined at a particular timestamp is called Period of Interest (*POI*). The *POI* can be described as, a sliding window of user specified length, which keeps on advancing as time goes by giving the most recent sequential patterns. The formal definition of *POI* can be stated as:

Definition 2: Period of Interest (*POI*) is a sliding window, whose length is a user specified time interval. The sequences having elements whose timestamps fall into this period, *POI*, contribute to | *DB* | for current sequential patterns. On the other hand, the sequences having elements with timestamps older than *POI* are pruned away from the sequence database immediately and do not contribute to the | *DB* | thereafter [3]. This is illustrated in Fig.2.1.

The algorithm used to mine patterns from progressive databases is called *PISA* (Progressive mIning of Sequential pAtterns) (Fig 2.2). This algorithm uses the *M-ary Tree* structure to generate and maintain candidate patterns dynamically.

- ***The M-ary Tree (MTree)***

While mining for patterns the *PISA* algorithm keeps track of time and newly arriving elements in the database. At each timestamp the insertion of elements into the *M-ary*

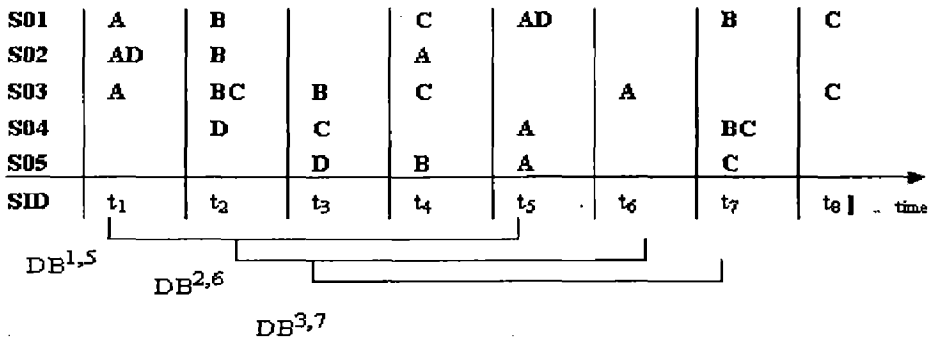


Figure 2.1: A Sample database

```
Algorithm PISA
var Mtree;
var CurrTime;
var eleSet;
while(there is new transaction)
    eleSet = read all ele at CurrTime;
insert(Tc,Mtree);
CurrTime++;
End
```

Figure 2.2: Algorithm PISA

tree results in an updated tree for the next timestamp. The algorithm traverses each node in the tree at time *t* in post order, deletes obsolete elements and updates the sequences according to data at current timestamp. The aim is to first insert new elements into existing candidate sequence and later identify any new frequent patterns. The procedure of manipulating candidate patterns in the *Mtree* is given in Fig.2.3.

- ***Mining Frequent Patterns from the M-ary tree***

Whenever a series of elements appear in a sequence (refer *SID* in Fig.2.1), path from the root is created labeled by the respective elements of the pattern with the corresponding sequence number on which this pattern occurred. This path from root to node called the candidate pattern. If a path already exists the concerned fields of the nodes are updated with the respective information. The timestamp for each node of the candidate sequential pattern is marked according to timestamp of the starting element of the candidate pattern. An obsolete element (i.e. element which lies out of the *POI*) and a node having no sequence numbers in its sequence list are pruned from the sequence list of the node and the M-ary tree respectively, ensuring only up to date candidate patterns in the M-ary tree [3].

After all the candidate sequential patterns are generated, the algorithm checks for the number of sequence IDs in a sequence list of all nodes. If the number of sequence IDs

```
Procedure insert (Tc , Mtree)
for(each node of Mtree in post order)
  if(node is Root node)
      for(ele of every seq in eleSet)
       for(all combination of elements in ele)
         if(element ==label of one of node.child)
          if(seq is in node.child.seq_list)
           update timestamp of seq to Tc;
           else
          create a new sequence with timestamp = Tc;
         else  // create a child node
         create child node with element, seq, timestamp = Tc;
    else // for a common node
      for(every seq in the seq_list)
       if(seq.timestamp <= Tc - POI)
         delete seq from seq_list and move to next seq;
       if(there is new ele of seq in eleSet)
        for(all combinations of elements in ele)
          if(element is not on the path from Root)
           if(element == label of one of node.child)
            if(seq is in node.child.seq_list)
              child.seq_list.seq.timestamp = node.seq.timestamp;
             else
          create new sequence with timestamp = seq.timestamp,
       else //create a child
      create a new child with element,seq, timestamp = seq, timestamp.
       if(seq_list.size ==0)
        delete this node and all of its children from its parent;
     if(seq_list.size>=support*sequence number)
      output labels of path from Root to this node as sequence pattern;
End
```

Figure 2.3: Procedure Insert

in a particular node is larger than minimum support multiplied number of sequences in the current *POI*, the path from the root till that node is considered as a frequent sequential pattern. The working of PISA for sample data (Fig.2.1) is given in Fig 2.4 [3].

2.5 Privacy Preserving Data Mining Techniques

Privacy preserving data mining techniques include a large spectrum of methods such as randomization, k-anonymity, l-diversity etc. Certain methods that were studied while designing of the proposed research work are briefly explained below:

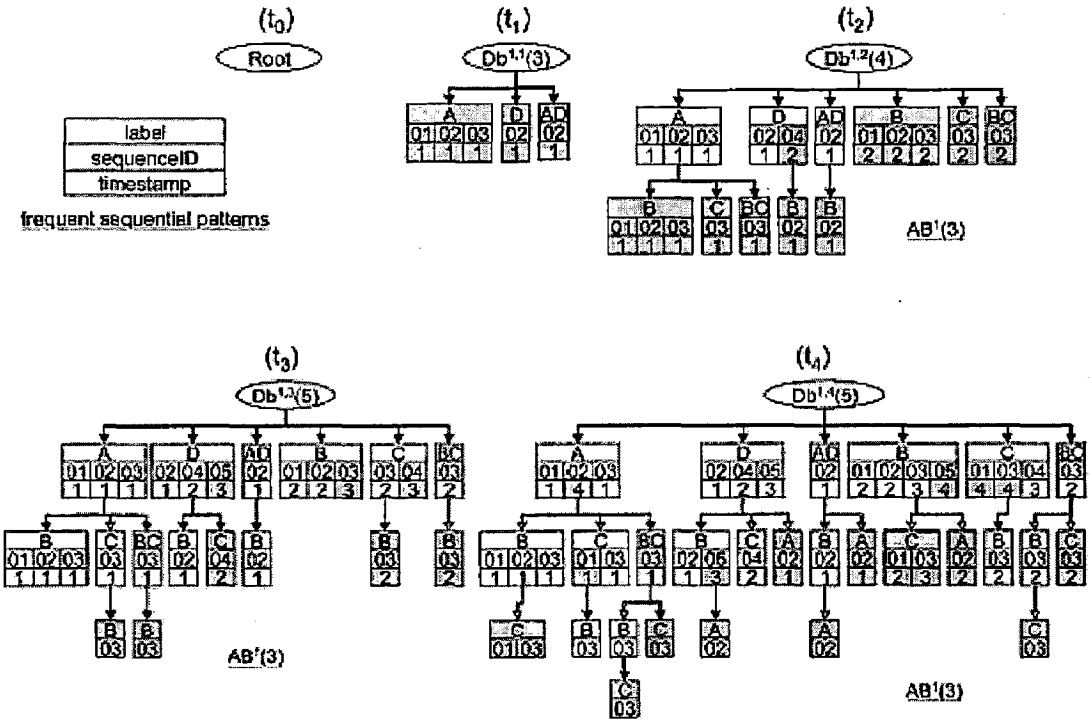


Figure 2.4: Working of PISA

- ***The Randomization Method***

In randomization technique for privacy-preserving data mining, noise is added to the data in order to mask the attribute values of records. The noise added is sufficiently large so that individual record values cannot be identified from the randomized data [10]. At times, this method provides privacy at the cost of accuracy. Therefore, techniques are designed to derive and work with aggregate distributions such as in [11]. Some secure protocols use randomization in order to hide original data from the various participating sites.

- ***Selective Result Generation***

This method is also known as downgrading application effectiveness [10]. Certain privacy preserving techniques are specific to the form of knowledge mined. The knowledge mined can be of various forms like clusters, association rules, predictions etc. In some cases although the data may not be private, leakage of mined knowledge can lead to privacy breach. In order to preserve privacy certain parameter values are

altered, in permissible limits so that the modified results hide/suppress the sensitive knowledge. As a result selective results are generated. The two most common methods of used in this technique are blocking and distortion. The parameter values to be modified depend on the type of knowledge to be suppressed. The applications of these techniques include association rule hiding, downgrading classifier effectiveness, query auditing etc.

- ***Cryptography-Based Techniques***

In cryptography-based techniques, the data entered by the people is first encrypted by using different cryptography algorithms such that at the end of the multiparty computation, no one knows anything except his own input and results. Depending on the type of application, many cryptography algorithms like RSA, DES, etc. are used for encrypting the data [10]. For privacy preservation of the data in this technique, the encrypted data is either kept by a server and the miner queries the server for mining on the data or it is shared by several miners, who can only jointly mine this data.

Apart from the standard privacy preserving techniques, certain secure sub-protocols have also been designed to securely carry out certain common tasks such as Secure Sum, Secure Dot product, Secure Comparison etc. These protocols generally use randomization with cryptography based techniques to preserve privacy while carrying out their defined task.

2.6 Privacy Preserving Distributed Data Mining

The first privacy preserving distributed data mining approach that come to mind is that, the algorithm is applied for each site independently and combines the result. This method however often fails to achieve a globally valid result, because it can cause an inconsistency between local and global results due to the following reasons [12]:

- Values for a single entity may be split across sources. Data mining at individual sites will be unable to detect cross-site correlations.

- The same item may be duplicated at different sites, and will be over-weighted in the results.
- A single site may not have information about all distinct items to be considered in the data mining process.

To overcome the above problems, algorithms were proposed for partitioning data between sites. There has been a lot of work addressing Secure Multiparty Computation. Goldriech proved existence of a secure computation for any feasible function [13], many algorithms based on his Circuit Evaluation Protocol. But this general method, which is based on boolean circuits, is inefficient for large inputs. Many other algorithms were proposed for privacy preservation across various scenarios of distributed databases, which included the use of cryptographic techniques [14], use of homomorphic encryption [15] etc. However these could incur a lot of communication overhead while mining on progressive databases with large number of items involved. Efforts were also made to apply privacy preserving techniques to specific data mining tasks such as clustering [4], [5], [6] and association rule mining over distributed databases.

Privacy preserving sequential pattern mining started gaining ground in 2004. Zhan et al. in [16] have proposed an approach, which transforms the databases of each collaborating party, followed by the execution of a secure protocol and results in the preservation of privacy as well as provides correct results. Although theoretically, this approach is robust and secure, it has serious limitations relating its applicability to real world problems. Other approaches proposed towards privacy preserving sequential pattern mining were those of using data perturbation [17] and secure two-party computations [18] which could lead to large overheads as the number of items increased. Kapoor et al in [19] proposed a method using bit vectors. Zhan in [20] proposed another method of privacy preservation using homomorphic encryption and digital envelopes.

2.7 Research Gaps Found

The major limitation of many of the existing works in the field of privacy preserving sequential pattern mining ([16], [18],[19] etc.) is that the methods proposed were applicable only to static databases. Any increment to the database requires re-computation of patterns across the entire data. This can be a limitation when data is to be mined frequently over large databases. Also the presence of obsolete data may affect the current frequent patterns. Thus a database model that could keep the data updated was proposed [3]. The collaborating parties may want to mine knowledge from their joint data from such databases. At the same time might want to preserve their sensitive data. Considering the real world applications of distributed sequential pattern mining, there was a need to use a privacy preservation mechanism for this process. The research gaps were identified can hence be summarized as follows:

- None of the existing works in privacy preservation sequential pattern mining none had considered mining of sequential patterns across progressive fragmented databases.
- The methods proposed to mine patterns from vertically and arbitrarily fragmented databases do not consider the case of mining discontinuous patterns.
- A method to hide sensitive patterns [21] from such dynamic databases with limited effects on the global results could be added to enhance privacy.

CHAPTER 3

Proposed Work for Privacy Preservation

In this chapter we discuss the methods for privacy preservation while mining frequent sequential patterns in distributed databases. We also suggest an approach to suppress sensitive co-occurring patterns. This approach could serve as a data sanitizing procedure in order to preserve privacy.

3.1 Proposed Scheme for Privacy Preservation

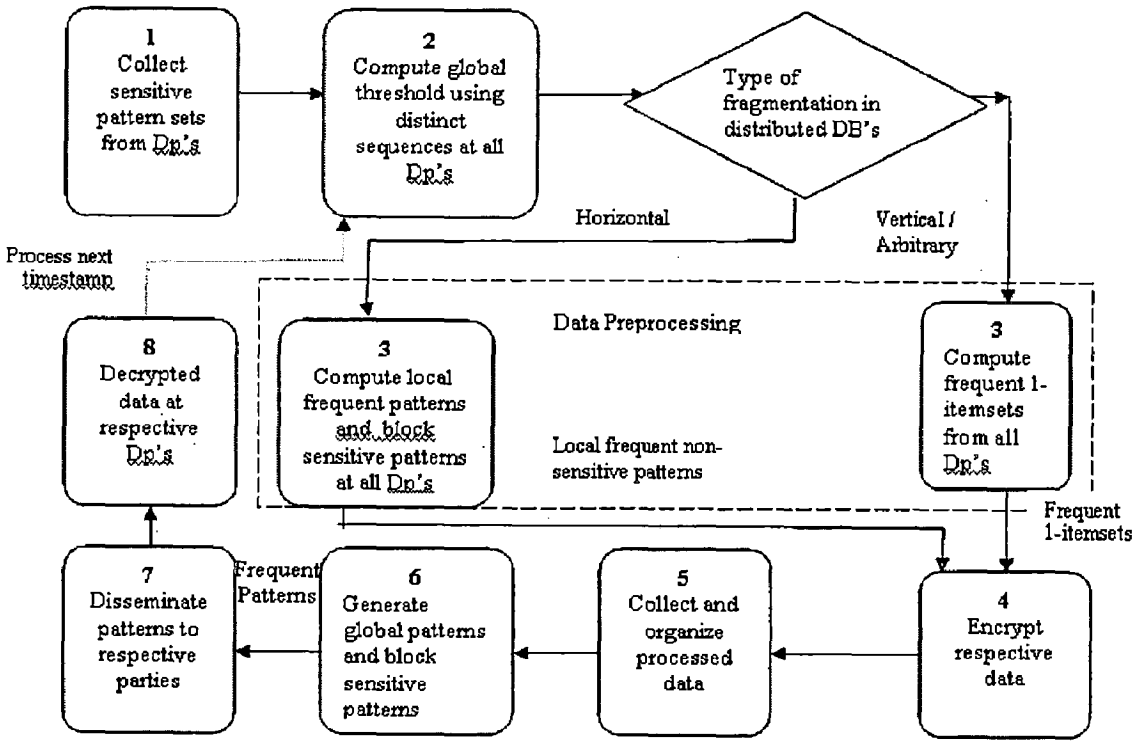


Figure 3.1: Proposed Scheme for Privacy Preservation

Assumptions

- The scheme proposed introduces a Third party (*Tp*) which acts a consolidator /mediator amongst all the data sharing parties (*Dp*).
- All the parties included in the system are assumed to be *semihonest*, i.e. they

follow the protocol as per the norms but may decipher knowledge from the information conveyed to them during the protocol.

- Determining which set of patterns is sensitive depends on the individual interest of the participating parties and would be based on domain knowledge

As shown in Fig.3.1, the proposed work can be broadly divided into 8 steps. Each of which is explained below. The detailed description of each module may differ depending on the type of fragmentation and will be discussed in respective sections.

Step 1: Collect Sensitive Pattern Sets from *Dp*'s

In case of data shared across multiple parties, in order to block the co-occurrence of sets of patterns, the information about the sensitive sets of patterns needs to be sent to the *Tp*. The data structure used to store the sets of sensitive patterns is called a *blockSet*.

The function *getSensitveSet()* models this step in Algorithms PisaInVertiFrag (Fig.3.2) & PisaInHoriFrag (Fig.3.7). In this step each *Dp* encrypts the sets of sensitive patterns it wishes to suppress using the public key its own and public key of the *Tp* and sends it to the next *Dp*. Each new set added by the *Dp* is inserted into the *blockSet*. This process continues until each party has added its sensitive pattern sets into the *blockSet*. The last *Dp* sends this *blockSet* to the *Tp*. The *Tp* later decrypts the information in the *blockSet* in order to process patterns.

Step 2: Compute Global Threshold using Total Number of Distinct Sequences

The global threshold (*Gthresh*) is calculated as

Gthresh = Total number of distinct sequences * *minSup* ..... (1)

where *minSup* is the minimum support.

The computation of total number of distinct sequences varies according to the type of fragmentation. The details of which are given in Sections 3.2, 3.3.

Step 3: Data Preprocessing

This step is used at each timestamp, to determine the data that needs to be sent to the *Tp*. The data to be sent can be in the form of frequent patterns or frequent 1-itemsets.The type of data to be sent depends on the nature of fragmentation. The details of the methods of obtaining these frequent patterns / frequent 1-itemsets are discussed in Sections 3.2, 3.3, 3.4.

Step 4: Encryption

The data to be sent to the *Tp* is encrypted by the *Dp's* in order to make it indecipherable at the *Tp*. The use of public key encryption is made to encrypt items, whereas hashing techniques are used to encrypt sequence numbers across all parties. The method of key distribution varies as per the nature of fragmentation. These methods are discussed in Sections 3.2, 3.3.

Step 5: Collect and Organize Data

The encrypted preprocessed data coming from all *Dp*'s needs to be organized at the *Tp* by matching timestamps and sequence numbers. This process identifies the *n*-itemsets that could not be identified earlier due to the fragmented nature of data. Data may also be organized on the basis of party information in order to facilitate dissemination of patterns.

Step 6: Generation of Global Patterns and Blocking Sensitive Patterns.

The generation of global patterns includes assimilation of the preprocessed data into a suitable data structure. The module also suppresses co-occurring patterns present in the *blockSet*. The methods of generation of global patterns and blocking of sensitive patterns depend upon the type of fragmentation and are discussed in detail in Sections 3.2, 3.3.

Step 7: Dissemination of Patterns to Respective Parties.

In this step the patterns generated are shared amongst the parties that contribute to the pattern. The methods for dissemination of patterns is discussed Sections 3.2, 3.3.

Step 8: Decryption of Received Patterns

The received patterns are decrypted in order to be used at the respective *Dp*.

3.2 Sequential Pattern Mining Across Vertically Partitioned Databases

Assumptions

- Each party knows the total number of parties sharing their data.
- The minimum length of the pattern and minimum number of parties is >=3
- Although the total number of sequences considered at each party is the same, all the sequences need not be updated at any given time.
- Each party (including the third party) has its own key pair (*e*, *d*) to be used for encryption and decryption. The third party shares its public key with all data sharing parties.
- All the parties share a uniform one way hashing function *H(m)*.

Description

The algorithm needs to mine both continuous and discontinuous patterns. As a result we need to mine global frequent patterns by consolidating candidate frequent items at the *Tp*. This requires certain modifications are required in the basic algorithm *PISA* in order to avoid loss of patterns.

The algorithm proposed Progressive mIning of Sequential Patterns In Vertically Fragmented database (*PisaInVertiFrag)* is shown in Fig 3.2. The steps of this algorithm are discussed below inline with those described in Section 3.1.

Step1 for this algorithm is similar to that discussed in the previous section.

Step 2: Computing Global Threshold using Total Number of Distinct Sequences

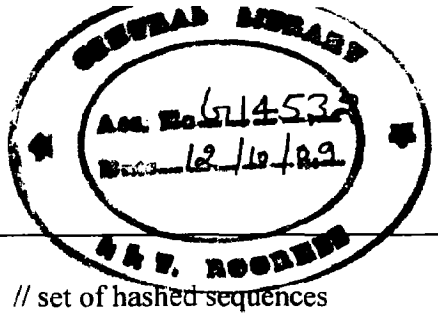
The total number of distinct sequences is computed at each timestamp by the *Tp* in order to generate the global threshold (*Gthresh*) using eqn..1. The total number of distinct sequences is computed by securely computing a union of the sequence numbers held by each *Dp*. The sequence numbers, corresponding to profile IDs' are hashed using a one way hash function at each *Dp*. Since the same hashing function is shared across all the data sharing parties to it helps to match the sequences, thus providing anonymity. The method to securely compute the union is given in Fig 3.3.

```
Algorithm PisaInVertiFrag( minSup , poi)
Var currTime = 0 ;
Var noParty ;
List fdat[] = List [ noParty ] ;
Var TSeq ;
List sendList ;
getSensitiveSet();                          //Step 1
while (new data is available at any site)
    for (each party)
      TotalSeq = calcSeq();                 //Step 2
      dat[ i ] = get FreqItems();           //Step 3 & 4
sendList =mine global patterns using dat;  //Step 5 & 6
send results in sendLIist to respective parties //Step 7 & 8
currTime++;
End
```

Figure 3.2: Algorithm PisaInVertiFrag

Step 3: Data Preprocessing

At each timestamp, each *Dp* computes and sends frequent 1-item (*getFreqItems*())sets based its local threshold. The frequent 1-item sets are locally computed using the *PISA* algorithm at the root node of the *Mtree* at each *Dp*. The *Dp's* do not send item sets that have been frequent in both, the previous and current timestamp and have not been updated in the current timestamp. For items which were frequent in the previous timestamp and have been updated in the current timestamp the party sends only the updated part of the frequent item information.



```
Procedure calcSeq()
Var seqSet ;                    // set of hashed sequences
  for (each party)
      seq' = use hash function to generate a copy of seq;
      for(every element in seq')
          if (seq' does not exist in the seqSet)
              add to the seqSet
      Pass seqSet to the next party
      send |seqSet| to the Tp
End
```

Figure 3.3: Procedure calcSeq

Step 4: Encryption

The 1-itemsets are encrypted by using the public key of each *Dp*. The sequence numbers of each of these elements are hashed using a 1-way hash function.

This data obtained in Step 4 is organized as mentioned in Step 5 of Section 3.1

Step 6: Generate Global Frequent Patterns and Block Co-occurring Sensitive Frequent Patterns

This data is now added into the global pattern tree (*GMtree*) as described in the Procedure *Modified Pisa* given below. The *GMtree* is a member of the class *gpatMiner*.

Procedure Modified Pisa

This version of the insert function, *modPisa* (Fig 3.4) is proposed to deal with data occurring at a differed timestamps. The procedure ensures minimal loss of candidate patterns from the original data. This variant differs with from the original procedure in the following aspects:

- The node contains an added field *origts* for keeping track of the original timestamp of occurrence in addition to the existing 3 fields (viz label, sequence number and timestamp).
- While adding data into the tree the algorithm checks for the timestamp of the element along with the other pre-existing checks.

```
Procedure modPisa (Tc , Mtree)
for(each node of Mtree in post order)
if(node is Root node)
    for(ele of every seq in eleSet)
      for(all combination of elements in ele)
        if(element ==label of one of node.child)
          if(seq is in node.child.seq_list)
           update timestamp of seq to Tc;
           update orig_ts of seq to Tc;
           else
          create a new sequence with timestamp =orig_ts= Tc;
        else  // create a child node
        create child node with element, seq, orig_ts =timestamp = Tc;
else     // for a common node
    for(every seq in the seq_list)
      if(seq.timestamp <= Tc - POI)
        delete seq from seq_list and move to next seq;
      if(there is new ele of seq in eleSet)
        for(all combinations of elements in ele)
          if(element is not on the path from Root)
            if(ts of element >node.seq.orig_ts)
              if(element == label of one of node.child)
                if(seq is in node.child.seq_list)
                  child.seq_list.seq.timestamp = node.seq.timestamp;
                  child.seq_list.seq.orig_ts = element.ts;
                 else
                   create new sequence with timestamp = seq.timestamp,
                   orig_ts =element.ts;
            else  //create a child
                create a new child with element,seq, timestamp = seq.timestamp
    else if (ts of element == node.seq)
            create a new joint_label with element.label + node.label;
        if ( node.parent.child.label!=joint_label )
            create a new child node at node.parent with joint_label,node.seq ,
           timestamp =  node.seq.timestamp, orig_ts =element.ts;
     if(seq_list.size ==0)
       delete this node and all of its children from its parent;
if(seq_list.size>=support*sequence number)
 output labels of path from Root to this node as sequence pattern;
End
```

Figure 3.4: Procedure modPisa

The algorithm generates 2/n-itemsets dynamically in case they could not be discovered earlier. In order to block co-occurring patterns, *modPisa* algorithm is further modified subjected to the conditions mentioned in *co-occurBlock* algorithm (Fig.3.5) and procedure *modInsert* (Fig.3.6).

```
Algorithm co-occurBlock( )
Var currentTime;
Var threshold;
Var blockSet;                    //Stores sensitive sets of patterns
Mtree root;
while(there exist new data at currentTime)
    Extract data from database
    Calculate threshold;
    for(every pattern in every set in blockSet)
      prune obsolete sequences
      if(pattern.support >= threshold)
          set pattern.threshFlg;
      else
          reset pattern.threshFlg;
    root.traverse()
    currentTime ++;
End
```

Figure 3.5: Algorithm co-occurBlock

Algorithm co-occurBlock

Sometimes although a particular pattern is not interesting, its co-occurrence with some other patterns may reveal certain sensitive information. The proposed algorithm suppresses co-occurring frequent sensitive patterns sets by blocking some of the sensitive patterns. As a result if pattern set $P = (A, B)$ is sensitive, the proposed method blocks either A or B; avoiding both A and B to be frequent together at a particular time instance. This avoids the co-occurrence of A, B as frequent patterns at the same timestamp.

As discussed in Step 1, the information about sets of patterns, considered sensitive, is maintained in the *blockSet*. The proposed method maintains updated information about the status of patterns in a *blockSet*. The *threshFlg* associated with a pattern indicates whether the support of that pattern in the *blockSet* has crossed the threshold value. The algorithm updates the status of the pattern while it updates the Mtree at every timestamp as explained in Fig.3.5.

Fig.3.6, describes the changes to be incorporated in the insert function (Fig 2.3). Each node denoting a sensitive pattern has its *bcandid* field set to 1. While adding a sequence to a node, which denotes a sensitive pattern, the algorithm first checks for

```
Procedure modInsert( root, blockSet)
for(each node of Mtree in post order)
   if(adding new element to the tree)
      if (element creates pattern that exists in blockSet )
         add element and set element.bcandid = 1;
      if(adding new sequence to the node)
         if(node.bcandid ==1)
            if(pattern.support > threshold for rest n-1 patterns in set)
               block sequence;
            else
               add sequence;
               if(node.support >=bthresh)
                  set pattern.threshFlg;
End
```

Figure 3.6: Procedure modInsert ( )

the values of *threshFlg* of other patterns in the set. If the value of *threshFlg* of not more than *n-2* patterns in a set are 1 then the pattern adds the sequence to the node else blocks the sequence. If a node that denotes a sensitive pattern, is added into a tree, this node is marked by setting its *bcandid* field to 1.

Step 7: Dissemination of Globally Frequent Patterns

After frequent patterns are mined, the *Tp* generates a message containing the following information:

- Random sequence of *Dp*'s that have contributed to the pattern
- Encrypted frequent pattern

This message is sent to the first *Dp* in the random sequence. On receiving the encrypted pattern each *Dp* identifies its share of items in the pattern. It decrypts these elements and substitutes these elements in the pattern by their encrypted form. The encryption in this round is done using the public key of the last *Dp* in the random sequence. This newly encrypted pattern is sent to the next party in the sequence. This process is carried out until the last *Dp* in the sequence receives the pattern.

Step 8: Decryption

After the last *Dp* has processed the pattern, the *Dp* decrypts all the elements using its public key and broadcasts the pattern all the *Dp*'s in the random sequence.

3.3 Sequential Pattern Mining in Horizontally Fragmented Databases.

Assumptions

- We assume that all (Dp_i's) own a random number of tuples/sequences from the total database.
- The set of items contained in the sequences of all parties remains the same.
- The algorithm assumes all data sharing parties to use the same key pair generated using a public key encryption scheme. This ensures that the same pattern gets uniformly encrypted across all the parties.

Description

The proposed algorithm, Progressive mIning of Sequential pAtterns In Horizontally Fragmented databases (PisaInHorFrag) is given in Fig 3.7. Some steps which need detailed discussion (as mentioned in Section 3.1) are given below.

The step 1 of the proposed algorithm is similar to that mentioned in Section 3.1.

Step 2: Calculating Global Threshold using Total Number of Distinct Sequences

The process of computation of total number of distinct sequences is described in the function *getSeq* (Fig.3.8).This function is triggered by the *Tp*. The *Tp* generates a random number say *R*. It also generates a random order sequence of *Dp*s. The *Tp* sends *R*, and the random sequence to the first *Dp* in the generated random sequence. When a Dp_i receives input from another party, it adds to the input *R*, the number of distinct sequences the in the current *POI* in its own fragment of data. The Dp_i then deletes its name from the random sequence and sends the resulting sequence list and

```
Algorithm PisaInHoriFrag (minSup , poi)
Var currTime ;                    // Stores current time
Var noParty ;                     //Stores no. of Dps
List fpat[] = List [ noParty ] ;
Var TSeq ;
List Gpvector;
List sendList ;
getSensitiveSet();                //Step 1
while (new data is available at any Dp)
    for (each party)
        TSeq = getSeq();          //Step 2
        fpat[ i ] = getPatterns( currTime ,poi,minSup)  //Step 3 & 4
        Gpvector = calcGpvector();                      //Step 5 & 6
        sendList = calcGpat(GPvector);
        send data in sendList to respective parties;  //Step 7 & 8
currTime++;
End
```

Figure 3.7: Algorithm PisaInHoriFrag

modified random number to the next party in the list. Hence if X_A is number of distinct sequences at party A then party B receives $R' = R + X_A$ from party A and so on.

This process of cumulative addition of individual portions of data continues till all Dp_i are processed. The last Dp_i sends the final modified value of R' to the Tp. The Tp now subtracts the random number it had initially sent, from R' to get the total number of distinct sequences.

```
Procedure getSeq ( noParty)
Var Order [] = Var [ noParty ] ;
Order = randomize(noParty);
int Random = random number generated by Tp ;
int Random' = Random ;
for ( each partyNo in Order )
   Random' = Random' + no. of distinct sequences in Db in party[ partyNo] ;
    Remove partyNo from Order
    Send Random, to Tp
    Random' = Random' – Random ;
End
```

Figure 3.8: Procedure getSeq

Step 3: Data Preprocessing

Each *Dp* computes frequent patterns on the data in its fragment using algorithm *PISA*. The algorithm also hides any sensitive co-occurring patterns as per the conditions mentioned in Fig.3.5 and Fig 3.6.

The *Dp* needs to send these frequent patterns to the third party as candidate patterns for mining globally frequent patterns. These patterns are sent in the form of a vector where the pattern forms the dimension and its support count the corresponding magnitude. The vector at each site is a resultant of absolute change in each dimension. The procedure to compute the vector is given in Fig.3.9. Here we introduce a new term called *virtual support.*

The virtual support of a pattern can be defined as the change in the support count of a pattern since the last timestamp. In this case, the support of a pattern, if it is not frequent, is considered 0. Hence if a pattern is newly frequent and has not been frequent in the previous timestamp, it is added to the vector as a new dimension and its virtual support count is equal to its magnitude. Similarly if a pattern has been frequent in the previous timestamp and the current timestamp and there is a change of support count, it is reflected in the vector:

$$Virtual\ Support_t = support_t - support_{t-1} \quad \ldots \quad (1)$$

```
Procedure getPatterns ( currTime ,poi,minSup)
List t_past[] ;
List t_curr[] ;
t_past = t_curr ;
List Fp = frequent patterns from pisa(currTime ,poi,minSup)
t_curr = Fp ;
for(each pattern in Fp)
   if(pattern[i] exist in t_past )
        pattern[i].support = t_curr.pattern[i].support - t_past.pattern[i].support ;
   else
        pattern[i].support = t_curr. Pattern[i].support ;
        Add pattern to the list of patterns to be sent Fp';
return Fp'
```

Figure 3.9: Procedure getPatterns

The *Tp* keeps a track of sequential patterns sent by each party in a log and automatically prunes them when the pattern becomes infrequent.

Step 4: Encryption

The non-sensitive local frequent patterns at each *Dp* are encrypted using the public key shared by all the *Dp's.*

Step 6: Computation of Globally Frequent Patterns

In Step 5, the *Tp* consolidates all patterns obtained from the *Dp* and adds them to the Global pattern vector (*Gpvector*) of the previous timestamp by matching the patterns (Fig 3.10). It tries to find patterns which have a support count greater than *Gthresh.* For every pattern that has a threshold above the global, the algorithm checks if the pattern exists in the *blockSet.* If the pattern forms a part of a sensitive set, the *Tp* checks for the support count of the pattern against the blocking conditions as described in Fig 3.6.This procedure determines whether a pattern can be declared to be globally frequent or it needs to be blocked.

```
Proc calcGpvector ( )
    List Gpvector_past ;
    List Gpvector_curr ;
    Gpvector_curr = Gpvector_past
    for(every pattern in fpat)
            if( pattern not in Gpvector_curr )
                    add pattern to Gpvector_curr ;
            else if( pattern exist in Gpvector_curr )
                    add pattern.support to Gpvector_curr [ pattern]
                    if( Gpvector_curr [pattern].support <= 0)
                            remove pattern from Gpvector_curr
            else if( partyNo does not exist in Gpvector[pattern].partyList)
                    add partyNo to Gpvector_curr [ pattern].partyList
return Gpvector_curr
```

Figure 3.10: Procedure calcGpvector

If a pattern has a support count greater than *Gthresh,* the *Tp* declares the pattern to be globally frequent.

The *Tp* then segregates patterns with threshold lesser than the global threshold according to the parties in which they are infrequent. Each party hence receives a list of infrequent patterns from the *Tp*, and returns the support count of the corresponding pattern from its candidate pattern list. These support counts are now added to the *Gpvector* to find if any of the patterns is globally frequent (Fig.3.11). The patterns that are globally frequent are added to a list.

Step 7: Dissemination of Global Patterns

The patterns in the list are shared only amongst *Dps* where the pattern was originally frequent.

Step 8: Decryption

Since the key to encryption is shared by each *Dp*, the patterns received can be decrypted by each party independently.

```
Proc calcGpat(Gpvector_curr)
List sendList ;
List Gpvector' = Gpvector_curr;
Gthresh = TSeq * minSup ;
for(each pattern in GPvector' )
    if( Gpvector' [pattern].support > Gthresh)
        remove pattern from Gpvector'
        add to sendList ;
   else
        get pattern.support from all parties not in partyList
        update Gpvector'.pattern.support with partyNo.pattern.support
for( each pattern in GPvector')
    if( Gpvector' [pattern].support > Gthresh)
        add to sendList ;
return sendList
```

Figure 3.11: Procedure calcGpat

3.4 Sequential Pattern Mining over Arbitrarily Fragmented Databases.

Assumptions

- Each party can have any item at any time instance.

- Also the number of sequences under consideration need not remain constant.
- The system uses a public key cryptosystem, with each party having information of the public keys of the other parties. The parties share a common key pair to encrypt all the items at all parties uniformly. The parties also share a common hashing function in order to hash sequence numbers and items uniformly.

Description

The generalized nature of this fragmentation scenario makes it necessary to share more number of parameters as compared to the previous two cases. The method proposed here uses ideas discussed in the previous two fragmentation scenarios, but with a slight variation. The underlying algorithm is similar to Algorithm *PisaInVfrag* (Fig.3.2). The steps in which this module differs from the generalized model are discussed below.

Step 1 and 2 of this algorithm are similar to the steps mentioned in Section 3.2.

Step 3: Data Preprocessing

Unlike the case of vertical fragmentation, the items occurring in a party can occur in another party for the same sequence at a later timestamp. As a result it is difficult to decide if a particular item is frequent.

This module aims at getting a union of distinct sequences over which a particular item occurs (Fig.3.12). The cardinality of this set of distinct sequences is now compared with the *Gthresh*. The set of items whose cardinality is above the threshold are selected to be candidates for mining global frequent patterns. Each party now sends information pertaining to these candidate items to the third party.

Step 4: Encryption

Each party encrypts the data it sends to the *Tp* using the public key in the common key pair shared across all parties. The sequence numbers are hashed using the one way hashing function in order to provide anonymity.

In Step 5, the *Tp* organizes the encrypted data by matching the sequence numbers and timestamps. The *Tp* also maintains information about the elements sent by each *Dp* at at each timestamp.

Step 6: Computation of Globally Frequent Patterns

The data organized at the *Tp* is now assimilated in the GMtree at the *Tp* using the procedure mentioned in Fig.3.4. The sensitive co-occurring patterns are blocked using the algorithm mentioned in Fig.3.5 and Fig.3.6.

Step 7: Dissemination of Global Frequent Patterns

After the patterns are mined at the *GMtree* the *Tp* segregates the patterns according to the contribution of the participating parties. The patterns are now sent to each party based on this segregation. Since all parties use the same encryption scheme, each party can decrypt a given sequence without any communication with other parties.

```
Procedure calcFreqLab()
List seqSet ;        // set of sequences of each item
  for (each Dp)
    for(each element occurring at a currTime)
      seq' = encrypt a copy of seq of element using Tp's public key;
      if (element does not exist in the seqSet)
        add element and seq' to the seqSet and element.seq';
      else
        if (element exist in the seqSet)
          if (element does not exist in the seqSet)
            add seq' to element.seq
  Pass seqSet to the next party
  if(party = = last Dp)
    for(each element in seqSet)
      if |element.seq| >Gthresh
        broadcast element to all Dp
End
```

Figure 3.12: Procedure calcFreqLab

CHAPTER 4

Implementation Details

The implementation of the proposed algorithms comprised mainly of the following tasks.

- Selection of programming language
- Design and development of the basic network architecture as assumed
- Designing databases and establishing connectivity
- Implementing the basic *PISA* algorithm
- Coding of the proposed algorithms

4.1 Selection of Programming Language

The implementation of the proposed algorithms and all its pre-requisites was done using Java. Java platform is freely downloadable and provides large assistance to amateurs by means of its documentation, forums and API's. The expanse of the language and its seamless integration with fields such as databases, networks, data structures etc was one of the main reasons for the choice. The modularity and object oriented nature of the language gives the user the freedom to individually build and test modules independent of each other.

4.2 Design and Development of the Basic Network Architecture Assumed

The network architecture assumed for the simulation of the distributed environment is that of a single client (known as the Third party (*Tp*) in the above text) and multiple data servers (known as data parties (Dp_i)).The architecture provides for one to one communication between the client and the servers and also in between the servers.

This system was simulated using the Remote Method Invocation (*RMI*) and socket programming concepts. Java RMI enables the programmer to create distributed Java technology-based applications, in which the methods of remote Java objects can be

invoked from other Java virtual machines. The inter server communication was established using socket programming. Some of the classes which play a role in the development of this architecture are listed below. The methods described in these classes are generic to all the fragmentation scenarios. The specific methods will be discussed in the respective sections.

class DataServer

This class is used to create and bind objects (which correspond to *Dp*'s) to the RMI registry. The arguments passed to the constructor allow dynamic creation of user defined number of *Dp*'s.

Interface DataIntf

The interface defines the client's view of the remote object. As a result any communication between the client and server needs to be made using the functions declared in the interface.

class DataImpl

The functions listed in the interface are defined or implemented in this class. This class interacts with the *Dp*'s using these functions and gets back the desired results. The sockets to communicate between various *Dp*'s are also created using the same class. Some important functions in this class are:

public ArrayList *process*()
public void *run*()
private void *sendThis* (int no, Object obj)

Table 4.1: Listing of generic methods in class DataImpl

- **public ArrayList *process*()**

This function returns the results of data preprocessing to the *Tp*. These results may be in the form of frequent patterns, items etc.

- **public void *run*()**

This function is responsible for receiving any message/object sent from any *Dp*. The object received is then typecast and used for further processing.

- **private void *sendThis* (int no,Object obj)**

This function is used to send messages/objects to any other *Dp*. The first argument in the function denotes the *Dp* identity with whom the communication is to be established and the second argument denotes the object to be sent.

class ThirdParty

This class simulates the *Tp* and maintains synchronization and timing between the multiple servers. This class responsible for the assimilation of data, generation of patterns from this data and dissemination of patterns.

Some of the important functions in this class are:

public void *setParam*()
public void *getMax*()
private void *timeCalc* ()
public void *getSensitiveSet*()

Table 4.2: List of generic methods in class ThirdParty

- **public void *setParam* ()**

This function sets values of parameters such as minimum support and period of interest for all the *Dp*'s in the network.

- **public void *getMax* ()**

This function computes the maximum time of simulation by consulting each data server.

- **public void *timeCalc* ()**

This method maintains synchronization between all the *Dp*'s. This function acts as the main processing block and calls other functions which would deal with tasks of message transfer, maintenance of data and data mining operations.

- **public void *getSensitiveSet* ()**

This function collects data to models the algorithm Co-occurBlock mentioned in Sections 3.2. This function initiates a procedure to securely obtain a collection of patterns sets which are considered sensitive by the *Dp*'s. These sets of patterns are stored in a structure called *blockSet*.

4.3 Designing Databases and Establishing Connectivity.

The DBMS used for managing the databases in the simulated environment is *MS-Access*. Multiple database connections are created in order to simulate the distributed environment. The data for the given problem statement is in the form as shown in Fig.2. The database tables exactly model the data similar to that given in the figure in a tabular format using attributes of timestamp and sequence number to represent the item / group of items. The **class DBdriver** is developed to manage all the database communication and data manipulation w.r.t the database.

4.4 Implementation of Basic Algorithm PISA

The implementation of the basic algorithm PISA is divided into two main classes: **class Pisa** and **class Mtree.** The class Pisa gives the overall structure and is the driver class of the algorithm described in Fig.2.2, whereas the class Mtree defines the M-ary tree data structure, its various operations and models the function *insert* described in Fig.2.3. The class Mtree is used in Horizontal fragmentation scenario for generating frequent patterns at each *Dp*.

Some of the main functions in class Mtree are:

public *Mtree* (String lab, int s, int t, int c)
public void *addToNode* (int s, int t, int c)
public void *insert* (Mtree m, int currTime, ArrayList b)
pubic Boolean *notLiesOnRoute* (String s)
public ArrayList *fpGen* (int currTime , int minSup)

Table 4.3: Listing of generic methods in class Mtree

- **public *Mtree* (String lab, int s, int t,int c)**

This function creates a node of an M-ary tree which in itself can at a later stage develop into an M-ary tree. This node has a label *lab*, sequence number where the label occurred as *s* and its timestamp of occurrence as *c*. The third parameter denotes the time of start of the pattern over that sequence.

- **public void *addToNode* (int s, int t, int c)**

This overloaded function is used to add a new sequence *s* into a node of the Mtree or may be used to add a child node to the node. This function can be overloaded as **public void *addToNode* (Mtree z)** to add a child node into the Mtree.

- **public boolean *notLiesOnRoute* (String s)**

This function checks if a particular label *s* already exists in the pattern. Since each pattern in the M-ary tree is denoted by a path from the root till a leaf node, in order to check for duplicates in a pattern it is necessary to back traverse the path from the leaf node till the root.

- **public void *insert* (Mtree m, int currTime, ArrayList b)**

This function is used to insert items and their associated information in the M-tree. The method of insertion used in the function is described in Fig. 2.3.The

function also blocks frequent sensitive co-occurring patterns, the information of which is stored in ArrayList *b*.

- **public ArrayList *fpGen* (int currTime, double minsup)**

This function computes the frequent patterns by calculating the threshold and checking if the number of sequences each node of the M-ary tree exceeds the threshold. If the number of sequences exceeds the threshold the function back traverses the path till the root declaring labels on that path to form a frequent pattern.

The class Mtree is modified to class GMtree for its usage in vertical and arbitrary fragmentation scenarios.

4.5 Implementation of Proposed Algorithms

This section gives an idea about implementation of the algorithms proposed in the previous section. Each section gives an overview of the important functions in the algorithm.

Algorithm co-occurBlock:

This module implements the algorithm co-occurBlock by modifying the class Mtree in case of horizontal fragmentation scenario and modifying *GMtree* class in case of vertically and arbitrary fragmentation scenario. Some of the important functions incorporated are:

public int *chkSensitive* (String s, ArrayList b)
public void *updateThresh* (int val, String s, ArrayList b)
private int *chkLock* (String s, ArrayList b)

Table 4.4: Listing of methods used for implementing Algorithm co-occurBlock

- **public int *chkSensitive* (String s, ArrayList b)**

This function checks if the path from the current node till the root is a sensitive pattern. This procedure is carried out by back traversing the tree and comparing it with the patterns in the *blockSet*.

- **public void *updateThresh* (int val, String s, ArrayList b)**

This function is used to update the value of the *thresh* field of a pattern *s* in the *Set*. The value of the *thresh* field may be set to 1 or reset to 0 depending upon the support count of the pattern.

- **public int *chkLock* (String str, ArrayList b)**

This function is used to check if a sequence can be added into a pattern *str*. If (n-1) patterns in the set are blocked; i.e their support counts have crossed the threshold, the sequence under consideration is blocked otherwise it is added.

Vertical Fragmentation Scenario:

Some of the important functions in **class DataImpl** are as follows:

public void *getNoSeq* (int currentTime , int poi)
public ArrayList *process* ( )
public void *getPat*(SendList x)

Table 4.5: List of *Dp* Functions in Vertical Fragmentation

- **public void *getNoSeq* (int currentTime, int poi)**

This function models the procedure *calcSeq* (Fig.3.3). This function is used to get a secure union of sequence numbers that are active in the current *POI*.

- **public ArrayList *process*( )**

This function returns the items that are frequent in the current timestamp. In

order to avoid redundancy the data at a party is organized into a single level *Mtree* and returns only updated frequent items at each time instance. These items are encrypted as in described step 4 of Section 3.2.

- **public void *getPat*(SendList x)**

This function is used to decrypt items and communicate between parties when a frequent item is received from the third party. The logic of the function is explained in step 6 of Section 3.2.

Some of the important functions in **class ThirdParty** are as follows:

public void *getData* (ArrayList flist[])
public void *Send* (ArrayList x)
public void *getPat*(SendList x)

Table 4.6: List of specific functions in class ThirdParty for Vertical Fragmentation

- **public void *getData*(ArrayList flist[])**

This function is used to get, organize and update data input from the various servers. The use of this function is to identify any 2/n-itemsets that could be obtained by combining data coming from two or more parties. The data is also organized according to the party it comes from in order to facilitate the sharing of patterns with the servers. This data needs to be updated and pruned so that only up to date data is processed.

- **public void *Send*(ArrayList x)**

This function is used to send encrypted elements from frequent patterns to the intended parties. The structure of the data sent is as discussed in step 6 of Section 3.2. This data is further organized to obtain decrypted patterns at a predefined server.

class VGpatMiner is used to organize and mine global patterns from distributed vertically fragmented databases. Some of the important functions in this class are as follows:

public ArrayList *gpatMine* (SeqList Gdata[])
public ArrayList *getPartyList* (ArrayList s)

Table 4.7: List of functions of in class VGpatMiner

- **public ArrayList *gpatMine*(SeqList Gdata[])**

This function works on the data *Gdata*[], organized by the getData( ) in class ThirdParty. This data is inserted into the *GMtree* using the modified Pisa algorithm as described in Fig.3.4. This function also triggers the function that mines frequent patterns from the *GMtree*.

- **public ArrayList *getPartyList*(ArrayList s)**

The list of frequent patterns *s* provided by the previous function needs to be segregated according to the parties that contribute to them. This function associates each element of the pattern with a party and also randomly determines the party where the pattern has to be sent.

Horizontal Fragmentation Scenario:

Some of the important functions in **class DataImpl** are as follows:

public ArrayList *genRandom*(ArrayList s)
public int *getNoSeq* (int currentTime, double minSup, int poi)
public void *procPat* (ArrayList c)

Table 4.8: List of important *Dp* Functions in Horizontal Fragmentation

- **public ArrayList *genRandom* (ArrayList s)**

This function is used to generate a random vector of a set of *Dp* names. The vector is used for randomization of inputs to a particular function in order to provide security.

- **public int *getNoSeq* (int currentTime, double minSup, int poi)**

This function models the procedure *calcSeq* (Fig 3.3).The function extracts the number of distinct sequences from the database for the *currentTime* and *POI.* These sequences are later added to get the total number of distinct sequences at the *Tp.*

- **public void *procPat*(ArrayList c)**

This function is actually responsible for calculating the frequent pattern vector. The procedure of calculating the frequent pattern vector is given in procedure *getPatterns* (Fig 3.9).This procedure of computing the frequent pattern vector is calculated as follows:

Consider the set of frequent patterns at time t_1 along with their respective support counts to be {*AB* (4), *BC* (5), *CA* (3)}.These can be represented as a vector:

$$4AB + 5BC + 3CA$$

Now consider the frequent patterns at time t_2 the frequent pattern set considers say {*AB*(5), *BC*(5),*CD*(4)}.The function compares the previous vector with the current set of frequent patterns to get:

$$1\ AB - 3CA + 4\ CD$$

This gives an indication that the support count of *AB* has incremented by 1, support count of *CA* has decremented by 3 and the support count of *CD* has either incremented by 4 or CD is a new pattern in the vector.

Class HgpatMiner is used to generate global patterns in the horizontally fragmented scenario. Some of the important functions in this class are as follows:

public ArrayList[] *calcGpat1*( )
public void *calcGpat2* (ArrayList a[])
public void *calcGpat3*( )

Table 4.9: Listing of important Functions in class HgpatMiner

- **public ArrayList[] *calcGpat1*( )**

This function organizes frequent pattern vectors obtained from each site according to the site they come from. The function also generates the global pattern vector by matching the dimensions from the previous timestamp and adding the corresponding magnitudes by usage of function *arrPat*(explained as *calcGpatvector* in Fig.3.10). The function returns the global pattern vector which is sent to calcGpat2 for further computations.

- **public void *calcGpat2*(ArrayList a[])**

This function models procedure calcGpat (Fig.3.11). This function compares the magnitude of each dimension in the vector (denoted by ArrayList a[]) with the global threshold (*Gthresh*) and eliminates the dimension from the vector. The dimensions whose magnitude is less than *Gthresh* are added to the search list of respective parties in order to get their support counts. The search list consists of list of parties who have not contributed to the candidate frequent pattern at the current timestamp. A part of the procedure given in Fig.3.11 is implemented in the next function.

- **public void *calcGpat3*( )**

This function assimilates data obtained after the supports of candidate frequent patterns are updated by support counts from the parties in the search list. The

new support values are now compared against *Gthresh* to get the final set of globally frequent patterns.

Arbitrary Fragmentation Scenario

This case of the algorithm mostly uses functions from the previous two cases. The two main functions where the functionality of this case differs from that of the other two cases are reported as follows:

- **public void *getData*(double gthresh)**

  This function defined in **class DataImpl** is used to get the candidate frequent itemsets. The function maintains a secure union set of sequences over which a particular item occurs. Each set is later compared with the global threshold *gthresh.* If the number of sequences exceeds *gthresh* and the sequences corresponding that item are obtained from each party. The logic of this function is explained in Fig.3.12.

- **public void *FinalSend*(ArrayList send)**

  This function is defined in **class AgpatMiner** and is used to send patterns in the ArrayList *send* to the participating parties. The class AgpatMiner mines global patterns from an arbitrarily fragmented database scenario.

CHAPTER 5

Results and Discussion

The following section discusses the performance of the proposed algorithms across various parameters. The results for the proposed work have been tested over a test dataset of 300 customers over 30 days. Number of items considered for mining is 15. This data was obtained using a synthetic dataset generator available at [22].This data was subjected to various types of fragmentation across variable number of data servers.

The values of *POI* considered for testing results (*7, 10, 15*) mark *1/4*, *1/3* and *1/2* of the total timespan under consideration. The values of minimum support considered are values around the knee point in the graph shown in Fig.5.1.

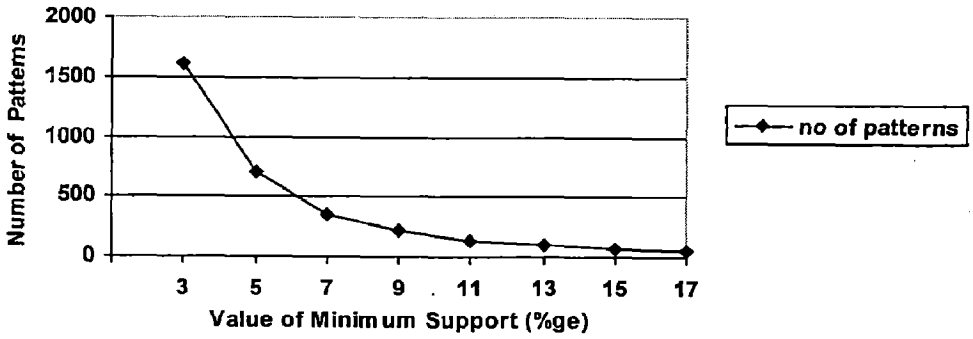


Figure 5.1: Number of Patterns v/s Percent Value of Minimum Support

5.1 Performance of Protocols with respect to Patterns Mined

The performance protocols can be tested for Precision. The *Precision* of the algorithms can be calculated as:

Precision: Number of Frequent Patterns Mined by the algorithm

No of Frequent Patterns Generated by *PISA*

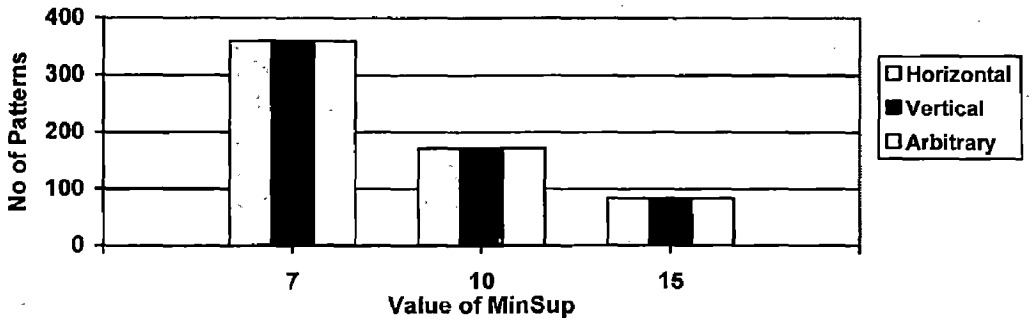


Figure 5.2: Number of patterns mined for various fragmentation scenarios for fixed value of *POI*=7

Fig 5.2 shows that all the algorithms mine 100% of the patterns generated by *PISA*(without using co-occurrence blocking algorithm), i.e. the precision of the algorithms is 100%. The above graph shows the number of patterns mined over the period of 30 timestamps using the given value of *POI*.

	No. of Patterns over 15 timestamps		No. of Patterns over 30 timestamps	
POI / Min. Support	7	10	7	10
7	150	360	360	1058
10	79	170	172	556
15	32	72	84	225

Table 5.1: Number of Patterns generated for various values of minimum support and *POI*

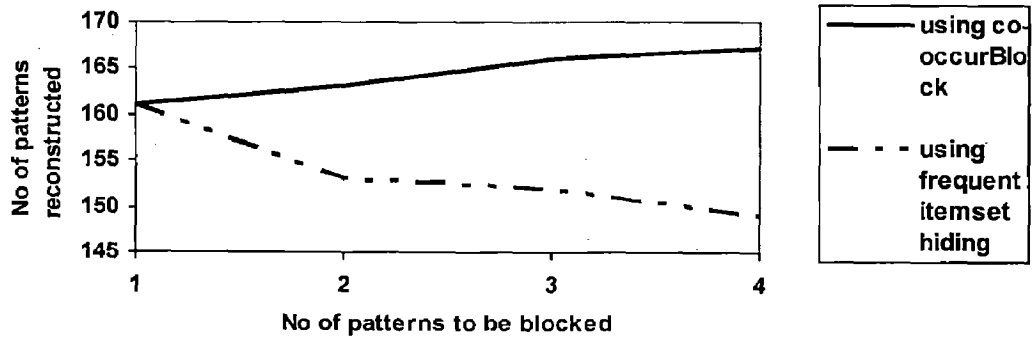
The results in Table 5.1 indicate that the number of patterns mined increase as the value of *POI* increases. Also the number of patterns is inversely proportional to the minimum support threshold. The adherence to these two properties serves a primary checksum to ensure the correctness of the algorithm.

5.2 Impact of the Co-occurrence Blocking Algorithm on Patterns Mined

No of blocking sets	Percentage of patterns mined after co-occurrence blocking		Percentage of patterns mined after frequent itemset hiding	
	Timestamp (ts) =15	At ts = 30	At ts = 15	At ts = 30
1	98.7	99.4	97.4	98.8
2	89.8	91.9	79	83.8
3	81	83.8	62	67.6

Table 5.2: Comparison of co-occurrence pattern blocking algorithm with frequent pattern hiding (min. support = 10, POI = 7)

The results in Table 5.2 are computed for a random set of patterns to be blocked over an arbitrary / vertical fragmentation scenario (both cases are considered simultaneously since trees generated by both the cases are isomorphic).These results are compared with the results of frequent itemset hiding. It is seen that the results are better since the number of patterns suppressed using frequent pattern set hiding are n times that of the proposed method, where n is the number of patterns in the set. Also since the number of suppressed patterns is less, the number of patterns which are lost due to the blocked patterns derived from these patterns is also less. The following graph compares the number of patterns reconstructed in with the increase in the size of the set to be blocked.



The use the approach proposed for co-occurrence blocking is amended for its usage in a horizontally fragmented scenario (as discussed in Section 3.3).However this method

No of sets	Horizontal fragmentation (%ge of reconstructed patterns)		Vertical/Arbitrary fragmentation (%ge of reconstructed patterns)	
	ts = 15	ts = 30	ts = 15	ts = 30
1	94.9	97.1	98.7	99.4
2	86.1	89.6	89.8	91.9
3	77.2	82.7	81.0	83.8

Table 5.3: Comparison of the two variants of co-occurBlock protocol

may lead to false positives due to the random order of blocking. The table 5.3 gives a comparison between the accuracy of two proposed variants. The number of patterns mined (without use of Co-occurrence blocking algorithm) were 79 and 172 at timestamp (ts) 15 and 30 respectively (with values of min. support = 10, POI = 7).

5.3 Information Disclosed by the Algorithms

Here, we discuss the information disclosed at each stage of the algorithm. The system assumes each party (including the *Tp*) to be semi-honest. As a result these parties follow the protocol sincerely without carrying out any malicious activity but are eager to decipher knowledge out of the information provided to them during the course of the protocol.

Case 1: Vertical Fragmentation

- In case vertical fragmentation each party has its independent encryption scheme. As a result an item encrypted by a party can not be decrypted by any other party.
- The nature of the fragmentation makes it necessary to send information about the sequence number and the timestamp along with the element. As a result the

scheme uses a common one way hash function across all parties to encrypt the sequence numbers. This ensures that a party can identify a sequence number, only if it itself has that sequence in its own list.

- In the preprocessing stage each party sends only information about the candidate frequent 1-itemsets. This ensures that only frequent 1 itemsets reach the *Tp*. Since the labels in these itemsets are encrypted, the *Tp* learns only about the statistics of the globally frequent 1-itemsets.

- In the pattern dispersion stage, the random vector ensures that two parties do not collaborate to find the data contributed by any third party. The encryption/decryption protocol introduced also anonymizes the data and avoids any party to decipher the contributions made by the previous parties in the vector.

Case 2: Horizontal Fragmentation

- Although a lot of data gets transferred to the third party, the use uniform encryption scheme amongst data sharing parties keep the *Tp* from identifying the data it is working upon. The only knowledge that may be obtained would be statistical knowledge of encrypted patterns.

- The computation of total number of distinct sequences across all the parties does not reveal any information about any party's contribution to the total sum nor does it disclose the total number of distinct sequences to any *Dp*. The random vector provided by the *Tp* avoids two *Dp's* collaborating to find information contributed by a third party.

- Although the support count of the pattern is known to the third party, the third party cannot decipher exactly the percentage value of support. Although a rough estimate of the support count can be made.

- The candidate count retrievals and pattern dispersion activities are also considered secure since they are carried out independent of any other party.

- Since global patterns are sent to the respective data parties where this pattern was locally frequent, no party other than the ones who contributed to the data receive the pattern.

Case 3: Arbitrary Fragmentation

The arbitrary fragmentation is the generalized case of the two cases discussed above. The algorithm uses a cryptographic setup similar to that in the case of horizontal fragmentation. The nature of the algorithm only differs in the third stage (choosing candidate frequent 1- itemsets). The rest stages are similar to either the vertical and horizontal cases respectively.

- In the third step, the sequence numbers pertaining to each element are hashed and added to the set as in the second stage. As a result, although the information passes over to the next party, the receiving party cannot decipher the value of the sequence number from the hash, more so they cannot conclude anything about the inputs given by a particular party.

5.4 Impact of Number of Servers on Execution Time

The simulated environment gives a limited scope to trace the execution time, this is because the segregation of the *Tp* and *Dp* activities is not feasible due to the presence of a single processing unit. However it was realized that the execution time increases linearly as the number of servers increase. This increase can be attributed to the increase in communication overhead incurred with the increase in the number of *Dp*'s.

5.5 Impact of POI on Execution Time

It can be noted that execution time is directly proportional to the value of *POI*. A reason for this is, the increase in time required to process the expanded data structure which would be required to store the candidate patterns. The ratio of this increase in execution time is lesser in case of vertical and arbitrary fragmentation due to selective filtering of candidate frequent 1-itemsets. In case of horizontal fragmentation with an

increase in *POI*, the number of candidate global frequent patterns also increases, thereby increasing the execution time.

5.6 Communication Overhead

The communication overhead incurred also plays an important role while analyzing the efficiency of an algorithm in a distributed environment. The following section computes the communication overhead for each of the proposed algorithm. The worst case number of messages transferred in each case can be analyzed.

The number of messages transferred for communicating 1 frequent pattern of length *(min) N* to *N* parties is computed as follows. It is also assumed that all the N parties contribute to the pattern:

Horizontal Fragmentation scenario

Computing total number of distinct sequences	*N+1*
Obtaining frequent pattern vectors	*2N*
Search for candidate global frequent patterns	*2N*
Communication of frequent patterns	N
Total	*6N +1*

Vertical Fragmentation scenario

Computing total number of distinct sequences	*N+1*
Obtaining frequent itemsets	*2N*
Communication and decryption of frequent patterns	*N*
Broadcasting patterns	*N+1*
Total	*5N+2*

Arbitrary Fragmentation scenario

Computing total number of distinct sequences	*N+1*
Obtaining frequent itemsets	*3N+1*
Broadcasting patterns	*N*
Total	*5N+2*

Table 5.4: Comparison of Communication Overhead

It can be seen that the communication overhead is of *O(n)*.Comparing the number of messages and their performance for varying number of servers, it can be seen that rate of growth in case of the horizontal fragmentation is greater than vertical and arbitrarily fragmented scenarios.

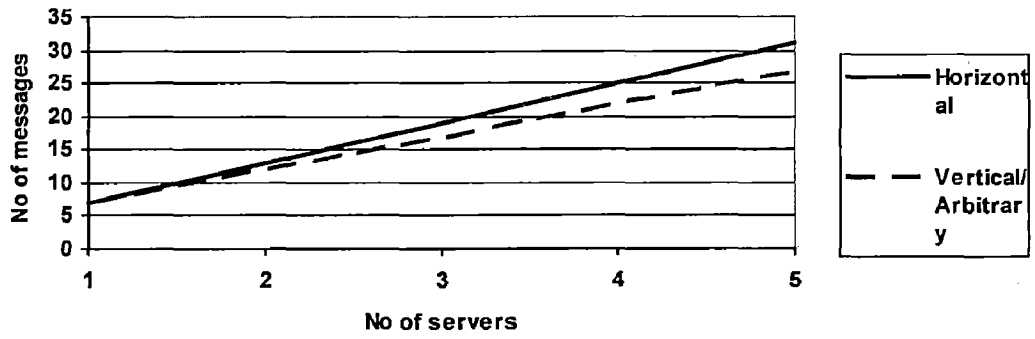


Figure 5.4: Communication overhead (no of msgs) v/s number of servers

CHAPTER 6

Conclusion

6.1 Conclusion

In this work we focused our attention on the problem of privacy preserving sequential pattern mining over distributed progressive databases. There are many existing methods for privacy preservation, however not many are applicable to a distributed progressive database scenario. The limitation of progressive databases is that, neither any prediction of future data could be made, nor could any manipulations to data collected in the past is possible. Hence despite the robustness and accuracy of the existing works, there is little real world scope of their application due to progressive nature of real world databases.

In our work, we have proposed a set of algorithms for discovering frequent patterns from a group of collaborating parties without breaching their individual privacy concerns. The data possessed by the group of parties could map to any of the three dat fragmentation scenarios (viz. Horizontal, vertical, arbitrary). We have therefore proposed a generalized scheme for these scenarios. The proposed work used public key cryptography and randomization as a basis to achieve privacy preservation. We also use a method to block co-occurring frequent patterns in progressive databases, thereby hiding sensitive information of each party from the others. A factor considered for choice of privacy preserving methods was to minimize the distortion of the original data and maintain a maximum possible accuracy.

The algorithms proposed over fragmented databases preserve privacy at four stages: Obtaining sensitive sets of patterns, determining the global threshold, collecting data from the participating parties and distribution of the global patterns. The method used for blocking is used to supplement these algorithms and hides any sensitive sets of co-occurring patterns.

The results achieved display a high amount of accuracy with respect to mining of globally frequent patterns. The communication overhead scales linearly as the

number of parties increase. The amount of information disclosure to each party (including the *Tp*) is limited and hence does not breach privacy under the current set of assumptions. Also the co-occurring pattern blocking method serves as a better method to mask sensitive patterns as compared to the existing methods to hide sensitive frequent pattern sets.

6.2 Suggestions for Future Work

The proposed work dealt with preserving privacy while mining sequential patterns across distributed progressive databases. The data assumed in this scenario was binary market-basket data. In order to widen the scope of application for this problem the proposed algorithms may be modified to suit categorical and numerical data.

According to studies carried out on PISA, a method to prioritize occurrences of patterns was proposed in our previous work [23]. Keeping this idea as a basis, an effort may be made to exploit the progressive nature of the data and proposed prioritizing mechanism in order to preserve privacy.

REFERENCES

[1] J.Han and M.Kamber, "Data Mining:Concepts and Techniques", Series Editor Morgan Kaufmann Publishers, ISBN 1-55860-489-8,2000.

[2] V.S. Verykios, E. Bertino, I.N. Fovino, L.P. Provenza, Y. Saygin, and Y. Theodoridis, "State-of-the-Art in Privacy Preserving Data Mining," *ACM SIGMOD* Record, Vol. 3, No. 1, pp. 50 - 57, 2004.

[3] J.W Huang, C.Y.Tseng, J.C Ou, and M. S.Chen, "A General Model for Sequential Pattern Mining with a Progressive Database," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 20, No. 9, pp. 1153 - 1167, 2008.

[4] Ali Inan, Yücel Saygın, Erkay Savas, Ayça Azgın Hintoglu, Albert Levi, "Privacy Preserving Clustering on Horizontally Partitioned Data", *Proc 22nd Intl Conf on Data Engineering Workshops*, pp. 95 ,2006.

[5] Jaideep Vaidya, Chris Clifton, "Privacy Preserving K Means Clustering over Vertically Partitioned Data," *Proc of the 8th Intl Conf on Knowledge discovery and data mining (SIGKDD)*, Canada, pp. 206 - 215, 2003.

[6] G.Jagannathan and R.Wright, "Privacy-Preserving Distributed *k*-Means Clustering over Arbitrarily Partitioned Data," *Proc of* 11th *Intl Conf on Knowledge discovery and data mining* (*SIGKDD*), USA, pp. 593 - 599, 2005.

[7] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc.11th Int'l Conf. Data Engg. (ICDE '95)*, Taiwan, pp. 3-14, 1995.

[8] M.J.Zaki. "Sequence Mining in Categorical Domains: Incorporating Constrains," *Proc.9th Int. Conf. on Information and knowledge management*, USA, pp. 422 – 429, 2000.

[9] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential Pattern Mining Using a Bitmap Representation," *Proc. 8th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD'02)*, pp. 429-435, 2002.

[10] C.Aggarwal and Philip S. Yu, "Privacy-Preserving Data Mining Models and Algorithms", Springer, New York Inc, ISBN: 0-387-70991-8.

[11] W. Du and Z. Zhan," Using Randomized Response Techniques for Privacy-Preserving Data Mining," *Proc of 9th Intl conf on Knowledge discovery and data mining(SIGKDD)*, USA, pp 505-510, 2003.

[12] J. Vadya and C. Clifton, "Privacy Preserving Association Rule Mining in Vertically Partitioned Data", *Proc of 8th Intl Conf on Knowledge Discovery and Data Mining (SIGKDD)*, Canada, pp. 639-645, 2002.

[13] O. Goldreich, S. Micali, A. Wigderson, *"How to play any mental game—a completeness theorem for protocols with honest majority", Proceedings, 19th ACM Symposium on the Theory of Computing,* pp. 218–229, 1987.

[14] B. Pinkas, "Cryptographic techniques for privacy preserving data mining", *ACM SIGKDD Explorations Newsletter, Vol.4 No.2, pp.12 - 19, 2002.*

[15] J. Zhan, "Using Homomorphic Encryption For Privacy-Preserving Collaborative Decision Tree Classification," *Computational Intelligence and Data Mining (CIDM)*, USA, pp. 637-645, 2007.

[16] J. Zhan, L. Chang, and S. Matwin, "Privacy-preserving collaborative sequential pattern mining," *Workshop on Link Analysis, Counter-terrorism, and Privacy in conjunction with SIAM Intl Conf. on Data Mining*, USA , pp. 61-72, 2004.

[17] W Ouyang, H Xin and Q Huang , "Privacy Preserving Sequential Pattern Mining Based on Data Perturbation," *6th Intl Conf on Machine Learning and Cybernetics 2007(ICMLC 2007)*, China, pp. 3239 - 3243, 2007.

[18] W. Ouyang and Q. Huang, " Privacy Preserving Sequential Pattern Mining Based on Secure Two-Party Computation ," *5th Intl Conf on Machine Learning and Cybernetics 2006(ICMLC 2006)*, China, pp. 1227 - 1232, 2006.

[19] V.Kapoor, P.Poncelet and F.Trousset, "Privacy preserving sequential pattern mining in distributed databases," *Proc of 15th ACM Intl. Conf. on Info and Knowledge Mgmt (CIKM 2006)*, pp. 758 – 767, 2006.

[20] J. Zhan, "Using Homomorphic Encryption and Digital Envelope Techniques for Privacy Preserving Collaborative Sequential Pattern Mining," *IEEE Intl Conf on Intelligence and Security Informatics (ISI 2007)*, Germany, pp. 331 - 334, 2007.

[21] O. Abul. "Hiding Co-Occurring Frequent Itemsets," *2nd Intl Workshop on Privacy and Anonymity in the Info. Soc. (PAIS'09)*, Russia, 2009.

[22] *http://www.datasetgenerator.com/*

[23] A. Mhatre, M. Verma and D. Toshniwal, "Extracting Sequential Patterns from Progressive Databases: A Weighted Approach," *Int'l Conf. on Computer Design and Applications(ICCDA 2009)*, Singapore,2009

LIST OF PUBLICATIONS

[1] **Amruta Mhatre,** Dr. Durga Toshniwal, "Hiding Co-occurring Sensitive Patterns from Progressive Databases," *Int'l Conf on Methods and Models in Computer Science 2009 (ICM2CS)*, JNU, Delhi, India (Communicated).

[2] **Amruta Mhatre,** Dr. Durga Toshniwal, "Privacy Preserving Scheme for Sequential Pattern Mining Over Fragmented Progressive Databases", *Int'l Journal of Advancements in Computing Technology(IJACT)* (Communicated, to be published by AICIT, South Korea).

APPENDIX A: SOURCE CODE LISTING

Code for Implementing Data Party in Vertically Fragmented Scenario

```
import java.io.*;
import java.rmi.RemoteException;
import java.sql.*;
import java.math.BigInteger;
import java.net.*;
import java.util.*;
import javax.net.ssl.SSLServerSocket;
import ppvp2.ServerInfo;
import sun.nio.ch.SocketAdaptor;
import javax.crypto.*;
import java.security.spec.*;
import javax.crypto.spec.*;

public class DataImpl extends java.rmi.server.UnicastRemoteObject implements DataIntf,Runnable
{
  String type;
  int poi;
  int currTime;
  double minSup;
  Pisa p;
  Obj t;
  Integer myno;
  String cip;
  int cport;
  Vector<ServerInfo> servers;
  ServerSocket sock;
  RSA1 r;

  public DataImpl (String s,String fname)  throws java.rmi.RemoteException        //constructor for Dp
  {
    super();
    t=new Tp();
    type = s;
    p = new Pisa();
    myno = new Integer(type);
    r =new RSA1(type);
    try
    {
      ObjectInputStream oin = new ObjectInputStream(new FileInputStream(fname));
      servers = (Vector<ServerInfo>) oin.readObject();
    }
    catch(Exception ex)
    {
      System.out.println("Error while storing to file....."+ex);
      ex.printStackTrace();
    }
  }

public void putParam (int Poi,double MinSup)  throws java.rmi.RemoteException     // sets parameters
  {
    poi = Poi;
    minSup = MinSup;
    p.setParam(Poi,MinSup,type);
  }

public ArrayList process()
  {
    ArrayList f=new ArrayList(p.process());
```

```
    r.encrypUsingtPub (f);
    return f;
  }

public void getSeq (ArrayList x)  throws java.rmi.RemoteException
  {
    if(x == null)
    {
      x=new ArrayList();
    }
    ArrayList sequences=new ArrayList<Integer>();
    int lowerBound = this.currTime - poi + 1;
    try
    {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection("Jdbc:Odbc:Amruta"+type);
      Statement stmt  = con.createStatement();
      ResultSet rs = null;
      rs = stmt.executeQuery("SELECT DISTINCT seqno FROM Table4 where timestamp<="+ currTime +" and
timestamp >"+ lowerBound);
      while(rs.next())
      {
        Integer i = Integer.parseInt(rs.getString(1));
        sequences.add(i);
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
    for(int i=0;i<sequences.size();i++)
    {
      Integer c = (Integer)sequences.get(i);
      int c1 = c.valueOf(c);
      int flag = 0;
      if(x != null)
        for(int j=0; j<x.size();j++)
        {
          Integer d =(Integer)x.get(j);
          int d1=d.valueOf(d);
          if(c1==d1)
          {
            flag=1;
            break;
          }
        }
      if(flag == 0)
      {
        x.add(c);
      }
    }
    if(myno + 1 < servers.size())
      sendThis(myno+1,x);
    else
    {
      try
      {
        Socket scli=new Socket(cip,cport);
        Integer i = x.size();
        ObjectOutputStream cout = new.ObjectOutputStream(scli.getOutputStream());
        cout.writeObject(i);
      }
```

```
        catch(Exception ex)
        {
          ex.printStackTrace();
        }
      }
    }

public void run()
  {
    while(true)
    {
      try
      {
        Socket cc=sock.accept();
        ObjectInputStream ins=new ObjectInputStream(cc.getInputStream());
        Object m=ins.readObject();
        if(m instanceof String)
        {
          String k=(String)m;
        }
        else if(m instanceof SendList)
        {
          SendList x =(SendList)m;
          x.nodeList.remove(0);
          getPat(x);
        }
        else if(m instanceof ArrayList)
        {
          ArrayList g =(ArrayList)m;
          this.getSeq(g);
        }
        else if (m instanceof BlockSet)
        {
          BlockSet b = BlockSet(m);
          this.setSensitiveBlock(b)
        }
      }
      catch(Exception ex)
      {
        ex.printStackTrace();
      }
    }
  }

public void getPat  (SendList x)  throws java.rmi.RemoteException
  {
    try
    {
      if(x.nodeList.size()>0)
      {
        this.decryptEncryptAnd Send(r,x);
        Integer t=x.nodeList.get(0);
        sendThis(t,x);
      }
      else
      {
        String s = r.decryptUsingPriv(x.pattern);
        for(int i =0;i< x.bnodeList.size();i++)
        {
          Integer t=x.bnodeList.get(i);
          sendThis(t,s);
        }
```

```
        try
        {
          Socket scli=new Socket(cip,cport);
          String ok="go ahead";
          ObjectOutputStream cout=new ObjectOutputStream(scli.getOutputStream());
          cout.writeObject(ok);
        }
        catch(Exception ex)
        {
          ex.printStackTrace();
        }
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
}

public void setSensitiveBlock (BlockSet b)
{
    RSA1 r1 = new RSA("ThirdParty",myno);
    ArrayList x[] = new ArrayList[20];
    for(int i = 0; i<20 ;i++)
    {
      x[i] = new ArrayList<String>();
    }
    try
    {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection("Jdbc:Odbc:Amruta"+dbNo);
      Statement stmt = con.createStatement();
      ResultSet rs = null;
      rs = stmt.executeQuery("select * from block");
      while(rs.next())
      {
        int set = Integer.parseInt(rs.getString("set"));
        String element= rs.getString("pattern");
        x[set].add(element);
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
    for(int i = 0; i <20;i++)
    {
      if(x[i].size()!=0)
      {
        String a[]=new String[x[i].size()];
        for(int j = 0; j<x[i].size();j++)
        {
          a[j]=(String)x[i].get(j);
          a[j]=r1.encryptUsingPub(a[j].getBytes();
        }
        block b1 = new block(a);
        b.add(b1);
      }
    }
}
if(myno + 1 < servers.size())
  sendThis(myno+1,b);
```

```
    else
    {
      try
      {
        Socket scli=new Socket(cip,cport);
        ObjectOutputStream cout=new ObjectOutputStream(scli.getOutputStream());
        cout.writeObject(b);
      }
      catch(Exception ex)
      {
        System.out.println("Error while sending to client....."+ex);
        ex.printStackTrace();
      }
    }
}
```

Code for Implementing Third Party in Vertically Fragmented Scenario

```
import java.io.*;
import java.rmi.*;
import java.util.*;
import java.net.*;
import sun.nio.ch.ServerSocketAdaptor;

public class ThirdParty{
   int maxTime;
   static int currentTime= 0;
   static double minSup;
   static int seq[] = new int[2] ;    //stores no of dist seq in each party  //change here
   static int poi;
   static DataIntf c[] = new DataIntf[2];//change here
   static gpatMiner g = new gpatMiner();
   ArrayList fList[] = new ArrayList[2];  //change here
   static SeqList Gdata[];
   static ArrayList<node> pLabel[]=new ArrayList[2];
   ServerSocket csock;
   boolean ok=false;
   public static int freqCnt = 0;
   RSA1 t =new RSA1("ThirdParty");
  public static ArrayList blockset = new ArrayList<block>();

ThirdParty (double x,int y)                    //sets minimum support and poi
   {
     pLabel[0] = new ArrayList<node>();
     pLabel[1] = new ArrayList<node>();
     minSup = x;
     poi = y;
     try
     {
       csock=new ServerSocket(12345);
       for(int i = 0; i < 2; i++)                    //change here
       {
         String s = new String(String.valueOf(i));
         c[i] = (DataIntf)Naming.lookup("rmi://localhost/Pisa"+s);
         c[i].setCparams(InetAddress.getLocalHost().getHostAddress(), 12345);
         seq[i] = 0;
       }

     }
     catch (Exception e)
     {
       e.printStackTrace();
```

```
    }
  }

public void timeCalc ()
  {
    try
    {
      c[0].setSensitiveBlock();
      waitforc();
      while(currentTime <= maxTime)
      {
        for(int i = 0 ; i < 2 ; i++)
        {
          c[i].putTime(currentTime);
          fList[i]=c[i].process();
        }
        ArrayList a=new ArrayList();
        c[0].getSeq(a);
        waitforc();
        g.calcSeq();
        getData(fList);
        ArrayList x=g.gpatMine(Gdata);
        Send(x);
        currentTime++;
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }

public void Send (ArrayList x)
  {
    for(int t=0; t<x.size();t++)
    {
      genRandom(((SendList)x.get(t)).nodeList);
      Integer sendParty=((SendList)x.get(t)).nodeList.get(0);
      try
      {
        ((SendList)x.get(t)).nodeList.remove(0);
        c[sendParty.intValue()].getPat((SendList)x.get(t));
        waitforc();
      }
      catch (Exception ex)
      {
        ex.printStackTrace();
      }
    }
  }

public ArrayList genRandom(ArrayList s)    //to be sent when decrypting patterns
  {
    for(int i=0;i<s.size();i++)
    {
      int rand=(int)((Math.random()*10)%s.size());
      Integer temp =(Integer) s.get(rand);
      s.remove(rand);
      s.add(temp);
    }
    return s;
  }
```

```
private void waitforc()
  {
    try
    {
      Socket sc=csock.accept();
      ObjectInputStream sin=new ObjectInputStream(sc.getInputStream());
      Object gh=sin.readObject();
      if(gh instanceof String)
      {
        String s =(String)gh;
      }
      else if(gh instanceof Integer)
      {
        Integer i=(Integer)gh;
        g.Gseq=Integer.valueOf(i);
        System.out.println("Total no of global sequences are" +g.Gseq);
      }
      else if (gh instanceof BlockSet)
      {
        BlockSet b =BlockSet(gh);
        decryptBlock(b);
      }
    }
    catch(Exception ex)
    {
      System.out.println("Error "+ex);
      ex.printStackTrace();
    }
  }
}
```

Code for Global M-ary Tree in Vertical Fragmentation

```
import java.util.*;
import java.sql.*;
import java.util.ArrayList;

public class Mtree
{
  String label;
  int seq[];                    //stores sequence nos
  int timeStamp [];             //stores  timestamps
  Mtree link[];                 //stores links to children
  Mtree parent = null;          //parent of the current node
  int current, noLink;
  int delFlag;
  ArrayList freqpattern = new ArrayList<dbOp>();
  public static int poi=0;
  int origts [];
  double bthresh;
  String pattern = new String();
  int bcandid;

public Mtree (String lab, int s, int t,int ot)    // creates an object of type MTree
  {
    delFlag = 0;
    label = lab;
    seq = new int[1];
    seq[0] = s;
    current = 1;                //stores no of sequences
```

```
    timeStamp = new int[1];
    timeStamp[0] = t;
    link = new Mtree[1];
    link[0] = null;
    noLink = 0;                    //stores no of children
    origts = new int[1];
    origts[0] = ot;
    bcandid = 0;
  }

public void addToNode (int s, int t,int ot)
  {
    int x;
    current++;
    int temp[] = new int[current];
    for( x = 0; x < current - 1; x++)
      temp[x] = seq[x];
    temp[x] = s;
    seq = temp;
    temp = new int[current];
    for(x = 0; x < current - 1; x++)
      temp[x] = timeStamp[x];
    temp[x] = t;
    timeStamp = temp;
    temp = new int[current];
    for( x = 0; x < current - 1; x++)
      temp[x] = origts[x];
    temp[x] = ot;
    origts = temp;
  }

public void addToNode (Mtree z)  // add a child node
  {
    int x ;
    noLink++;
    Mtree temp[] = new Mtree[noLink];
    for(x = 0; x < noLink -1; x++)
      temp[x] = link[x];
    temp[x] = z;
    link = temp;
    z.parent = this;
  }

public void insert (Mtree node,int currentTime,ArrayList blockset)      //modified insert function
  {
    int flagLabel, flagSeq;
    if(node.label.equals("root"))
    {
      if(gpatMiner.Gdata!= null)
      {
        for(int i=0; i<gpatMiner.Gdata.size(); i++)
        {
          SeqList l =(SeqList)gpatMiner.Gdata.get(i);
          for(int j=0; j<l.nodeList.size(); j++)
          {
            node z=(node)l.nodeList.get(j);
            int sequenceNo = l. timeStamp;  //here field timestamp is used for seqno
            int length = (Integer)z.name.length();
            int len = (int)Math.pow(2,length);
            String []element = new String[len];        //to store combinations
            element = node.combinations(z.name);
            for(int x=0; x<len; x++)
```

```
        {
          flagLabel = 0;
          if(element[x]!=null)
          {
            for(int a=0 ; a < node.noLink ; a++)//check for node's chidren
            {
              flagSeq = 0;
             if((element[x].equals(node.link[a].label)))
              {
                flagLabel =1;
                for(int b = 0; b < node.link[a].current; b++)
                {
                  if(sequenceNo==node.link[a].seq[b])
                  {
                    flagSeq = 1;
                    if(node.link[a].origts[b]< z.val)
                    {
                      node.link[a].timeStamp[b] = z.val;
                      node.link[a].origts[b] = z.val;
                      break;
                    }
                    if(node.link[a].origts[b]>=z.val)
                    {
                     break;
                    }
                  }
                }
                if(flagSeq==0 && flagLabel==1)
                {
                  int add = 1;
                  {
                    node.link[a].addToNode(sequenceNo,z.val,z.val);
                   }
                }
              }
            }
            if(flagLabel== 0)
            {
              int flag =0;
              node.addToNode(new Mtree(element[x],sequenceNo,z.val,z.val));
            }
          }
        }
      }
    }
  }
}
else
{
  for(int i = 0 ; i < node.current && node.current!=0 ; i++)
  {
    if(node.timeStamp[i] <= currentTime-poi)
    {
      for(int e=i+1 ; e < node.current ; e++)
      {
        node.seq[e-1]=node.seq[e];
        node.timeStamp[e-1]=node.timeStamp[e];
        node.origts[e-1]=node.origts[e];
      }
      node.current =node.current-1;
      i-- ;
      if(node.current==0 )
```

```
    {
       node.delFlag = 1;
    }
    continue;
  }
  for(int j = 0 ; (gpatMiner.Gdata!= null)&&(j < gpatMiner.Gdata.size()); j++)
  {
    SeqList l=(SeqList)gpatMiner.Gdata.get(j);
    if(l.timeStamp == node.seq[i])
    {
      for(int k =0; k < l.nodeList.size(); k++)
      {
        node z = (node)l.nodeList.get(k);
        int sequenceNo = l.timeStamp;  //here timestamp is used for seqno
        int length = (Integer)z.name.length();
        int len = (int)Math.pow(2,length);
        String []element = new String[len];        //to store combinations
        element = node.combinations(z.name);
        for(int m=0; m<element.length ;m++)
        {
          flagLabel = 0;
          if(element[m]!=null)
          {
            if((node.notLiesOnRoute(element[m])) && (z.val>node.origts[i]))
            {
              for(int x = 0; x < node.noLink && node.link[x]!=null; x++)
              {
                if(node.link[x].label.equals(element[m]))
                {
                  flagLabel = 1;
                  flagSeq = 0;
                  for(int y = 0; y<node.link[x].current;y++)
                  {
                    if(sequenceNo == node.link[x].seq[y])
                    {
                      flagSeq = 1;
                      if(z.val>node.link[x].origts[y])
                      {
                        node.link[x].timeStamp[y]=node.timeStamp[i];
                        node.link[x].origts[y]=z.val;
                        if(node.link[x].bcandid == 1)
                        {
                          node.link[x].updateTime(node.link[x].seq[y],node.
                          timeStamp[i],blockset);
                        }
                      }
                      if(z.val <=node.link[x].origts[y])
                      {}
                      break;
                    }
                  }
                  if(flagSeq == 0)
                  {
                    int flagAdd = 0;
                    if(node.link[x].bcandid == 1)
                      {
                        flagAdd = chkLock(node.link[x].pattern,blockset);
                      }
                    else
                        flagAdd = 1;
                      if(flagAdd == 1)
                      {
```

```
                                int val =0;
                                node.link[x].addToNode(sequenceNo,node.timeStamp[i],z.val);
                                if(node.link[x].bcandid ==1)
                                {
                                   node.link[x].addseq(node.link[x].pattern,sequenceNo,
                                   node.timeStamp[i],blockset);
                                   if(node.link[x].current>= bthresh)
                                   {
                                      val = 1;
                                      node.link[x].updateThresh(val,node.link[x].pattern,blockset);
                                   }
                                }
                               }
                            }
                          }
                         if(flagLabel == 0)
                         {
                            String s =node.backTraverse1();
                            s =  s +" "+ element[m];
                            int sl =s.length();
                            int val = chkSensitive(s,blockset);
                            if(val == 1)
                            {
                               int flagAdd =1;
                               if(bthresh ==1)
                               {
                                   flagAdd =chkLock(s,blockset);
                               }
                              if(bthresh > 1)
                              {
                                 flagAdd =1;
                              }
                              if(flagAdd ==1)
                              {
                                 node.addToNode(new Mtree(element[m],sequenceNo,
                                 node.timeStamp[i],z.val));
                                 node.addtoblock(s,sequenceNo,node.timeStamp[i],blockset);
                                 node.link[node.noLink-1].bcandid=1;
                                 node.link[node.noLink-1].pattern=node.link[node.noLink-1].backTraverse1();
                              }
                              if(bthresh ==1)
                              {
                                 node.link[node.noLink-1].updateThresh(val, s, blockset);
                              }
                            }
                            if(val == 0)
                               node.addToNode(new node. Mtree(element[m],sequenceNo,timeStamp[i],z.val));
                         }
                       }
                     }
                   }
                 }
               }
             }
           }
         }
       }
     }
```

public void traverse (int time,Mtree m,ArrayList blockset) // traverses Mtree in post order

```
  {
```

```
    int no=0;
    if(m.noLink!=0)
    {
     while(no < m.noLink)
      {
        traverse(time,m.link[no],blockset);

        if(m.link[no].delFlag == 1)
        {
            int i=0;
          m.link[no] = null;
          for(i = no; i < m.noLink - 1; i++)
          {
            m.link[i] = m.link[i+1];
          }
          m.noLink--;
          no--;
        }
        no++;
      }
     insert(m,time,blockset);
    }
    else
      insert(m,time,blockset);
  }

public void postTraverse (int time,Mtree m,double threshold,ArrayList freqpattern)
  {                                                                    // calculates frequent patterns
    int no=0;
    {
     while(no < m.noLink)
      {
        postTraverse(time,m.link[no],threshold,freqpattern);
        no++;
      }
     if(m.calc() >= threshold)
     {
        String pan=m.backTraverse();
        int nit, count;
        nit = 0;
        count = 0;
        nit = pan.indexOf(" ", nit);
        while(nit!=-1)
        {
          nit++;
          nit = pan.indexOf(" ", nit);
          count++;
        }
        if(count >= 2)
        {
          PisaClient.fcnt++;
          for(int i =0 ;i<m.current;i++)
          {
            dbOp d =new dbOp(m.seq[i],pan);
           freqpattern.add(d);
          }
        }
     }
    }
  }

public String backTraverse()              // back traces the path from node till root
```

```
  {
    int j;
    int i = 1;
    String pattern = new String();
    Mtree temp = this;
    while(!(temp.label.equals ("root")))
    {
      temp = temp.parent;
    }
    return pattern;
  }

public double calc()                          //calculates support count
  {
    return this.current;
  }

public void addseq (String pattern,int seqno,int time,ArrayList b)     //adds sequence to pattern in blockset
  {
    String s =pattern;
    tuple t =new tuple(seqno,time);
    for(int i =0 ;i< b.size();i++)
    {
      block c =(block)b.get(i);
      for(int j = 0; j < c.set.size(); j++)
      {
        String str = (String)c.set.get(j);
        if(s.equals(str))
        {
          nodeList x = (nodeList)c.minSeq.get(j);
          x.seq.add(t);
        }
      }
    }
  }

public int chkSensitive (String s,ArrayList b)          //checks if the given string exists in the blockset
  {
    for(int i = 0; i< b.size(); i++)
    {
      block c =(block)b.get(i);
      for(int j = 0; j < c.set.size(); j++)
      {
        String t = (String)c.set.get(j);
        if(t.equals(s))
        {
          return 1;      // if s exists in current blocked set
        }
      }
    }
    return 0;
  }

public int chkLock (String str,ArrayList b)     //checks if  a sequence can be added to a node
  {
    String s =str;
    int flag[] = new int[b.size()];    // checks if pattern s exists in ith blocked set
    int lockset[] = new int[b.size()]; // set to be locked
    for(int i = 0; i< b.size(); i++)
    {
      int count = 0;
      block c =(block)b.get(i);
```

```
        for(int j = 0; j < c.set.size(); j++)
        {
          int l;
          String t = (String)c.set.get(j);
          if(t.equals(s))
          {
            flag[i] = 1;      // if s exists in current blocked set
          }
          else
          {
            l = (Integer)c.thresh.get(j);
            if(l==1)
              count++;
          }
        }
        if(count == c.set.size() - 1)              // what for single element set
        {
          lockset[i] = 1;
        }                                  //need to lock this set
      }
      for(int i = 0 ; i < b.size() ;i++)
      {
        if((flag[i] == 1) && (lockset[i] == 1))
          return 0;                                        //cannot add
      }
      return 1;
    }

public void chkBcandid (ArrayList b)          //checks if a pattern corresponds to a sensitive pattern
    {
      if(bcandid ==0)
      if(!(label.equals ("root")))
      {
        String s = backTraverse1();
        if(b!=null)
        for(int i = 0; i< b.size();i++)
        {
          block bl =(block)b.get(i);
          for(int j = 0;j< bl.set.size();j++)
          {
            String t =(String)bl.set.get(j);
            t = t.substring(0,t.length()-1);
            if(s.equals(t))
            {
              this.bcandid=1;
              break;
            }
          }
        }
      }
    }
}
```

Code for class gpatMiner in Horizontally Fragmented Scenario

```
import java.util.*;
import java.sql.*;

public class gpatMiner
{
  int Gseq = 0;
  int initial = 0;
```

```
  ArrayList gData;
  double threshold;
  static ArrayList partyPat[] = new ArrayList[2];  //keeps track of sequences contributed by each party
  static ArrayList freqPat = new ArrayList<patEle>();
  ArrayList tempPat = new ArrayList<patEle>();
  ArrayList sendPat = new ArrayList<String>();
  public ArrayList gblockset = new ArrayList<block>();

gpatMiner()
  {
    for(int i=0;i<partyPat.length;i++)
      partyPat[i]=new ArrayList<dbOp>();
  }

public void calcSeq()
  {                                                    //calculates global threshold
    int maxT =0;
    for(int i=0; i<PisaClient.seq.length ;i++)
    {
       maxT = PisaClient.seq[i]+maxT;
    }
    maxT = maxT - ThirdParty.rno;
    Gseq = maxT;
    threshold  = ThirdParty.minSup * Gseq;
  }

public void arrPat(ArrayList fList,int party)
  {                                                          //adds recently acquired freqpatterns to
previously stored
    int flag = 0;                                          //frequent patterns
    if(fList != null)
    {
      for(int i = 0; i <fList.size(); i++)
      {
        flag = 0;
        dbOp p = (dbOp)fList.get(i);
        if(freqPat != null)
        {
          String name = null;
          int count = 0;
          for(int j=0;j< freqPat.size();j++)
          {
            patEle q=(patEle)freqPat.get(j);
            name = q.label;
            count = q.count;
            if(p.pat.equals(q.label))
            {
              flag=1;
              q.count = q.count + p.count;
              if(q.count == 0)
              {
                freqPat.remove(j);
                j--;
              }
              else
              {
                int flag1=0;
                for (int c=0;c<q.partyNames.size();c++)
                {
                  Integer party1 = (Integer)q.partyNames.get(c);
                  System.out.println(party1);
                  if(party1 == party)
```

```
                    {
                        flag1=1;
                        break;
                    }
                }
                if(flag1 == 0)
                {
                    q.partyNames.add(party);
                }
            }
            break;
        }
    }
    if(flag == 0)
    {
        if(name != null)
        {
            patEle x = new patEle(p.pat,p.count,party);
            freqPat.add(x);
        }
    }
    flag=0;
  }
 }
 if(freqPat.size()==0)
 {
    for(int i=0;i< fList.size();i++)
    {
        dbOp d=(dbOp)fList.get(i);
        if(!d.pat.equals(null))
        {
            if(d.count > 0)
            {
                patEle e = new patEle(d.pat, d.count, party);
                freqPat.add(e);
            }
        }
    }
 }
 if (fList.size()!=0)
 {
    tempPat.clear();
    int x = freqPat.size();
    for(int i=0;i<x;i++)
    {
        patEle p = (patEle)freqPat.get(i);
        patEle q = new patEle(p.label,p.count,p.partyNames);
        tempPat.add(q);
    }
 }
 }
}

public ArrayList[] calcGpat1()
 {                                          //stage 1 to calculate global frequent patterns
    ArrayList arr[] = new ArrayList[2];
    arr[0] = new ArrayList<String>();
    arr[1] = new ArrayList<String>();
    if(freqPat!=null)
    for(int i=0;i<freqPat.size();i++)
    {
        patEle p = (patEle)freqPat.get(i);
```

```
        if(p.count >= threshold)
        {
           chkBlock(p.label,gblockSet);
           sendPat.add(p.label);
           freqPat.remove(i);
           i--;
        }
        else
          for(int j=0;j<partyPat.length;j++)                    //assign pattern to parties
          {
            int flag = 0;
            for(int z=0;z<p.partyNames.size();z++)
            {
              if((Integer)p.partyNames.get(z) == j)
              {
                flag = 1;
                break;
              }
            }
            if(flag == 0)
            {
              arr[j].add(p.label);
            }
          }
      }
      return arr;
    }

public void calcGpat2(ArrayList a[])                            //gets candidate patterns from other Dps
 {
    int flag;
    ArrayList subset = new ArrayList<patEle>();
    if(a != null)
      subset  = getFromDp(a);
      if(subset != null)
        for(int j=0;j<subset.size();j++)
        {
          flag = 0;
          patEle d = (patEle)subset.get(j);
          if (freqPat!=null)
            for(int i=0;i<freqPat.size();i++)
            {
              patEle e = (patEle)freqPat.get(i);
              if(d.label.equals(e.label))
              {
                e.count = e.count +d.count;     //update support counts
                flag = 1;
                break;
              }
            }
            if(flag == 0)
            {
              freqPat.add(d);
            }
        }
    }

    public void calcGpat3()
    {                                                           //computes frequent patterns in round 2
      if(freqPat!= null)
        for(int i = 0;i<freqPat.size();i++)
```

```
    {
      patEle d = (patEle)freqPat.get(i);
      if(d.count >= threshold)
      {
        int chk = chkBlock(d.label , gblockSet )
        if (chk ==1) sendPat.add(d.label);
      }
    }
    freqPat.clear();
    for(int i=0;i<tempPat.size();i++)
    {
      patEle p = (patEle)tempPat.get(i);
      freqPat.add(tempPat.get(i));
    }
  }

  public void send()                          // decides which pattern goes to which party
  {
    for(int i =0; i <sendPat.size() ; i++)
    {
      String s = (String)sendPat.get(i);
      for(int j = 0; j < partyPat.length;j++)
      {
        for(int k = 0 ; k<partyPat[j].size() ; k++)
        {
          dbOp d = (dbOp)partyPat[j].get(k);
          if(d.pat.equals(s))
          {
            try {
                 ThirdParty.c[j].sendPattern(s);
                }
            catch(Exception e)
            {
              e.printStackTrace();
            }
            break;
          }
        }
      }
    }
    sendPat.clear();
  }
 }
}
```

Code for Implementing Third Party in Horizontal Fragmentation

```
import java.rmi.*;
import java.net.*;
import java.io.*;
import java.util.*;
import sun.nio.ch.ServerSocketAdaptor;

/* connects to multiple Dps manages time and other mining activities*/

public class ThirdParty
{
  int maxTime;
  static int currentTime= 0;
  static double minSup;
  static int seq[] ;                          //stores no of dist seq in each party
  static int poi;
```

```
static DataIntf c[];
gpatMiner g;
ArrayList freqPat[];
ServerSocket csock;
boolean ok = false;


ThirdParty (double x, int y, int t)
  {                                                    //sets minimum support and poi
    minSup = x;
    poi = y;
    int no=t;
    seq =new int[t];
    c = new DataIntf[2];
    g = new gpatMiner();
    try
    {
      csock=new ServerSocket(12345);
      for(int i=0;i<no;i++)
      {
        c[i] = (DataIntf)Naming.lookup("rmi://localhost/Dp"+s);
        c[i].setCparams(InetAddress.getLocalHost().getHostAddress(), 12345);
        seq[i]=0;
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }


public void timeCalc()
{
    int flag=0;
    try{
      while(currentTime<= maxTime)
      {
        flag=0;
        g.ginit();
        for(int i=0;i<2;i++)
        {
          c[i].putTime(currentTime);
          seq[i] = c[i].getNoSeq(currentTime, minSup, poi,ThirdParty.rno);
          c[i].process();
          g.gpatMine1(i);
        }
        g.calcSeq();
        ArrayList arr[] = g.calcGpat1();
        if(arr!= null)
          g.calcGpat2(arr);
        g.calcGpat3();
        g.send();
        currentTime++;
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
```