DELTA : A Fast Algorithm for Incremental Mining of Association Rules

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Shrividya

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

June 1997

Contents

Acknowledgments

I take this opportunity to thank my project guide, *Dr. Jayant Haritsa*, for all the help and encouragement he has given me during the project. I am very grateful for his suggestions and advice at every stage of the project. I am indebted to him for providing me with very good facilities to work with in the Database Systems Lab.

I would like to thank all my lab mates, Binto, Mouli, Nalin, Nars, Ramesh, and Shri for their suggestions and support.

I am grateful to all my teachers here in the institute and elsewhere, who are responsible for creating the hunger for knowledge in me.

I would like to take this opportunity to thank all my friends in IISc., especially Laksh, Anu, and Apu, who have made my stay in IISc., a truly memorable one.

Finally, I would like to thank my parents, elders, and all my well wishers, who are responsible for what I am today, for their love and encouragement.

Abstract

Data mining is an emerging field in the database technology. The goal of data mining is knowledge discovery, that is, to excavate information from historical organizational databases that can be used to guide business strategies and decisions. We are currently developing a software tool called **MINTO**, which is a data mining tool customized for *m*anufacturing data. In the first phase, which is currently operational, we have provided a comprehensive set of algorithms for discovering *a*ssociation rules – A rule of the form, if a person buys coffee he is likely to buy sugar too in 60% of the cases is association rule. In particular, we have implemented AIS, APRIORI, APRIORITID, DHP, and PARTITION algorithms.

Since manufacturing databases are typically huge, of the order of *terabytes* in many cases, it is important to provide mechanisms for reducing the amount of data that is scanned in the mining process. For achieving this reduction, we have considered two methods: (1) Sampling and (2) Incremental mining, incremental mining uses the rules obtained in the original database and finds of the rules for the entire database, after some modifications are done for the database. Mining algorithms based on both these reduction methods have been implemented in MINTO. Included in this set is a new technique for incremental mining developed by us called DELTA (Differential Evaluation of Large iTemset Algorithm). Using a detailed simulation model, we analyze the performance characteristics of *DELTA*. Our experimental results show that DELTA is significantly faster than previously proposed incremental mining algorithms for a variety of databases.

Finally, two interfaces have been provided for MINTO: a graphical user interface, built using X-toolkit and Motif libraries, as well as a SQL-like query interface.

Chapter 1

Introduction

1.1 Motivation

Organizations world wide have been collecting huge amounts of data over the years. In the competitive business environment, these terabytes of raw data need to be processed to give significant insights to guide the marketing, investment, and management strategies of companies. These historical data can be used to get *patterns*, which will help in making decisions. These patterns may be simple facts such as "coffee powder and sugar are always purchased together", or some previously unknown fact. How are we supposed to excavate these useful information from the gold mines of historical data? *Data mining* is a proposed solution for the problem of *quarrying* useful information from large amounts of data.

Data mining helps in business decision making. For example, once a manager finds out the fact that coffee powder and sugar are purchased together, he can decide to place coffee powder and sugar in adjacent shelves for customer convenience. This will also help him to prepare purchase orders, as he can order for coffee powder and sugar at the same time. Data mining has an enormous breadth of applications, from analyzing customer purchase pattern to analyzing Government decisions, from discovering patterns in health care to discovering reasons for failures in games.

Data mining is a relatively new field from the database perspective. Pattern recognition was done using artificial intelligence and neural network methods from a long time.

Data mining does pattern recognition from a database view point. It has gained prominence only in this decade.

How is data mining different from data querying? Data is retrieved, when a data base is queried. Here the data base manager knows what exactly it is looking for and searches for the data. On the other hand when a person is quarrying (mining) a database, the person does not know as to what exactly he/she is looking for, rather the person asks the mining engine to get some useful information. Hence data mining is considered to be a way to achieve knowledge discovery.

One might need to find different types of patterns in different cases. For example, in the case of supermarket data the person might be interested in finding inter transaction purchase patterns(patterns found in purchases done at a time), whereas in case of advertising strategies, one might be interested in finding out the target set, that is the set of people who are likely to buy the goods. The idea here is to partition the set of people into two categories, the set of people who are likely to buy the product, and the others and this is a sort of classification problem. Hence there are some differences in the types of data mining.

1.2 Classes of Database Mining Problems

Most of the data mining applications belong to three major categories as shown in Figure 1.1. These types are Association Rules, Classification Rules, and Sequencing.

1.2.1 Association

Consider the following example of a supermarket database. In the supermarket the items purchased by a customer is stored as a transaction. Then, if one can find out some inter transaction *association* between the purchased items, it will be very useful for the marketing strategy. One example for such a pattern may be that people buy coffee powder and sugar together in 1% of the transactions, and whenever a person buys coffee, the person buys sugar too in 80% of the cases. Such information can be used for floor

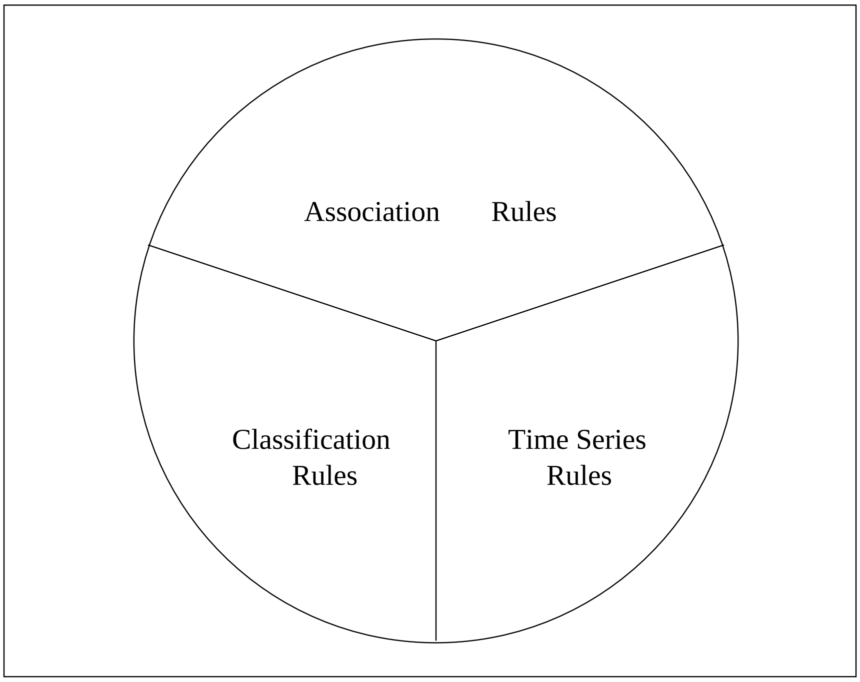

Figure 1.1: Types of Mining Rules

planning, i.e., placement of the items in the supermarket and also while getting supplies. This might help in advertising strategies as well.

1.2.2 Classification

As the name suggests, the *classification* problem deals with finding rules that *partition* the given database into disjoint groups. Store location problem can be taken as an example of classification problem. It is a well known fact that success of a store is determined by the neighborhood characteristics. For example, a pharmacuitical store will flourish, if there is a hospital nearby. Hence, when starting a new branch, the company will be interested in identifying neighborhoods that should be primary candidates for the location of a proposed new store. Other applications include credit approval, determining the target set for advertising campaigns etc.

Most of the algorithms used to mine for *classification* are based on neural network and artificial intelligence techniques.

TransactionId	Potatoes	Onions	Tomatoes	Carrots	Beans
1	N	N	N	N	Y
2	Y	Y	N	Y	Y
3	Y	Y	Y	N	N
4	Y	Y	N	N	Y
5	Y	Y	Y	N	N
6	Y	Y	Y	Y	N
7	Y	Y	Y	N	N
8	Y	N	N	N	Y
9	Y	Y	Y	N	N
10	Y	Y	Y	N	Y

Table 1.1: Example Database

1.2.3 Sequences

Sequences, as the name suggests, is related to temporal data such as stock market and point of sales data. An example of rule in such a case is the following. *When Reliance stock goes up on two consecutive days and ACC stock does not fall during this period, Telco stock goes up the next day* 80% *of the time.* Another example of such a query is *What items are sold in sequence?.*

1.2.4 Example

We have concentrated on finding association rules in this project. Let us see an example, in which data mining is done to find out the association rules. Consider a toy database shown in Table 1.1 as an example.

This example is taken from [Har96]. The example database is pertaining to a small vegetable vendor who sells potatoes, tomatoes, onions, carrots, and beans. The database given is of the customer purchases.

Each transaction has six entries in it. The first entry will be Transaction id, and the next five entries will be the bit fields corresponding to the above mentioned vegetables, Potatoes, Onions, Tomatoes, Carrots, and Beans.

One of the rules obtained by mining this database is

- $Potatoes, Onions \Longrightarrow Tomatoes \mid (60, 75)$

The above rule means that people bought Potatoes, Onions, and Tomatoes in 60 % of the cases and whenever a person bought both Potatoes and Onions, he bought Tomatoes too in 75 % cases.

(Note that the rules obtained above are not really valid, as we have considered a very small database. To actually obtain meaningful rules, we have to have a huge database with thousands of customer records. Then the database will record consistent patterns and will not depict a transient phenomenon.)

1.3 Why and What are Sampling and Incremental Mining?

The goal of database mining is to discover information from large amount of historical data stored in organizations. Developing efficient algorithms for mining has become an active area of research in the database community in the last few years. However, all the current research prototypes are addressed towards commercial retail data, not manufacturing data which is far more complex, larger in size and different in nature.

As the size of the database plays an important role in determining the efficiency of mining algorithms, any method to reduce the size of the database plays a major role in decreasing the CPU time taken to do mining. Two methods to obtain reduction in size are

1. Sampling
2. Incremental techniques

In sampling techniques, a sample of the database is used to search for patterns. In this case, one might not get all the actual patterns or might get some patterns, which are not correct, especially if the sample is not a true representative of the underlying database. This problem can be overcome by using special techniques. Finding out patterns using only a sample set, and later scanning the whole database to check whether the patterns obtained are truly meaningful, is one such technique.

To understand why incremental mining becomes necessary consider the following scenario: A database has been mined and some patterns are recognized and stored. Then some more transactions are added to the database. Because of the addition of these new transactions, some of the original patterns might cease to exist. More over, one might get some new patterns. Now, two options are there to do mining. One way is to mine the whole database again. Otherwise, use the previous information and the additional tuples to do mining. The later technique is called *incremental mining*.

1.4 The MINTO Project

'

A data mining tool, called *MINTO* which is tuned to do mining for manufacturing applications, is being built in the Database Systems Lab in Indian Institute of Science, Bangalore, India. At present, *MINTO* only has the capability of finding association rules. It does sampling and incremental mining as well. This is the first step in the project *MINTO*.

The following things are done as a part of the project.

- A mining tool with a set of classical algorithms
- A new incremental mining algorithm called DELTA
- Sampling to do mining.

The *MINTO* tool contains five classical mining algorithms : AIS [AIS93a], APRIORI [AS94], APRIORITID [AS94], DHP [PCY95], and PARTITION [SON95] to find association rules. All these algorithms are explained in Chapter 3. One algorithm based on sampling, which is fully dealt in Section 4.1 is also implemented in the tool.

Our aim was to do incremental mining more efficiently. The algorithm FUP [CHN+96], further details of which are given in Section 5.2, is the only algorithm to do incremental mining, in the literature, to the best of our knowledge. *MINTO* includes this algorithm as well. DELTA, which is explained in Section 5.3 is a new algorithm proposed by us. We have studied its performance with respect to APRIORI and FUP. The results of our experiments show that DELTA performs much better than FUP in a wide variety of cases and is effective, even when there is skew in the data. These performance evaluation results are in Chapter 6.

We wanted to give a graphical user interface for the tool, as well as to have a SQL like query interface. The graphical user interface is given for the ease of use of the tool. Even the people who do not know SQL can use the tool. But on the other hand, people who are familiar with SQL might want to use the tool without having to use the graphical interface. Moreover, we can run the tool even if X-windows are not supported in the system, as we have this query interface. Query interface makes the embedding of the queries into other languages feasible.

1.5 Organization of the Report

The rest of the report is organized as follows: A discussion of mining for association rules is given in Chapter 2. Some of the algorithms to find association rules are described in Chapter 3. Sampling as related to data mining is dealt with in Chapter 4. A full description of incremental mining can be found in Chapter 5, which describes algorithms FUP and DELTA as well. The analysis of performance of the new algorithm DELTA is described in Chapter 6. Implementation details of the data mining tool are given in Chapter 7. We conclude in Chapter 8 with the summary of our work and suggest possible avenues for future work.

Chapter 2

Mining for Association Rules

Database technology has been used successfully in traditional business data processing. Now, it is being tried in new application domains. One such application domain is *database mining*, known as *data mining* in short.

We start this chapter with explanation of related work going on in database mining. Then we move on to explain model considered. As mentioned previously we have concentrated on mining for association rules. The solution strategy is given and the chapter ends with an example database.

2.1 Related Work

The data mining algorithms mentioned in the introduction are all multi pass algorithms. An algorithm is called k-pass algorithm, if one can specify that the algorithm will terminate in k passes. The first k-pass algorithms, APRIORI and APRIORITID are given in [AS94]. These algorithms are described in Chapter 3. The sampling as again data mining is studied in [PYC96] and [MTI94a]. An algorithm to do data mining using the technique of vertical fragmentation can be found in [SLW96]. A two pass algorithm based on partioning the database is given in [SON95]. This algorithm is explained in Chapter 3. [HF95] and [SA95] deal with *multi level* and *hierarchical* association rules, respectively. Section 3.4 refers to a hash based data mining algorithm, called DHP, which is given in [PCY95]. [CHN+96] came up with the first incremental mining algorithm. Among the classification rules [LSL95] is based on neural network principles. Time series is addressed in [ALS+95] and shapes of histories in [APW+95].

Most of the initial work in this area has concentrated on discovering association rules. Later, interest has shifted from finding association rules to other application areas like classification and sequential data mining. Most of the classification algorithms are based on neural networks and artificial intelligence. Now, researchers are thinking of moving to object oriented data mining. There are some groups of researchers who are looking at applying mining for object oriented data, at present.

Though there are lots of studies in the area of mining for association rules, incremental mining has not gained much attention till date. As a consequence we have concentrated on incremental mining algorithms. After developing DELTA, we studied its performance against already existing algorithms APRIORI and FUP.

2.2 Problem Description

The model of the database considered in our work is the same as that presented in [AIS93a]. The same model is used in [AS94], [PCY95], [SON95], [CHN+96], etc. The model is described in the following section.

2.2.1 Model

Each transaction in the database is a tuple represented as $t =$ (Transaction Id, Customer Id, $\mathcal{I}$), where $\mathcal{I} = I_1, I_2, I_3, \cdots, I_m$ is a set of binary attributes called items. $\mathcal{I}$, associated with a transaction is such that $I_k = 1$ if $item_k$ is bought in transaction t, else $I_k = 0$. Transaction Id is an integer, and is a *key attribute*. If $\mathcal{X}$ is a set of items in the database, a transaction t is said to *satisfy* $\mathcal{X}$, if for all k such that $I_k \in \mathcal{X}$, $t[k] = 1$, i.e., $I_k = 1$ in the corresponding $\mathcal{I}$. In other terms, a transaction t *satisfies* $\mathcal{X}$ if all items I_k such that $I_k \in \mathcal{X}$ is bought in transaction t.

We mine the above database to find *association rules*. An *association rule* is an implication of the form $\mathcal{X} \Longrightarrow \mathcal{Y}$, where $\mathcal{X} = I_{a_1}, I_{a_2}, \cdots, I_{a_x}$ and $\mathcal{Y} = I_{b_1}, I_{b_2}, \cdots, I_{b_y}$. Moreover, $a_j \neq b_k$ for $1 \leq j \leq x$ and $1 \leq k \leq y$. A rule $\mathcal{X} \Longrightarrow \mathcal{Y}$, is said to have a *support* factor s iff at least $s\%$ of the transactions in $\mathcal{T}$ satisfies $\mathcal{X} \cup \mathcal{Y}$. A rule $\mathcal{X} \Longrightarrow \mathcal{Y}$, is satisfied in the set of transactions $\mathcal{T}$, with the *confidence* factor c iff at least $c\%$ of the transactions in $\mathcal{T}$ that satisfy $\mathcal{X}$ will also satisfy $\mathcal{Y}$.

Note that both *support* and *confidence* are fractions such that $0 \leq support \leq 1$ and $0 \leq confidence \leq 1$. *Support* and *confidence* should not be confused. *Support* is a measure of statistical significance, whereas *confidence* is a measure of the strength of a given rule.

Only a few of the rules may be of interest, and the aim is to find out all those interesting rules. These interesting rules, might have certain constraints. There are basically two types of constraints.

1. *Syntactic Constraints* : Syntactic constraints as the name suggests, will specify the rule and will ask for the support and confidence factors.

2. *Support Constraints* : One might be interested in only those transactions, whose support is greater than a minimum $s\%$ or confidence is greater than $c\%$, or both. Such sort of constraints are referred to as support constraints. This constraint come from the fact that if the support is not enough, the rule may not be worth considering for business strategies. The same holds true for confidence as well. The rule may be so weak that it does not mean anything.

3. *Support and Syntactic Constriants* : Not all items can appear in a rule. For example, one may be interested in finding out only those rules in which item I_k appears in the antecedent of the rule or consequent of the rule. Or another person may be interested in finding out all rules which have a set $\mathcal{X}$ as the antecedent and $\mathcal{Y}$ as the consequent, along with other items. Along with syntactic constraints support constraint values like minimum support and minimum confidence for the rules are also specified.

Mining for *association rules* can now be defined as finding in the database all those rules, that satisfy the given constraints.

2.2.2 Solution Strategy

From the above description it is can be seen that the problem of mining can be divided into two subproblems.

- Generate all subsets $\mathcal{X}$ of $\mathcal{I}$, which have support greater than the minimum threshold, given by the user called *minSupport*. These subsets X are called *large* itemsets. If there are syntactic constraints, then one has to make sure that $\mathcal{X}$ has all the items needed to satisfy the syntactic constraints, along with satisfying the constraint for being *large*.

- For a given large itemset $\mathcal{X} = I_1, I_2, \cdots, I_k, k \geq 2$, generate all rules of the form $\mathcal{Y} \Longrightarrow I_j, 1 \leq j \leq k$ and $\mathcal{Y} = \mathcal{X} - I_j$. If $\mathcal{X}$ is *large*, then a subset $\mathcal{Y}$ of $\mathcal{X}$ is surely *large*, and hence its support will be known. Now the confidence of the rule $\mathcal{Y} \Longrightarrow I_j$ will be given by $\dfrac{Support\ for\ \mathcal{X}}{Support\ for\ \mathcal{Y}}$.

While giving the model description, a rule is specified to be having a form $X \Longrightarrow Y$, where X and Y are itemsets. But here the rule is of the form $X \Longrightarrow I_j$ where I_j is one of the items. But note that $X \Longrightarrow I_j$ and $X \Longrightarrow I_k$ together will specify that $X \Longrightarrow \{I_j, I_k\}$ and which is specified in the model description. Hence, for the ease of generating rules, rules with only single items on the righthand side is generated.

Supposing we have all the large itemsets given to us. For a large set of size k, only k different (k-1)-subsets will be present. Hence we need to calculate the confidence for all these k subsets to find the rules generated by this large itemset. Hence, the second subproblem is straight forward once the solution to the first subproblem is known. So the aim is to develop better and faster algorithms to do the first subproblem, that is, finding all the *large* itemsets with the given constraints.

Agrawal, Imelinski, and Swami [AIS93a] formalized the above problem and divided it into two subproblems. Researchers have concentrated on getting a more efficient solution to the first subproblem, as the second part is simple and is not very CPU intensive.

In this chapter a overview of the data mining work given by researchers has been given. As the tool *MINTO* concentrates on mining for association rules, the former model of the data base is presented. The database consists of a set of transactions and each transaction is a set of boolean values along with transaction identifiers. When mining for association rules, the rules needed might have some support and syntactic constraints. Finally, note that the problem of mining is divided into two subproblems, first to find all large itemsets, and the next to find out the rules from them.

Chapter 3

Data Mining Algorithms

This chapter describes all the algorithms implemented as a part of the data mining tool, MINTO. MINTO has five data mining algorithms to find association rules. They are

AIS [AIS93a] AIS is a multi pass data mining algorithm. It is one of the first data mining algorithms. AIS calculates the support for a very large number of itemsets and hence is not very efficient.

APRIORI [AS94] This is a k-pass algorithm, where k is the number of elements in the biggest large itemset. APRIORI is one of the first and the best algorithms to do data mining, and is given by Agrawal and Srikant. This algorithm introduced the notion of calculating all large subsets of size i in ith iteration.

APRIORITID [AS94] This algorithm is similar to APRIORI, and is a k pass algorithm finding all the large k-itemsets in kth iteration. This algorithm will perform better in some of the cases, because here the database is reduced in successive passes. The reduction in size of the database is brought about by deleting those transactions which will not contain any of the potential large $i+1$ itemsets in the ith pass.

DHP [PCY95] A *hash*-based algorithm for mining association rules. This is also a k pass algorithm. This algorithm is supposed to improve the performance by reducing the number of candidate sets in all the passes other than the first pass. In the ith pass of DHP, the database will be trimmed in both horizontal and vertical directions.

PARTITION [SON95] This is a *two pass* algorithm to do data mining. Here divide and conquer policy is used to do mining and hence, succeed in reducing the number of passes over the data base to two. But deciding partition size is a difficult task. Also, as the size of the database increases, to achieve two pass, one has to divide the database into a large number of parts. In that case, the number of candidate sets might increase, and hence, one might do a lot of unneccesary calculations, affecting the efficiency of the algorithm.

3.1 The AIS Algorithm

AIS [AIS93a] is one of the first data mining algorithms. It is a simple algorithm to understand and implement. A pseudo code for the algorithm and a detailed description of AIS are given in this section of the report.

3.1.1 Description

AIS does the mining in two phases: First, it generates all the large itemsets and then it produces all the relevant rules from the set of large itemsets.

The pseudo code for the algorithm AIS is given in figure 3.1. There are two types of itemsets encountered in this algorithm: One is the *frontier set* and the next is the *candidate set*. Frontier set in each pass consists of all the candidates generated in the pass, whose extensions are not considered in the present pass, whereas candidate set consists of all the itemsets, whose count is being calculated in the current pass. Initially frontier set contains a single element, the null set. But candidate set will not contain anything to start with.

AIS is a multiple pass algorithm. The algorithm will terminate when frontier set becomes null. The number of passes will depend on how C_f, that is extensions of f is calculated, where $f \in frontierset$. There is a trade off between the number of passes and the number of unnecessary calculations. There are different ways in which these extensions can be calculated. One obvious way is to say that all the sets X, such that

```
procedure LargeItemsets
begin
    let Large set L = φ
    let Frontier set F = { φ }
    while F ≠ φ do begin

        // make a pass over the database
        let Candidate set C = φ;
        for all database tuples t do
        for all item-sets f ∈ F do
            if t contains f then begin
                let C_f = candidate item-sets that are
                extensions of f and contained in t;
                for all item-sets c_f ∈ C_f do
            if c_f ∈ C then
                c_f.count = c_f.count + 1;
                    else begin
                        c_f.count = 0;
                        C = C + c_f;
                    end
            end

        // consolidate
        Let  F = φ;
        for all item-sets c ∈ C do begin
            if count(c)/dbsize > minsupport then
                L = L + c;
            if c should be used as a frontier in the next pass then
                F = F + c;
        end
    end
end
```

Figure 3.1: Algorithm AIS

```
procedure Extend ( X:itemset, t:tuple )
begin
    let item I_j be such that ∀I_j ∈ X, I_j ≥ I_l;
    for all items I_k in the tuple t such that I_k > I_j do begin
        output ( XI_k );
        if (XI_k) is expected to be large then
            Extend ( XI_k, t );
    end
end
```

Figure 3.2: Extend

```
procedure Reclaim Memory
begin
    –first obtain memory from the frontier set
    while enough memory has not been reclaimed do
        if there is an itemset X in the frontier set from which
        no extension has been generated then
            move X to disk;
        else
            break;
    if enough memory has been reclaimed then return;

    –now obtain memory by deleting some candidate item-sets.
    find the candidate itemset U having maximum number of items;
    discard U and all its siblings;
    let X = parent(U);
    if Z is in the frontier set then
        move Z to disk;
    else
        disable future extensions of Z in this pass
end
```

Figure 3.3: Memory Management

$f \subset X$ is an extension of *f*. In this case AIS will turn out to be a one pass algorithm. But then, all the subsets of all the itemsets in the data base is an element of candidate set in this case. The number of elements in candidate set will be very large and might not even fit in memory. This will lead to lots of unnecessary calculations, as most of these elements, will turn out to be small. Another way is to say that, an itemset X is an extension of *f* only if, $f \subset X$ and $\mid X \mid = \mid f \mid +1$. The number of passed in AIS cannot be predicted in advance as it depends on the memory available, the extend function used, etc.

The AIS algorithm implemented in the mining tool *MINTO* uses the following idea: All the one extensions X of *f* will be an extension of *f*. If X is expected to be large, then one extensions of X also will belong to *f*. An itemset X is expected to be large if the calculated expected support is greater than a given threshold called *minSupport*. How to calculate the expected support?

Let the itemset be found for the first time after c tuples in the database have already been seen in the present pass. Then $(x - c)/dbsize$ is the actual support for X in the remaining portion of the database. Expected support is given by

$$\bar{s} = f(I_1) * f(I_2) * \cdots * f(I_k) * (x - c)/dbsize$$

The quantity of memory available for the mining process will be one of the limiting factors for the algorithm. The number of elements in the candidate set cannot be very large because of this limitation. If all the elements in the candidate set will not fit in memory, then, as shown in the memory management routine in figure 3.3, some of the frontier set members are moved to the next frontier set, i.e, they are considered as elements of frontier set during the next pass. Along with moving some of the members of frontier set to the next round, all the candidate sets, which are derived as an extension of this frontier set are also discarded from the candidate set list in the current pass. Because of this memory management policy, the algorithm will terminate only if all the extensions of any member of the frontier set along with that frontier set are together able to fit in memory. This imposes the lower bound on the memory required to execute the algorithm to completion.

```
L_1 = large 1-item-sets;
for ( k=2; L_k − 1 ≠ 0; k++ ) do begin
    C_k = apriori-gen ( L_k − 1 );              // New Candidate
for all transactions t ∈ D do begin
    C_t = subset (C_k, t );                     // Candidates contained in t
        for all candidates c ∈ C_t do
            c.count++;
    end
    L_k = { c ∈ C_k | c.count ≥ minsup }
end
Answer = ⋃_k L_k;
```

Figure 3.4: Algorithm APRIORI

```
insert into C_k the result of the following query
    select p.item_1, p.item_2, ···, p.item_{k−1}, q.item_{k−1}
    from L_{k−1}p, L_{k−1}q
    where pitem_1 = q.item_1, p.item_2 = q.item_2, ···,
        p.item_{k−2} = q.item_{k−2}, p.item_{k−1} < q.item_{k−1};
    for all item-sets c ∈ C_k do
        for all (k-1) subsets s of c do
            if ( s ∉ L_{k−1} ) then
                delete c from C_k;
```

Figure 3.5: Algorithm AprioriGen

3.2 The Apriori Algorithm

The APRIORI [AS94] is a k pass algorithm. This introduced the concept of calculating all large i-itemsets in the ith pass of the database.

The number of candidate sets is very much small, when compared to the algorithm AIS. Moreover, in the algorithm APRIORI, the number of passes is equal to the size of the largest large itemset, whereas in AIS the number of passes depends on how the extensions to the frontier set members are calculated. The efficiency of APRIORI over AIS is due to the reduction in the number of candidate sets considered.

A pseudo code for the algorithm APRIORI and a brief description are given along with some comments about the algorithm.

3.2.1 Description

The first pass of the algorithm counts item occurrences to determine large 1-itemsets. In all the subsequent passes, say kth pass, $k > 1$, all large k itemsets, i.e., all itemsets with k elements, are found. First, all k-candidate itemsets are found using large k-1 itemsets, found in the previous pass. Then the database is scanned once and the supports for these candidate k-itemsets are calculated. Once the supports are found, one can find out the large k-itemsets. Hence every iteration, except the first, has two steps: the first to find out the candidate k-itemsets and the next to find out large k-itemsets, as shown in the Figure 3.4.

AprioriGen, shown in Figure 3.5 is a routine, which is used to find all k-candidate itemsets from k-1 large itemsets. We know that any k-itemset $\mathcal{I}$ can be large, iff all k-1 sets X such that $X \subset \mathcal{I}$ are large. Hence, only 1-extensions of the large k-1 itemsets can become large k-itemsets. That too, not all 1-extensions will become large. Only those 1-extensions of the large k-1 itemsets, all of whose k-1 subsets are large can become large. The algorithm AprioriGen will select all such k-itemsets, which are potential large k-itemsets.

Once all the potential large k-itemsets (candidate k-itemsets) are found, the database is scanned and counts for each of these candidate itemsets are collected. It is trivial to find out the large k-itemsets, if the counts are known. An itemset is large if the corresponding count is more than the product of *minSupport* and the size of the database.

3.2.2 Data Structures Used

The major task involved in the APRIORI algorithm is to find out all the candidate itemsets present in the given tuple and increment the corresponding count. Another task is to find out the candidate sets themselves. In order to effectively implement both these functionalities, APRIORI uses a data structure, called the *hash tree*.

Hash tree is a tree in which each internal node is a hash node and the leaf nodes will have itemsets. The root node is said to be at level 0. Nodes at level k will point to nodes at level k+1, where $k \geq 0$. At level k, the kth element of the itemset is hashed to get the

```
L_1 = large 1-item-sets;
C̄_1 = database D;
for ( k = 2; L_{k-1} ≠ φ; k++ ) do begin
    C_k = apriori-gen ( L_{k-1} );                    // New Candidates
    C̄_k = φ;
    for all entries t ∈ C̄_{k-1} do begin
        // determine candidate item-sets in C_k contained
        // in the transaction with identifier t.TID
        C_t = { c ∈ C_k — ( c - c[k] ) ∈ t.set-of-item-sets ∧
                     ( c - c[k-1] ) ∈ t.set-of-item-sets };
        for all candidates c ∈ C_t do
            c.count++;
        if ( C_t ≠ φ ) then C̄_k += < t.TID, C_t >;
    end
    L_k = { c ∈ C_k | c.count ≥ minsup }
end
Answer = ⋃_k L_k;
```

Figure 3.6: Algorithm APRIORITID

search node at level k+1.

The subset operation is calculated as follows: Suppose $X = x_0, x_1, \cdots, x_n$ is the tuple and T is the root of the hash tree. Then at level k search for all the paths lead by $hashFunct(x_k), hashFunct(x_k + 1), \cdots, hashFunct(x_n)$. If on the way a leaf node is encountered, compare all the itemsets present in the leaf node with X and if it is a subset of X increment the count by one. In this way, we can avoid calculating all the k-subsets of X and then finding out whether they are present.

3.3 The Aprioritid Algorithm

APRIORITID [AS94] is similar to the APRIORI algorithm, in a way that APRIORITID is also a k pass algorithm, where k is the size of the largest large itemset. It improves the performance by reducing the size of the database for later passes. It achieves this by deleting those transactions from the database in their ith pass, which do not contain any of the potential large (i+1) large itemsets.

3.3.1 Description

APRIORITID is a k pass algorithm, where k is the size of the biggest large itemset. Here the first pass is limited to finding out the counts for all items in the database, that is to obtain all the large 1-itemsets. In the kth pass, all large k-itemsets are found. In the kth pass, candidate sets are found by using AprioriGen as given in Figure 3.5. Then, the count for all the itemsets in candidate set is found by scanning the database.

The size of the database will be reduced in each successive scan of the database. In the kth pass of the algorithms all those transactions, which will not have any of potential large (k+*1*) large itemsets can be removed from the database. But since, one can not remove transactions from a database to do mining, temporary databases are produced in each scan of the database. This will increase the I/O activity considerably as one has not only to read the transactions in the database, but also to write the intermediate database into the disk.

3.3.2 Data Structures Used

Each candidate itemset is assigned a unique number called its ID. Each set of candidate itemsets C_k is kept in an array indexed by the IDs of the itemsets in C_k. So, a member of $\bar{C}_k$ is of the form $< TID, \{ID\} >$. Each $\bar{C}_k$ is stored in a sequential structure.

There are two additional fields maintained for each candidate itemset. They are

1. *generators* : This field of itemset c_k store the IDs of the two large (k-*1*) itemsets whose join generated c_k.

2. *extensions* : This stores IDs of all the itemsets c_{k+1} obtained as an extension of c_k.

Now, t.set_of_itemsets of $\bar{C}_{k-1}$ gives the IDs of all the (k-*1*)-candidates contained in transaction t.TID. For each such candidate $\bar{c}_{k-1}$ the extensions field gives T_k the set of IDs of all the candidate k-itemsets that are extensions of c_{k-1}. For c_k in T_k, the generators field gives the IDs of the two itemsets that generated c_k. If these itemsets are present in the entry for t.set_of_itemsets, c_k is present in transaction t.TID. Hence add c_k to C_t.

3.4 The DHP Algorithm

Direct Hashing with Efficient pruning for Fast Data Mining (DHP) [PCY95] is a *hash*-based algorithm for mining association rules. Here, the size of the database is reduced in the successive scans of the database. Unlike in the case of APRIORITID, in DHP transactions are deleted, as well as some members of the transactions are removed from the database for later passes. Hence, the deletion is both vertical and horizontal. Also, the hashing will help in reducing the number of candidate sets.

A pseudo code for the algorithm and a detailed explanation of the DHP algorithm are given in the following sections.

3.4.1 Description

Part 1 of the algorithm finds out the large 1-itemsets and also sets the hash-table H_2 for the second pass. Part 2 generates the set of large k-itemsets for k-passes based on hash-table H_k and generates hash-table H_{k+1}. It also reduces the size of the database for the next pass as explained in Section 3.4.2. If the reduction in number of candidate sets in the next pass due to the use of hash table turns out to be small, the DHP algorithm effectively reduces to APRIORI, the only difference being the reduction in the size of database. The algorithm DHP reduces to algorithm APRIORI in the third part of the algorithm.

Part 2, like Apriori given in Section 3.2, consists of two phases: In the first phase, candidate sets are found and in the next phase, counts for these candidate sets and hence large itemsets are calculated. In the second phase along with getting the counts for the candidate sets, the hash table for the next pass will be constructed by the algorithm. Part 3 is exactly similar to part 2, except for the fact that it neither uses nor constructs the hash tables. The algorithm will move on to the third part, when creating the hash table is not productive, that is, it will not reduce the number of candidate sets by a significant amount.

```
                              /* Part 1 */

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

                              /* Part 2 */

k = 2;
D_k = D;                                    /* database for large k-itemsets */
while ( | {x | H_k[x] ≥ s} |≥ LARGE ) {
    /* make a hash table */
    gen_candidate ( L_{k-1}, H_k, C_k );
    set all the buckets of H_{k+1} to zero;
    D_{k+1} = φ ;
    for all transactions t ∈ D_k do begin
        count_support ( t, C_k, k, t̂ );                              /* t̂ ⊆ t */
        if ( | t̂ |> k ) then do begin
            make_hasht ( t̂, H_k, k, H_{k+1}, ẗ );
            if ( | ẗ |> k ) then D_{k+1} = D_{k+1} ∪ {ẗ};
        end
    end
    L_k = {c ∈ C_k | c.count ≥ s} ;
    k++;

}

                              /* Part 3 */

gen_candidate ( L_{k-1}, H_k, C_k );
while ( | C_k |> 0 ) {
    D_{k+1} = φ;
    for all transactions t ∈ D_k do begin
        count_support ( t, C_k, k, t̂ ); /* t̂ ⊆ t */
        if ( | t̂ |> k ) then D_{k+1} = D_{k+1} ∪ {t} ;
    end
    L_k = {c ∈ C_k | c.count ≥ s} ;
    if ( | D_{k+1} |= 0 ) then break;
    C_{k+1} = apriori_gen ( L_k );                       /* refer to figure 3.5*/
    k++;
}
```

Figure 3.7: Algorithm DHP

```
Procedure gen_candidate ( L_{k-1}, H_k, C_k )
    C_k = φ;
    for all c = c_p[1] ··· c_p[k-2] · c_p[k-1] · c_q[k-1],
    c_p, c_q ∈| c_p ∩ c_q |= k-2 do
        if ( H_k[h_k(c)] ≥ s ) then
            C_k = C_k ∪ {c};            /* insert c into a hash tree */
end Procedure
```

Figure 3.8: Itemset Generation

```
Procedure count_support ( t, C_k, k, t̂ )
    /* explained later */
    for all c such that c ∈ C_k and c (= t_{i_1} ··· t_{i_k} ) ∈ t do
    begin
        c.count++;
        for ( j = 1; j ≤ k; j++ ) a[i_j]++;
    end
    for ( i = 0, j = 0; i <| t |; i++ )
        if ( a[i] ≥ k ) then do begin t̂_j = t_i; j++; end
end Procedure
```

Figure 3.9: Counting Support

```
Procedure make_hasht ( t̂, H_k, k, H_{k+1}, ẗ )
    for all (k+1)-subsets x (= t̂_{i_1} ··· t̂_{i_k} ) of t̂ do
        if ( for all k-subsets y of x, H_k[h_k(y)] ≥ s ) then do
            begin
                H_{k+1}[h_{k+1}(x)]++;
                for ( j = 1; j ≤ k+1; j++ ) a[i_j]++;
            end
    for ( i = 0, j = 0; i <| t̂ |; i++ )
        if ( a[i] > 0 ) then do begin
            ẗ_j = t̂_i; j++;
        end
end Procedure
```

Figure 3.10: Hash Table Construction

3.4.2 Reducing the size of the Transaction Database

DHP reduces the size of the database progressively by trimming each individual transaction size and pruning the number of transactions in the database. Now to see how this reduction in size is brought about. One knows that any subset of a large itemset must be a large itemset by itself. Hence, a transaction will be used to determine the set of large (k+1)-itemsets only if it consists of (k+1) large k-itemsets in the previous pass. Hence, some of the transactions, which does not have k+1 k-itemsets can be pruned. If a transaction contains some large (k+1)-itemsets, any item contained in these (k+1)-itemsets will appear at least in k of the candidate k-itemsets in C_k. Hence, an item in transaction t can be trimmed if it does not appear in at least k of the candidate k-itemsets in t. This concept is used in procedure *count_support* to trim a transaction t.

3.5 The Partition Algorithm

Unlike the previous algorithms, which are k pass, the PARTITION algorithm [SON95] is a *two* pass algorithm. It uses the divide and conqure policy. The entire database is divided into a number of parts. The large itemsets in each of these parts is calculated. This is done sequentially. But as each of this process is completely independent of the others, this can be done parallely as well. Once the local large element set is obtained, the counts and hence the support in the entire database can be calculated by scanning the database once, and getting the counts for all such local large itemsets.

3.5.1 Description

The database is logically partitioned into n partitions. In the first phase of the algorithm, local large itemsets are found for all the n partitions. This is done sequentially, in the sense that local large itemsets are found in these partitions one after another. But it can be done parallely as well, as the calculation in each part is independent of the other parts. Then the global candidate set is found as a union of all local large itemsets. Once, the global candidate set is available, the counts for the itemsets is calculated in the second

P = partition_database (D)
n = Number of partitions
for i = 1 to n **begin** // Phase I
    read_in_partition ($p_i \in P$)
    L^i = gen_large_itemsets (p_i)
end

for (i = 2; $L_i^j \neq \phi, j = 1, 2, \cdots, n$; i++) **do**
    $C_i^G = \bigcup_{j=1,2,\cdots,n} L_i^j$ // Merge Phase

for i = 1 to n **begin** // Phase II
    read_in_partition ($p_i \in P$)
    for all candidates $c \in C^G$
        gen_count (c, p_i)
end

$L^G = \{c \in C^G \mid c.count \geq minSup\}$

Figure 3.11: Algorithm Partition

Procedure gen_large_itemsets (p: database partition)
    L_1^p = { large 1-itemsets along with their tid lists }
    for (k = 2; $L_k^p \neq \phi$; k++) **do begin**
        for all itemsets $l_1 \in L_{k-1}^p$ **do begin**
            for all itemsets $l_2 \in L_{k-1}^p$ **do begin**
                if $l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \cdots \wedge$
                $l_1[k-2] = l_2[k-2] \wedge l_1[k-1] < l_2[k-1]$ **then**
                    $c = l_1[1] \cdot l_1[2] \cdots l_1[k-1] \cdot l_2[k-1]$
                if c cannot be pruned **then**
                    c.tidlist = $l_1.tidlist \cap l_2.tidlist$
                if $\frac{|c.tidlist|}{|p|} \geq minSup$ **then**
                    $L_k^p = L_k^p \cup \{c\}$
            end
        end
    end
    return $\bigcup_k L_k^p$

Figure 3.12: Large Itemset Generation

prune (c: k-itemset)
for all (k-1)-subsets s of c **do**
    if $s \notin L_{k-1}$ **then**
return "c can be pruned"

Figure 3.13: Local Large Itemset Generation

```
for all 1-itemsets do
    generate the tidlist
    for ( k = 2; C_k^G ≠ φ; k++ ) do begin
        for all k-itemset c ∈ C_k^G do begin
            templist = c[1].tidlist ∩ c[2].tidlist ∩ ··· ∩ c[k].tidlist
                c.count = c.count + | templist |
        end
    end
end
```

Figure 3.14: Final Count Generation

phase of the algorithm.

Local large itemsets are found out using the procedure given in Figure 3.12. First, the large 1-itemsets in the current partition is calculated. Each large 1-itemset has a tidlist associated with it. Tidlist is nothing but the tid's of the transactions, which contain the given itemset. Now, to find out the count of any itemset, it is enough to find the number of elements in the intersection of the tidlists of the elements in the itemset. Hence, as soon as the candidate itemset is found, its support can be calculated. Hence, only one scan of the database is enough to find out the local large itemsets.

An itemset will be globally large, only if it is large in at least one of the partitions. Hence, the global candidate set is calculated as a union of local large sets. The second scan of the database is needed to find out the global support. The count is calculated using the algorithm given in Figure 3.14.

The PARTITION algorithm being a two pass algorithm, as against the multi pass algorithms explained before is of interest. A practical difficulty however, is to select the proper partition size. If there are too many partitions, then there might be more number of candidate sets and hence, the final pass will become costly. On the other hand, if the number of parts is very less, number of elements in each partition will be so large that all the calculations, and results may not be stored in memory and hence, the claim that only one pass is needed to calculate the local large itemsets will no longer be true. So, it becomes very important to select a proper partitions size to get the maximum performance by this method.

Chapter 4

Speedup through Sampling

As seen in the previous chapters algorithms to do data mining are multi pass algorithms, except in case of the algorithm PARTITION which is a two pass algorithm. The size of the database plays a major role in deciding the CPU time taken to do mining. Even though the algorithm PARTITION takes only two passes, as the size of the database increases, in order to make sure that all the required values remain in memory for the partitions involved, the database has to be divided into a large number of parts and hence, the number of candidate sets might increase, and the CPU time to do mining will increase considerably with the increase in size of the database to be mined. Moreover, to get useful information, we have to consider large databases.

As the size of the database plays a major role in deciding the CPU time needed to do mining, any way to reduce the size of the database without much change in the results of mining will be of importance. Sampling is one method of achieving this goal. Sampling is used for database query processing as well, for the same reason that it helps in finding out the result in much lesser time. Hence, sampling the database to get a sample of the data will result in getting a small database, which reflect most of the properties of the original database. Several research groups have been trying to combine sampling and data mining to speed up the process of data mining. Some examples of doing mining using sampling are [JL95] [PYC96] [Toi96] etc.

There are two kinds of sampling algorithms: The first set gives importance to correctness and completeness. These algorithms will find out the a set of potential large itemsets by using a sample of the database. Then the whole of the database is scanned to find the actual support count for all the candidate elements found in the previous step. In this pass of the database, these algorithms will check for completeness, and if the result is not complete a second pass of the database might be necessary.

The second set of algorithms are ready to trade off correctness and completeness for the sake of speed of mining. These algorithms will yield the results within certain maximum probability of losing out some information. These algorithms will choose a sample and will work with the sample. They will not scan the whole database. They will calculate the support and confidence using the sample database itself. Hence the rules obtained might not be correct or complete. There might be some additional rules and some rules might be left out, if mining is done using this technique.

MINTO has one of the sampling algorithms [Toi96] to do mining. This algorithm belongs to the first variety of the algorithms mentioned. That is, this algorithm will give the correct and complete result after mining.

4.1 Algorithm to do Mining using Sampling

The sampling algorithm to do data mining implemented in MINTO is due to H. Toivonen. It is explained in [Toi96]. It is a one pass algorithm to do data mining. Initially, a uniform random sample is choosen and mined. Once the large itemsets are found in this sample, they are taken as members of the candidate set and the correct support for these candidate sets are calculated by a complete scan of the database.

4.1.1 Description

This algorithm uses only a random sample of the relation to find approximate regularities. Samples small enough to handle in main memory can most of the times might give reasonably accurate results. This is shown by the performance analysis done in [Toi96]. But

```
draw a random sample s of size ss from r;
P = 0;
low_fr = min_fr;
//Find frequent sets in the sample
C_1 = {{A} | A ∈ R};
i = 1;
while C_i ≠ φ do begin
      for all X ∈ C_i do begin
            if fr(X,s) < low_fr then do
                  p = Pr[X is frequent];
                  if p/(Δ−P) > γ then
                        low_fr = fr(X,s);
                  else P = 1 - ( 1 - P ) ( 1 - p );
            end;
            if fr(X,s) ≥ low_fr then
                  S_i = S_i ∪ {X};
      end;
      i = i + 1;
      C_i = Compute Candidates (S_{i−1});
end;
//database pass;
compute F = {X ∈ ⋃_{j<i} C_j | fr(X,r) ≥ min_fr};
for all X ∈ F do output X and fr(X,r);
report if there is a possibility of failure;
```

Figure 4.1: Algorithm SAMPLING

if one relies on results from sampling alone, there is a risk of loosing valid association rules because of the fact that the frequency of the subset in the sample is below the threshold. This algorithm involves two phases: In the first phase an approximate result is found using a random sample and in the second phase the accurate values are calculated using the result from the first phase. This approach requires only one full scan of the database, and two passes in the worst case. The worst case happens, when a failure is reported. In this case the sample has not reflected the true behavior of the database, and hence, was not able to capture all the large itemsets.

The algorithm to do mining using this technique is given in Figure 4.1. Let s be a sample of the database. Let $\mathcal{F}(s, min\_fr)$ be the set of all large itemsets with support greater than min_fr in s. This can be got by mining the sample s with a minimum support factor min_fr, which is less than the minSupport value given by the user to do mining. The *negative border* of a set of itemsets $\mathcal{S}$ is the set of all those sets which belong to $\mathcal{S}$ together with those sets $\mathcal{X}$, all of whose subsets are present in $\mathcal{S}$, but $\mathcal{X}$ is not present in $\mathcal{S}$. After the first step of finding $\mathcal{S}$ using the sample s, one has to scan the whole database and find the supports for the itemsets present in $\mathcal{S} \cup \mathcal{B}d^-(S)$, where $\mathcal{B}d^-(S)$ is the negative border of S. All those algorithms, in which all large k itemsets are found in the kth pass of the algorithm are called *level wise* methods. The algorithms APRIORI, APRIORITID, DHP are examples of level wise algorithms. One common point in all these algorithms is that, candidate elements are found for all the passes, except the first pass using the AprioriGen method given in Figure 3.5. In AprioriGen all those k itemsets, whose *k-1* subsets are found large in the previous iteration becomes the candidate set for the kth iteration. Hence, according to the definition of negative border of a set, $\mathcal{S} \cup \mathcal{B}d^-(S)$ consists of all sets that were candidates of level-wise method in the sample.

Now, once the counts for all elements in the $\mathcal{S} \cup \mathcal{B}d^-(S)$ set is found, large itemsets can be calculated using these elements. But still one cannot say, whether all large itemsets are found by this method. Some of the large itemsets might have missed because of sampling. If a large itemset has been missed, failure is said to occur. There has been a *failure* in the sampling if all frequent sets are not found in one pass, i.e., if there is a frequent set X in

$\mathcal{F}(s, min\_fr)$ that is not in $\mathcal{S} \cup \mathcal{B}d^-(S)$. A *miss* is a frequent set Y in $\mathcal{F}(s, min\_fr)$ that is in $\mathcal{B}d^-(S)$. If there are no misses the algorithm is guaranteed to have found all frequent sets. Misses are evaluated in the whole relation, and thus are not actually missed by the algorithm. Misses indicate a potential failure. If there is a miss then a second scan of the database is needed to calculate the support of the missed potentially large itemsets.

Chapter 5

Incremental Mining

As mentioned in Chapter 4, the time taken to do data mining depends heavily on the size of the database, due to the multi pass algorithms involved. Moreover, the rules discovered are valid only in statistical terms and rules discovered from a database only reflect the state of the database at the time at which rules are discovered. To have a full picture as to how the rules behave, a large amount of data need to be collected over a substantial amount of time. Hence, to efficiently maintain the data mining operation two factors become important.

- Efficient algorithms to do data mining.
- Efficient algorithm to update, maintain and manage results.

Latter problem is dealt with using incremental mining techniques.

5.1 Description of the problem

Incremental mining is done, when a database is already mined and then some transactions are added to the same database, and the rules at this point of time need to be calculated. Hence in effect one can consider the whole database to be divided into two parts.

- The original database *DB*.

- The extended database db.

The association rules for DB is known. For the time being let the minimum support s is the same in the association rules got for DB and the one required for $DB\cup$ db.

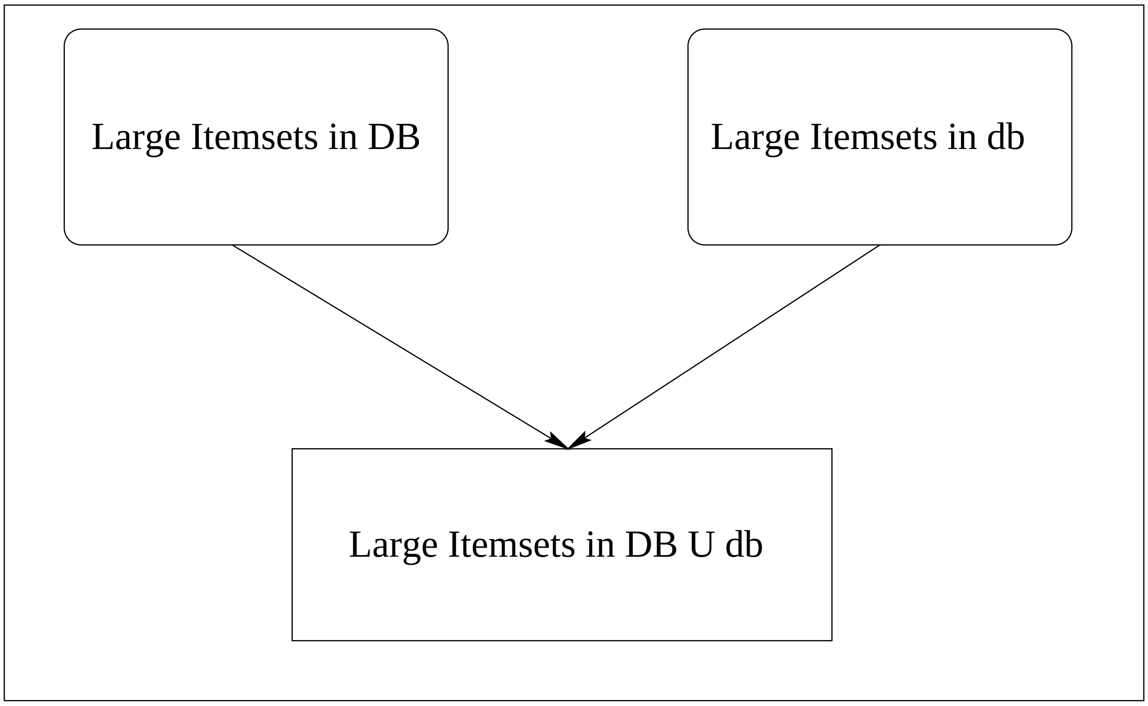

Figure 5.1: Large itemset in $DB \cup db$

The whole algorithm is based on the fact that an itemset cannot be large in $DB \cup db$ unless it is large in either DB or db. This fact is depicted in Figure 5.1. This is valid since for an itemset X to be large in DB we have to have $count_X^{DB} \geq s * DB$, and X is large in db only if $count_X^{db} \geq s * db$. Also X is large in $DB \cup db$ if $count_X^{DB\cup db} \geq s * (DB + db)$. Moreover, $count_X^{DB\cup db} = count_X^{DB} + count_X^{db}$.

The large itemsets in $DB \cup db$ will consists of three types of rules as shown in Figure 5.2.

1. The itemsets, which are large in both DB and db.

2. The itemsets, which are large in DB but not in db.

3. The itemsets, which are large in *db* but not in DB.

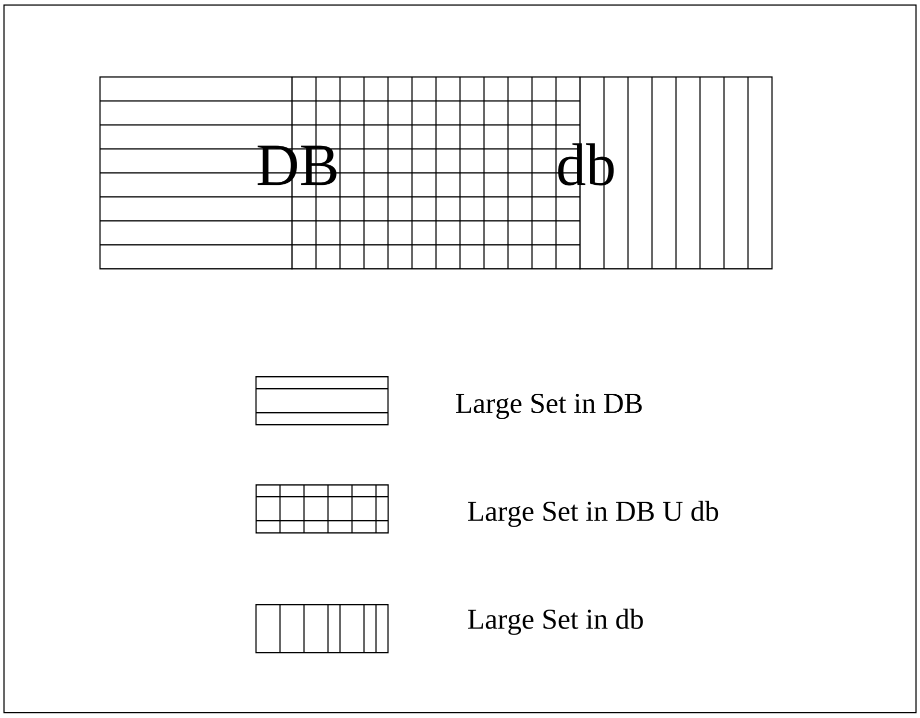

Figure 5.2: Distribution of large itemset in $DB \cup db$

The first two cases are easy to handle. They do not need a rescan of DB. In the first case, as the itemset is large in both DB and *db* $count_X^{DB}$ and $count_X^{db}$ are both known. Hence it is easy to calculate $count_X^{DB \cup db}$ as $count_X^{DB \cup db} = count_X^{DB} + count_X^{db}$.

In the second case, $count_X^{DB}$ is known. Also it is known that X is large in DB. Hence, add all such X to the candidate set while finding out large itemsets in *db* to get $count_X^{db}$. Then calculate $count_X^{DB \cup db}$ as $count_X^{DB \cup db} = count_X^{DB} + count_X^{db}$.

Only the third case causes problems. Here $count_X^{DB}$ is not known. But we know that X is large in *db*. So, it might be large in $DB \cup db$. Hence, $count_X^{DB}$ has to be calculated for all such X. And then calculate $count_X^{DB \cup db}$ as $count_X^{DB \cup db} = count_X^{DB} + count_X^{db}$.

The algorithm FUP, given in [CHN+96] is an algorithm to do incremental mining. This algorithm is explained in the next section. A new algorithm DELTA is developed to do incremental mining and is described fully in Section 5.3.

5.1.1 Notations

I The set of all items in data base

DB The size of original data base

db The size of extension to the original data base

L_k^{DB} All the large k-itemsets in DB

L_k^{db} All the large k-itemsets in db

L^{DB} All the large itemsets in DB

L^{db} All the large itemsets in db

L_k^T All the large k-itemsets in $DB \cup db$

L^T All the large itemsets in $DB \cup db$

N Set of itemsets which need rescanning the original data base DB

5.2 The FUP Algorithm

Fast Update Program (FUP) [CHN+96] is an algorithm for computing the large itemsets in the updated database. Here information from the old large itemsets is used in calculating the large itemsets for the entire updated database. They tried to solve the efficient update problem of association rules after a substantial amount of new records have been added to the database. They assume that the two thresholds, minimum support and confidence, do not change.

5.2.1 Notation

Let L be the set of large itemsets in the database DB, s be the minimum support, and DB be the number of transactions in DB. Let db of the new transactions is added to the original database DB, and db is the number of transactions in db.

1. DB: the original database (with its size, i.e., the total number of transactions, equal to DB);
2. L_k^{DB}: the set of all large k-itemsets in DB, where $k = 1, 2, \cdots, r$;
3. db: an increment database (with its size equal to db);
4. s: the minimum support threshold

Figure 5.3: Input for FUP

$L^{DB \cup db}$: The set of all large itemsets in $DB \cup db$.

Figure 5.4: Output for FUP

L_k^{DB} denotes the set of all large k-itemsets in DB and $L_k^{DB \cup db}$ is the set of all large k-itemsets in $DB \cup db$. $C_k^{DB \cup db}$ is the set of k-candidate sets in the k-th iteration of FUP. $suport_X^{DB}, support_X^{db}$, and $support_X^{DB \cup db}$ represent the support counts of an itemset X in DB, db and $DB \cup db$, respectively.

5.2.2 Description

Winners are those itemsets, which are large in $DB \cup db$. Losere are the itemsets, which are large in DB, but ceases to be large in db.

The first step, like in any other algorithm to do data mining is to find out the large 1-itemsets. Scan db, for all itemsets $X \in L_1^{DB}$, update its support count $support_X^{DB \cup db}$. If $support_X^{DB \cup db} < s * (DB + db)$, then X is a loser. Remove all such losers from L_1^{DB} and add the remaining elements in L_1^{DB} to $L_1^{DB \cup db}$. Along with elements of L_1^{DB}, counts for all candidate elements $C_1^{DB \cup db}$, calculates counts for $X \subseteq T$, which is not in L_1^{DB}. Now, if for $X \in C_1^{DB \cup db}$, and $support_X^{db} \geq s * db$, X can be large, hence scan DB to update the support count $support_X^{DB \cup db}$ for all such $X \in C_1^{DB \cup db}$. Then using the new support counts all elements in $L_1^{DB \cup db}$ can be found.

Now to see how to find $L_k^{DB \cup db}$ in the kth iteration. Some of the losers can be found even before scanning db. The set of losers $L_{k-1}^{DB} - L_{k-1}^{DB \cup db}$ have been identified in the

/* find $L_1{}'$, the set of all large 1-itemsets in $DB \cup db$ */
 $W = L_1^{DB}; C = \phi; L_1^{DB\cup db} = \phi; P = \phi;$
 /* W : winners, C : candidate sets, $L_1^{DB\cup db}$:initialized, P: for optimization */
 for all $T \in db$ **do** /* scan db */
  for all 1-itemset $X \subseteq T$ **do** {
   if $X \in W$ then $support_X^{db} + +;$
   else {
    if $X \notin C$
    then { $C = C \cup \{X\}; support_X^{db} = 0;$ }
   /* initialize the support count and add X into C */
   $support_X^{db} + +;$
  };

 for all $X \in W$ **do** /* put winners into $L_1^{DB\cup db}$ */
  if $support_X^{DB\cup db} \geq s * (DB + db)$
   then $L_1^{DB\cup db} = L_1^{DB\cup db} \cup \{X\};$

 for all $X \in C$ **do** /* prune candidate sets in C */
  if $support_X^{db} < s * db$
   then { $C = C - \{X\}; P = P \cup \{X\};$ }
   /* P will be used for optimization */

 for all $T \in DB$ **do** /* scan DB */
  for all 1-itemset $X \subseteq T$ **do** {
   if $X \in C$ **then** $support_X^{DB} + +;$
   if $X \in P$ **then** removes X from T;
   /* Transaction T is reduced */
  };

 for all $X \in C$ **do** /* put winners into $L_1 DB \cup db$ */
  if $support_X^{DB\cup db} \geq s * (DB + db)$
   then $L_1^{DB\cup db} = L_1^{DB\cup db} \cup \{X\};$

 return $L_1^{DB\cup db}$. /* end of the 1st iteration */

Figure 5.5: First iteration of FUP

/* for k = 2 or larger, repeat this program fragment to find $L_k^{DB\cup db}$, the set of all large k-itemsets in the updated database, until either $L_k^{DB\cup db}$ returned is empty or db = ϕ */

$W = L_k^{DB}$; $L_k^{DB\cup db} = \phi$;
/* W : winners, C : candidate sets, $L_k^{DB\cup db}$:initialized */
C = apriori-gen ($L_{k-1}^{DB\cup db}$) - L_k^{DB};
/* the size-k candidate sets */

for all k-itemset $X \in W$ **do**
  /* prune off losers in W */

  for all (k-1)-itemset $Y \in L_{k-1}^{DB} - L_{k-1}^{DB\cup db}$ **do**
    if $Y \subseteq X$ **then** $\{W = W - \{X\}$; break; $\}$

for all $T \in db$ **do** { /* scan db */
  for all $X \in Subset(W,T)$ **do** $suport_X^{db}++$;
    /* Subset(W,T) returns all the sets in W contained in T */

  for all $X \in Subset(C,T)$ **do** $support_X^{db}++$;
    /* find support of all $X \in C$ */
  Reduce_db (T);
  /* Some items in transactions in db can be removed */
}

for all $X \in W$ **do**
  /* put the winners from W into $L_k^{DB\cup db}$ */
  if $support_X^{DB\cup db} \geq s * (DB + db)$
    then $L_k^{DB\cup db} = L_k^{DB\cup db} \cup \{X\}$

for all $X \in C$ **do**
  /* prune candidate sets in C */
  if $support_X^{db} < s * db$
    then C = C - {X};

for all $T \in DB$ **do** { /* scan DB */
  for all $X \in Subset(C,T)$ **do** $support_X^{DB}++$;
  Reduce_Db (T); }
  /* Some items in transactions in DB can be removed */

for all $X \in C$ **do**
  /* put the winners from C into $L_k^{DB\cup db}$ */
  if $support_X^{DB\cup db} \geq s * (DB + db)$
    then $L_k^{DB\cup db} = L_k^{DB\cup db} \cup \{X\}$

return $L_k^{DB\cup db}$. /* end of the k-th iteration */

Figure 5.6: kth iteration of FUP

(k-1)th iteration, and will be removed from the set of winners even before scanning db. Scan the db to find out the support for L_k^{DB}. Just like in the first iteration, filter out the losers. The set of candidate sets, $C_k^{DB \cup db}$ is generated by applying the AprioriGen function, explained in Figure 3.5 on $L_{k-1}^{DB \cup db}$. The sets in L_k^{DB} are excluded while creating $C_k^{DB \cup db}$. Now, the count for elements in $C_k^{DB \cup db}$ is found, while scanning db. Then just like in the first iteration, after pruning $C_k^{DB \cup db}$, DB is then scanned to find the count for $X \in C_k^{DB \cup db}$.

5.2.3 Reduction in DB size

FUP uses the techniques used in DHP to reduce the size of the updated database. This technique is used to reduce the size of the database db. But a different method is used to reduce the size of DB.

In the first pass all those elements, which do not have enough support in db, are removed from the transactions in DB. At kth iteration, while FUP is counting the support for the sets in the candidate sets $C^{DB \cup db}$ and W, for each $I \in T$, the number of sets in $C^{DB \cup db}$ and W which contain I is calculated. If this number is smaller than k, then I cannot belong to any large $(k-1)$-itemset. Hence, I can be removed from the transaction in db. Reduce_db does this reduction.

Any item in DB which does not belong to any set in L_k^{DB} or $C_k^{DB \cup db}$ will not belong to any large (k+1)-itemset. Therefore, in the scanning of DB to compute the supports of sets in $C_k^{DB \cup db}$, all items that do not belong to any set in L_k^{DB} or $C_k^{DB \cup db}$ can be removed. Reduce_DB performs this reduction.

5.3 The DELTA Algorithm

Differential Evaluation of Large iTemset Algorithm (DELTA) is an algorithm to do incremental data mining proposed by us. DELTA deals with finding the association rules. It helps in finding the association rules for $DB \cup db$ using the rules for DB, obtained sometime earlier. It produces the large itemsets for db as a byproduct, unlike in FUP.

```
/* Get the large 1 itemsets */
/* collect the counts of the itemsets */
/* Set counters for all x ∈ I and initialize them to zero */
for all t ∈ db do
        for all x ∈ t do
                count_x^db ++;

for all x ∈ I do
        if count_x^db ≥ s * db then do
                L_1^db = L_1^db ∪ {x}
                if x ∈ L_1^DB ∧ count_x^db + count_x^DB ≥ s * (DB + db) then do
                        L^T = L^T ∪ {x}
                else if count_x^db ≥ s * (DB + db)
                        /* Assume that this itemset is not present in the old database at all.
                        Still this itemset is a large itemset and hence is added to the set of
                        large itemsets */
                        L^T = L^T ∪ {x}
                else do
                        N = N ∪ {x}
                        /* Add this element to the set, which has to be calculated
                        over the old DB */
        else if x ∈ L_1^DB ∧ count_x^db + count_x^DB ≥ s * (DB + db)
                L^T = L^T ∪ {x}
```

Figure 5.7: First Iteration of Algorithm DELTA

/* For k = 2 to n do the following till L^{db}_{k-1} is Nil */
k = 2
while $L^{db}_{k-1} \neq \phi$ **begin**
  $candidate_k = AprioriGen(L^{db}_{k-1})$
  $previous_k = L^{DB}_k - candidate_k$
  /* Set the counters for all $X \in candidate_k \cup previous_k$ and initialize them to zero */
  for all $t \in db$ **do**
    for all $X \subseteq t \wedge X \in candidate_k$ **do**
      $count^{db}_X + +;$
    for all $X \subseteq t \wedge X \in previous_k$ **do**
      $count^{db}_X + +;$
    Reduce_db (t);
  if $X \in candidate_k \wedge count^{db}_X \geq s * db$ **then do**
    $L^{db}_k = L^{db}_k \cup \{X\}$
    if $X \in previous_k \wedge count^{db}_X + count^{DB}_X \geq s * (DB + db)$ **then do**
      $L^T = L^T \cup \{X\}$
    else if $count^{db}_X \geq s * (DB + db)$
      /* Assume that this itemset is not present in the old database at all. Still this itemset is a large itemset and hence is added to the set of large itemsets */
      $L^T = L^T \cup \{X\}$
    else if $count^{db}_X \geq s * db$ **do**
      if there exists $Y \subset X \wedge Z \subset X$ such that $\mid Y \mid = \mid Z \mid = \mid X \mid -1 \wedge$
      $Y \neq Z$ 3cm **and** $Y \in L^{DB}_{k-1} \wedge Z \in L^{DB}_{k-1}$ **and**
      $count^{DB}_y + count^{DB}_Z - 2 * s * DB + count^{db}_X < s * (DB + db)$ **then**
        discard the itemset X as it cannot be large
                                        // Explained in 5.3.2
      else
        $N = N \cup \{X\}$
        /* Add this element to the set, which has to be calculated over the old DB */
    if $X \in previous_k \wedge count^{db}_X + count^{DB}_X \geq s * (DB + db)$
      $L^T = L^T \cup \{X\}$
  k = k+1
end

Figure 5.8: k-th Iteration of Algorithm DELTA

```
/* for all the elements in the new set find the count in DB */
if N ≠ φ then
    /* Set the counters for X ∈ N and initialize them to zero */
    for all t ∈ DB
        for all X ⊆ t ∧ X ∈ N
            count_X^DB ++;

for all X ∈ N
    if (count_X^db + count_X^DB ≥ s * (db + DB) ) then
        L^T = L^T ∪ {X}
```

Figure 5.9: Rescan of DB for Algorithm DELTA

5.3.1 Description

Now, in the first iteration, all the counts for 1-itemsets are found for db. If X is such that it belongs to the first category, then X is large and is added to $L^{DB\cup db}$. If X is such that X is large in DB, but not in db, then X is added to $L^{DB\cup db}$ if $count_X^{DB}+count_X^{db} \geq s*(DB+db)$. Now, if X is large in db, but not in DB add it to a set, which has to be analyzed later.

In k-th iteration, all k-large itemsets of the first two kinds will be found. From L_{k-1}^{db} generate the candidate sets for the k-th iteration by AprioriGen method explained in Figure 3.5. Add all those X such that $X \in L_k^{DB}$ and $X \notin candidateset$ to the candidate set. Now, scan db to find $count_X^{db}$ for all candidate sets. For all $X \in candidateset$ if $X \in L_k^{DB}, count_X^{DB\cup db} = count_X^{DB}+count_X^{db}$. Now, if $X \in candidateset$ and $X \notin L_k^{DB}$ add X to the set, which has to be analyzed later. While finding out the support for the candidate set, the size of the tuple can be reduced by using the same technique used in FUP, and explained in Section 5.2.3

After all large itemset of the first two categories are found, one has to find out all the large itemsets, which belong to the third category. All the potential large itemsets of the third category are stored during the iterations described previously. Scan DB to find $count_X^{DB}$ for all X in this set. $count_X^{DB\cup db} = count_X^{DB} + count_X^{db}$. If $count_X^{DB\cup db} \geq s*(DB+db)$ add X to $L^{DB\cup db}$. This ensures that we have all the large itemsets of db and $DB \cup db$.

5.3.2 Why itemsets need not be considered for $DB \cup db$

Lemma:

If there exists $Y \subset X \wedge Z \subset X$ such that $\mid Y \mid = \mid Z \mid = \mid X \mid -1 \wedge Y \neq Z$ and $Y \in L_{k-1}^{DB} \wedge Z \in L_{k-1}^{DB}$ and $count_Y^{DB} + count_Z^{DB} - 2 * s * DB + count_X^{db} < s * (DB + db)$ then itemset X cannot become large in $DB \cup db$.

Let $X = A \cup \{b\} \cup \{c\}$

and $Y = A \cup \{b\}$

and $Z = A \cup \{c\}$

Then we know that at least $count_A^{DB} - count_X^{DB}$ number of transactions will have A but not b. Similarly $count_A^{DB} - count_Y^{DB}$ number of transactions will have A but not c. Hence, we have $count_A^{DB} - count_X^{DB} + count_A^{DB} - count_Y^{DB}$ number of transactions having A, and which either do not have b or do not have c. Hence, this gives an upper bound for the number of transactions that can have A, b, and c, i.e., the number of transactions having $A \cup \{b\} \cup \{c\}$. In otherwords, the support for X has an upper bound given by $count_A^{DB} - (2count_A^{DB} - count_X^{DB} - count_Y^{DB}) = count_X^{DB} + count_Y^{DB} - count_A^{DB}$. If this upperbound is less than the support needed, X cannot become a large itemset in $DB \cup db$.

5.4 Improvement in DELTA

Till now, we have considered DELTA, where the *support* threshold is the same for DB and *db*. But we can change the algorithm slightly and make it work even when the support is changed. The algorithm FUP might also work in this case, but it is not dealt with at present.

First, to see what happens when $minSupport_{DB} < minSupport_{db}$. In this case, we do not have to rescan the database DB, as we will have all the rules in DB with support $> minSupport_{DB}$. Only thing to note here is, while doing incremental mining, take care to select only those elements in the large set of DB which have support greater than the minSupport specified now, and put it in $previous_k$. Also, for some of the X such that $X \in N$, N may be present in the large set of DB and hence $countX^{DB}$

will be known and so rescan to find out the count for X is not necessary. This will reduce the number of elements in N for rescanning considerably. Supposing the set X is such that $support_X^{db} + minSupport^{DB} < minSupport_X^{DB \cup db}$, then X cannot become large in $DB \cup db$ and hence can be removed from N before the scan of DB. So, note that $db + minSupport^{DB} < minSupport_X^{DB \cup db}$, then DB need not be scanned again and hence, incremental minining becomes incremental in the true sense, i.e., we can discard the old database DB.

We can extend this idea and store the large itemsets in the way of a bar graph as shown in Figure 5.10. The numbers put inside the boxes represent the range of support for the itemsets which belong to that box and the height of the box indicates the number of itemsets in the box. Then this data can be used for incremental mining, when additional tuples are added to the database.

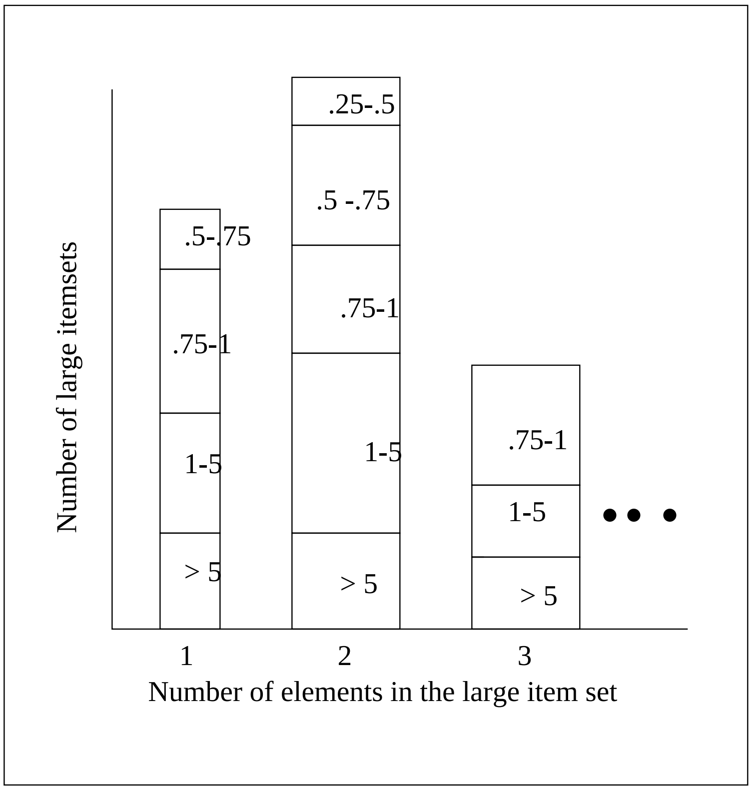

Figure 5.10: Graph to store large itemsets

On the other hand, if $minSupport_{DB} > minSupport_{db}$ incremental mining cannot be done, and the mining of whole database has to be done to get the rules.

Chapter 6

Performance Evaluation of DELTA

Experiments are conducted in order to assess the performance of DELTA (5.3), our new incremental mining algorithm. Experiments are conducted to compare its performance with that of Apriori (3.2) and FUP (5.2). Apriori was chosen, as it serves as a baseline for most of the other data mining algorithm studies done before this. FUP is the only other incremental mining algorithm that we are aware of in the literature. Hence, we study the performance of Delta against FUP.

6.1 Generation of Synthetic Data

The databases used for the experiments are synthetic data generated using the technique given in [AS94]. Here, the transactions attempt to mimic the transactions in the retailing environment. The model assumes that people tend to buy sets of items together. An example of such a set may be a dress, jacket, a pair of shoes and ornaments. But, some people may buy only some of the items from such a set. Each such set is potentially a maximal large itemset. A transaction may contain more than one large itemset. Transaction sizes are typically clustered around a mean, $\mid T \mid$. Typical sizes of large itemsets are also clustered around a mean, $\mid I \mid$.

There are different parameters which are of interest while creating a database using the simulator. Some of these parameters considered in the simulator are shown in table

N	Number of items
$\mid I \mid$	Mean size of the potentially large itemsets.
$\mid L \mid$	Number of potentially large itemsets.
$\mid T \mid$	Mean size of the transaction
D	Number of transactions in database DB.
d	Number of transactions in increment db.

Table 6.1: Parameter Table

6.1

To begin with a set of itemsets $\mathcal{T}$ is created. These are called potentially large itemsets. An itemset in $\mathcal{T}$ is generated by first picking the size of the itemset from a Poisson distribution with mean $\mid I \mid$. Items in the first itemset are chosen randomly. To ensure that some of the large itemsets have common items, some fraction of the items in subsequent itemsets are chosen from the previous itemset generated. The remaining items are picked at random. Each itemset in $\mathcal{T}$ has a weight associated with it, which corresponds to the probability of this itemset being picked. The number of large itemsets, such created is given by $\mid \mathcal{T} \mid = \mid L \mid$.

To model the phenomenon of all the items in large itemsets not always being bought together, the simulator uses some probability distribution, with which it will take only a few values from the itemset, rather than choosing the entire set. This is achieved by when adding an itemset to a transaction, drop an item from the itemset as long as a uniformly distributed random number between 0 and 1 is less than some small number c.

The specified number of transactions D is got as follows: To begin with the size of a transaction is determined using a poison distribution with mean $\mid T \mid$. If each item is chosen with the same probability p, and there are N items, the expected number of items in a transaction is given by a binomial distribution with parameters N and p, and is approximated by a Poisson distribution with mean Np.

Then, items are assigned to the transaction. Transaction is assigned a series of potentially large itemsets. If large itemset on hand does not fit in the transaction, the itemset is put in the transaction anyway in half the cases and the itemset is moved to the next

N	1000
$\mid I \mid$	4
$\mid T \mid$	10
$\mid L \mid$	1000
D	100000
d	1000, 10000, 50000, 100000

Table 6.2: Parameter Values

transaction in the rest of the cases.

For our experimental study we have set the values of the above parameters as given in table 6.2.

The database is a set of transactions. Each transaction has a transaction id, a customer id, number of items purchased in the transaction and a set of items, which are purchased in the transaction. The transaction id is a *key* attribute in the relation of the database being considered for mining. Each tuple in the database is of variable length, the length of which is given by the number of items purchased in the transaction. Since only very few items will be purchased in a transaction, it is advantageous to store the list of items purchased rather than representing the transaction using a bit vector. In the bit vector representation all the transactions are of equal size. Each transaction has a transaction id, and N bits, where kth bit is 1 if $item_k$ is bought in that transaction. If one stores the bit vector, N bits are needed to store each transaction, where N is the number of items in the database, which generally runs to thousands in case of a supermarket data. On the other hand, in the first representation, we will be using only the number of items purchased times size of an integer number of bits to represent a transaction. This reduces the size of database to a considerable extent.

The simulator used to produce the data to conduct the experiment is the one given in [AS94]. But [AS94] does not have a concept of an extension db to DB. So, we have given the number of transaction as DB + db to get the whole database in one go. Then we divide the database produced into two parts DB and db.

The above method will produce a uniformly distributed data in DB and db. But when the trend in marketing is seen, this may not be true. There may be quite a few changes between the large itemsets in DB and *db*. The simulator is modified in the following manner to get this effect: After DB transactions are produced, a certain percentage of the potentially large itemsets are changed. A potentially large itemset is changed as follows. First, it is decided whether the itemset has to be changed or not. Once, it is decided to change the itemset, each item in the itemset is changed to its mirror image with some probability. A mirror image of an item i is the item N-i. Once the large itemsets are changed, *db* number of transactions are produced with these changed large itemsets.

6.2 Experimental Results and Description

In order to show that our incremental mining algorithm DELTA performs better than the other algorithms, its performance is compared with two other algorithms, namely, APRIORI and FUP. The experiments are performed on 166MHz Ultrasparcs machines.

The entire database to be mined becomes $DB \cup db$. If we are using APRIORI, then $DB \cup db$ will be mined to get the association rules. On the other hand FUP or DELTA will use the previous results obtained by mining DB to mine $DB \cup db$. DELTA will produce the large itemsets in *db* in addition to itemsets which are large in $DB \cup db$. So the output set is richer in case of DELTA than in the case of both FUP and APRIORI.

6.2.1 Input Database

To analyze the performance, mining is done using APRIORI, FUP, and DELTA on three sets of databases.

The uniform distribution Here DB and *db* both are produced with the same uniform distribution.

Moderate Skew First DB is produced and then as explained before, the potentially large itemsets are changed with some probability and then *db* is produced.

High Skew First DB is produced and then as explained before, the potentially large itemsets are changed with probability greater than the one in the moderate skew case and then *db* is produced.

The data distribution is uniform in both DB or *db*. But the only thing that changes is when moving from DB to *db* the potentially large itemsets might change in the latter two cases and will not change in the first case.

What really makes the difference in these cases is the following. There are five types of large itemsets.

Persistent Those itemsets which are persistently large, i.e., are large in both DB and *db*, and hence are large in $DB \cup db$

Recent Itemsets which are of recent interest. These itemsets are large in $DB \cup db$ and in *db*, but are not large in DB.

History Itemsets which are of interest, because they are large in $DB \cup db$ and in DB, but are not large in *db*.

New Itemsets which are only of recent interest. These itemsets are large only in *db*.

Old Itemsets which are of historical interest. These itemsets are large only in DB and not large even in $DB \cup db$.

These values are of interest as they give an idea of the distribution of the large itemsets among the three types large itemsets in $DB \cup db$, explained in Chapter 5.

6.2.2 Experiments Done

As mentioned before mining is done on three sets of data using APRIORI, FUP, and DELTA. In each experiment the time taken by the CPU to do mining is found out varying the support levels from 6% to 0.75%. It the support value is more than 6% there are not many itemsets becoming large and the mining stops after the first iteration and hence,

ratio	support	$DB \cup db$	DB	db	Persistent	Recent	History	New	Old
100:1	6%	6	6	9	5	0	1	4	0
	4%	39	39	48	34	0	5	14	0
	2%	178	177	185	156	1	21	28	0
	1%	1238	1238	1603	986	3	249	614	3
	0.75%	1950	1953	2366	1687	0	263	679	3
100:10	6%	6	6	5	5	0	1	0	0
	4%	39	39	41	37	0	2	2	0
	2%	178	177	173	170	1	7	2	0
	1%	1227	1238	1244	1130	2	95	112	13
	0.75%	1948	1953	2015	1828	4	116	183	9
100:50	6%	6	6	5	5	0	1	0	0
	4%	39	39	38	38	0	1	0	0
	2%	176	177	176	173	1	2	2	2
	1%	1232	1238	1226	1188	9	35	29	15
	0.75%	1947	1953	1950	1894	17	36	39	23
100:100	6%	6	6	5	5	0	1	0	0
	4%	39	39	38	38	0	1	0	0
	2%	176	177	177	173	1	2	3	2
	1%	1235	1238	1236	1203	17	15	16	20
	0.75%	1945	1953	1951	1908	18	19	25	26

Table 6.3: Table of values for Uniform Distribution

ratio	support	$DB \cup db$	DB	db	Persistent	Recent	History	New	Old
100:1	6%	6	6	7	5	0	1	2	0
	4%	37	37	40	30	0	7	10	0
	2%	177	177	175	146	0	31	29	0
	1%	1236	1232	1541	1078	4	154	459	0
	0.75%	1943	1945	2205	1542	2	399	661	4
100:10	6%	6	6	5	5	0	1	0	0
	4%	36	37	32	31	0	5	1	1
	2%	177	177	172	158	0	19	14	0
	1%	1217	1232	1305	1136	8	73	161	23
	0.75%	1923	1945	1923	1722	3	198	196	25
100:50	6%	6	6	4	4	0	2	0	0
	4%	33	37	32	32	0	1	0	4
	2%	175	177	178	166	2	7	10	4
	1%	1201	1232	1277	1161	12	28	104	43
	0.75%	1892	1945	1939	1804	31	57	104	84
100:100	6%	5	6	4	4	0	1	0	1
	4%	34	37	32	31	1	2	2	4
	2%	177	177	176	165	5	7	6	5
	1%	1195	1232	1209	1161	15	19	33	52
	0.75%	1882	1945	1932	1815	44	23	73	107

Table 6.4: Table of values for High Skew

ratio	support	$DB \cup db$	DB	db	Persistent	Recent	History	New	Old
100:1	6%	6	6	7	5	0	1	2	0
	4%	38	39	44	30	0	9	14	0
	2%	177	177	181	149	1	27	31	1
	1%	1237	1236	1289	756	4	477	459	0
	0.75%	1946	1952	2269	1512	1	433	756	7
100:10	6%	6	6	7	5	0	1	2	0
	4%	38	39	32	29	0	9	3	1
	2%	176	177	172	162	1	13	9	2
	1%	1222	1235	1240	1029	6	187	205	19
	0.75%	1914	1952	1904	1638	2	274	264	40
100:50	6%	6	6	6	5	0	1	1	0
	4%	33	39	32	31	0	2	1	6
	2%	175	177	170	164	2	9	4	4
	1%	1114	1235	1193	1077	11	26	105	132
	0.75%	1860	1952	1850	1679	25	156	146	117
100:100	6%	5	6	6	5	0	1	1	0
	4%	33	39	30	28	0	5	2	6
	2%	175	177	169	164	3	8	2	5
	1%	1119	1234	1155	1078	19	22	58	134
	0.75%	1759	1952	1829	1686	26	47	117	219

Table 6.5: Table of values for Moderate Skew

the starting point of 6% is chosen. By the time mining is done for a support of 0.75% sufficient number of large itemsets are found and hence the support range is stopped at this point.

Along with varying the support counts, the experiments are conducted by varying the size of *db* from 1000 transactions to 100000 transactions. The ratio DB:db is varied from 100:1 to 100:100 and the time taken by the different algorithms to do mining is noted. The time taken is calculated using the system command "/usr/bin/time".

6.2.3 Observations and Explanation

The results of three sets of experiments are given in graphs shown in Figures 6.1, 6.2 and 6.3, respectively. The values of the five different types of large itemsets found for these data sets are given in tables 6.3, 6.4 and 6.5, respectively.

On the other hand, DELTA performs better than FUP in all the cases. In the worst case that is 100:100 case we get DELTA performing at least 1.5 times as fast as FUP and in the best case as much as thrice as fast as FUP. FUP does lots more disk accesses than DELTA, which is not depicted in these graphs, as these graphs plot the execution time or CPU time taken versus the support threshold chosen for the particular experiment.

These graphs clearly show the advantage of using DELTA over FUP and APRIORI. It can be noted that the time taken by DELTA and FUP will move towards that taken by APRIORI as the ratio DB:db goes from 100:1 to 100:100. This is because DELTA and FUP will have to perform mining on db and if needed will have to rescan DB. Since, the ratio between the size of DB and db decreases as we move from 100:1 case to 100:100 case, we spend more time in mining db, and hence, we observe the gap between the time needed by incremental mining algorithms will reduce with the ratio DB:db.

The time difference to get the large itemsets using the three algorithms, APRIORI, FUP and DELTA can be explained based on the number of comparisons made. The major work involved in all these algorithms is to count the support for the candidate sets and for that one has to find out the all subsets of a given transaction present in the candidate set and calculate its count and to do this, one has to compare the items in a given transaction with the members of candidate set and hence, number of comparisons made can explain most of the results. Hence the major work involved in any of the above algorithm is in finding out the subsets of a given transaction, present in the given candidate set and incrementing the support count of those subsets. Hence, the time taken to do mining by these three algorithms will heavily depend on the number of candidate sets. Theoretically the number of comparisons made can be calculated based on the following observations:

- In all the three algorithms (APRIORI, FUP and DELTA) candidate sets are represented using a *hash tree* data structure, explained in Section 3.2.2.
- At level i, (tlen - i) number of subtrees are going to be searched, where tlen is the average number of elements in the transaction.
- The number of leaf nodes searched will be equal to $tlen(tlen-1)\cdots(tlen-l)$ where l is the number of levels in the candidate tree.
- If we know the number of candidates and also the size of each candidate, say k, levels in the candidate tree is given as follows.

  level is a number such that $(\frac{tlen}{number^{level}})(k+2) \leq number < (\frac{tlen}{number^{level-1}})(k+1)$.
- Since, in most of the cases, only one element of the itemset will be compared, we assume only one comparison is being made in evaluating the number of comparisons.

So the number of comparisons made will be given by

$$\sum_{k} teln(teln-1)\cdots(tlen-level_k)$$

Then the time taken for the data mining will be proportional to

(number of comparisons)(number of transactions).*

In the case of APRIORI the number of comparisons is calculated using the number of candidate sets in each pass. In the case of FUP and DELTA, the number of candidate sets will be different for DB and db. Hence, the number of comparison is calculated separately for DB and db and then the two are added to get the total number of comparisons. The graphs in Figures 6.4, 6.5, and 6.6 represent the number of comparisons against the support threshold corresponding to the experiments given in Figures 6.1, 6.2, and 6.3 respectively. It can be noted from these figures that the corresponding graphs are similar.

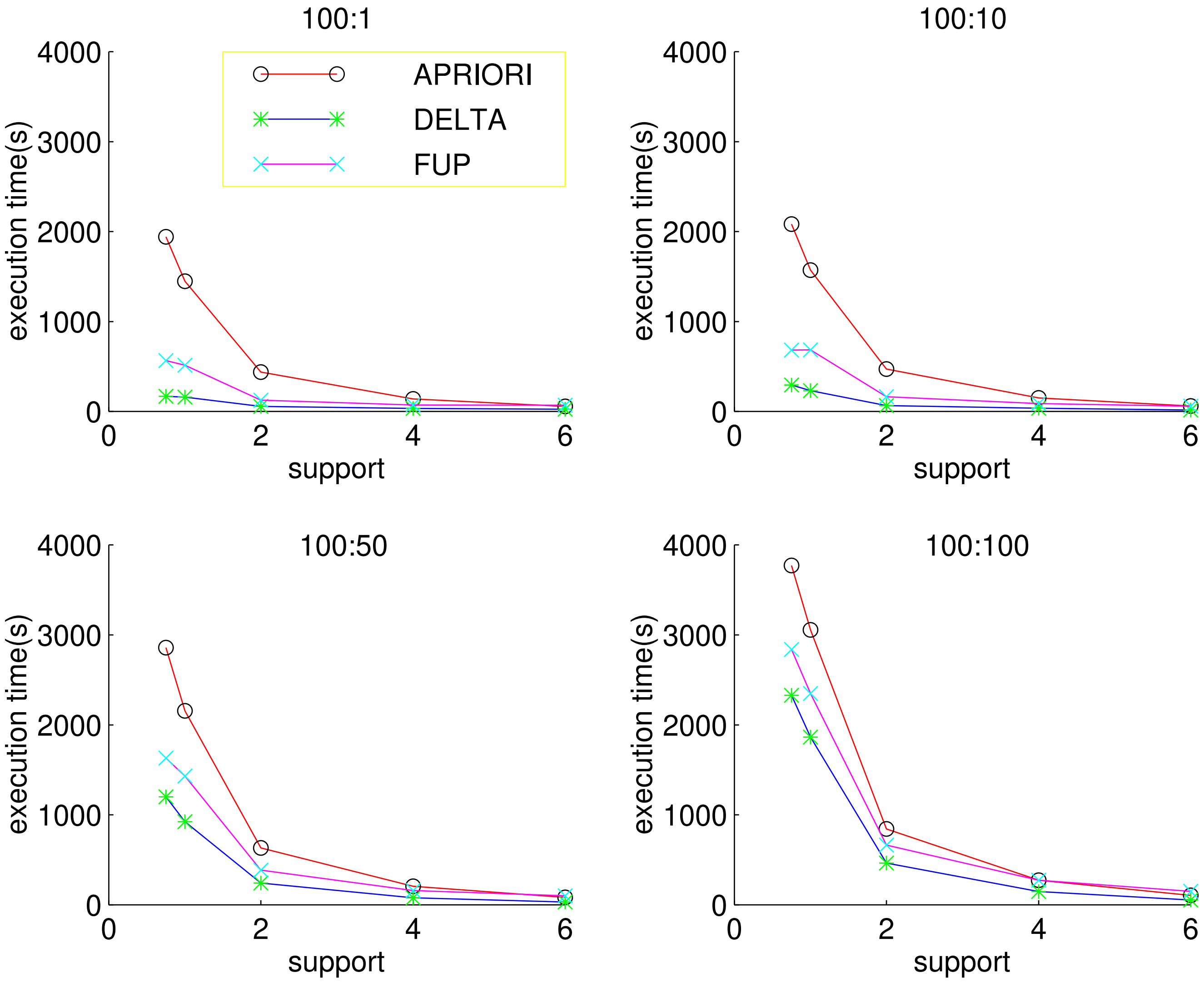

Figure 6.1: Uniform Distribution : Execution time vs support

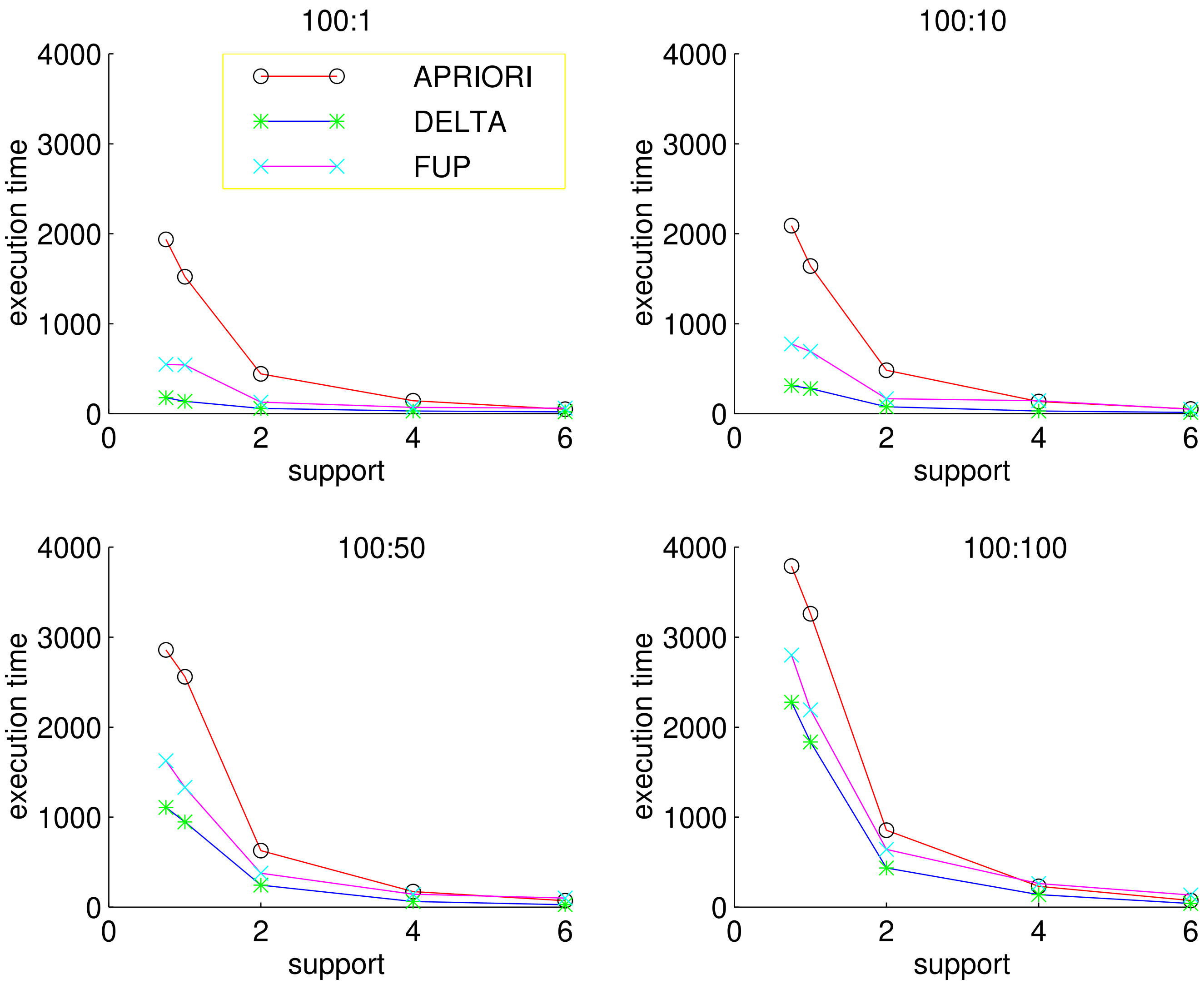

Figure 6.2: High Skew: Execution time vs support

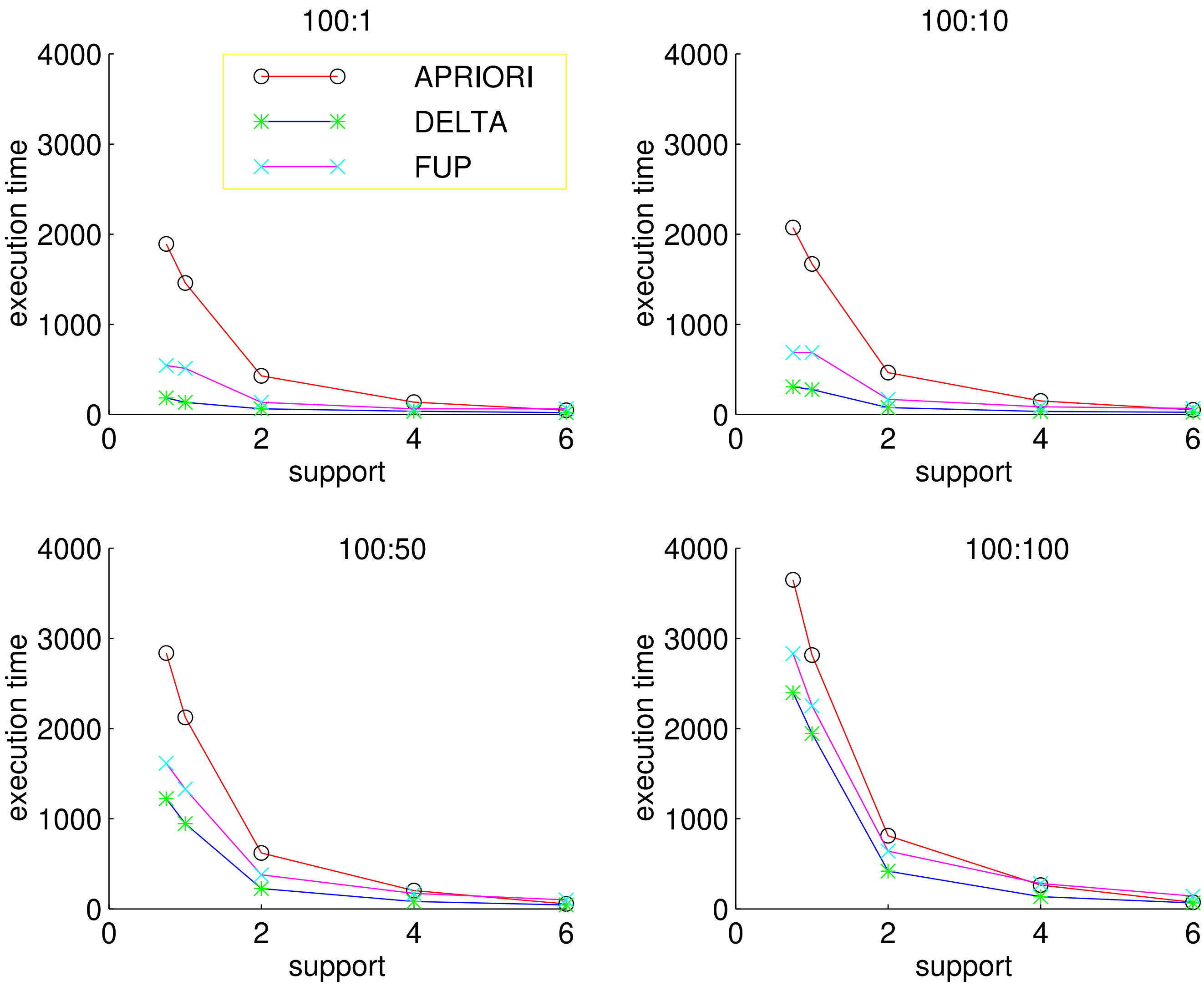

Figure 6.3: Moderate Skew: Execution time vs support

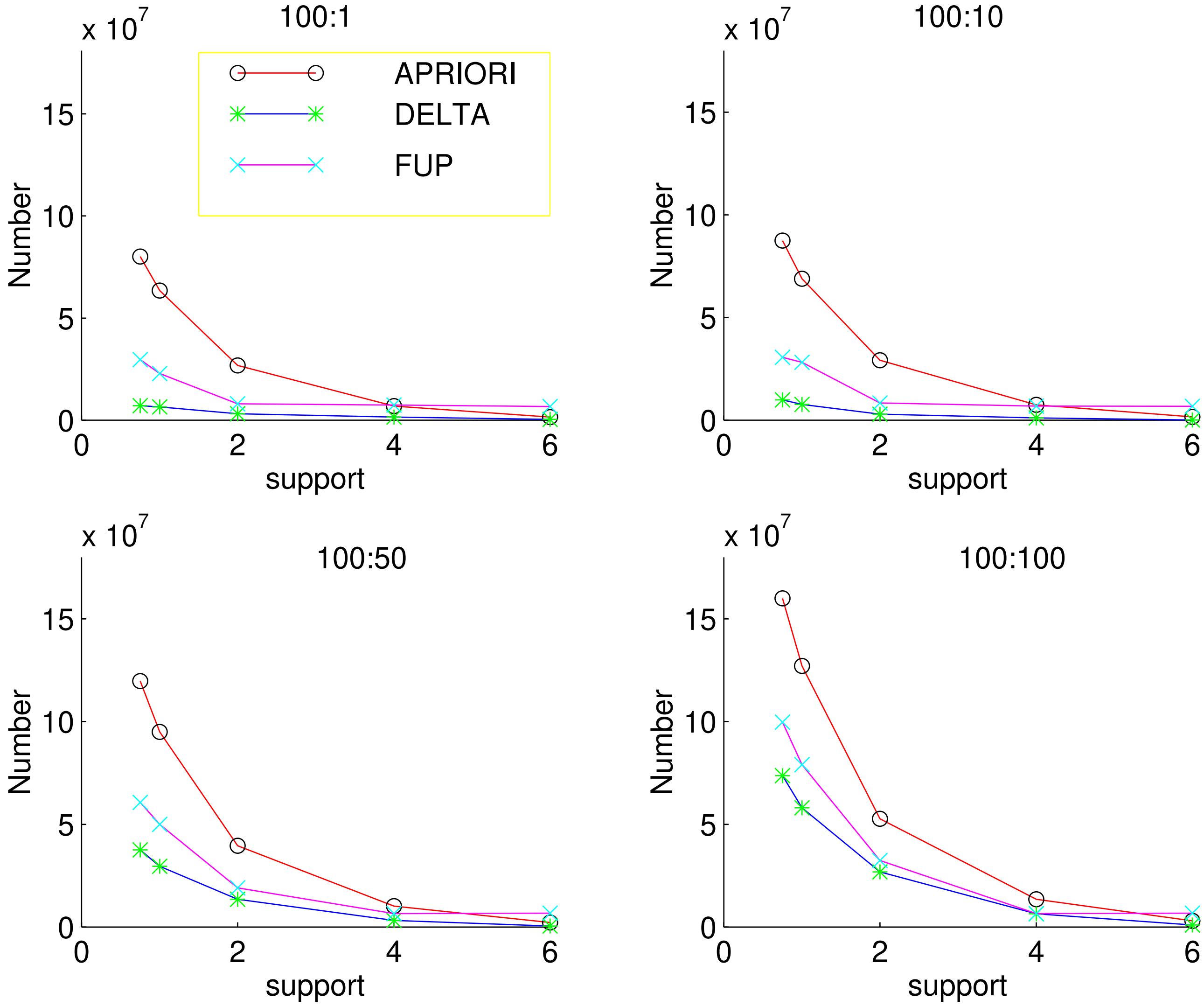

Figure 6.4: Uniform Distribution: Number of comparisons vs support

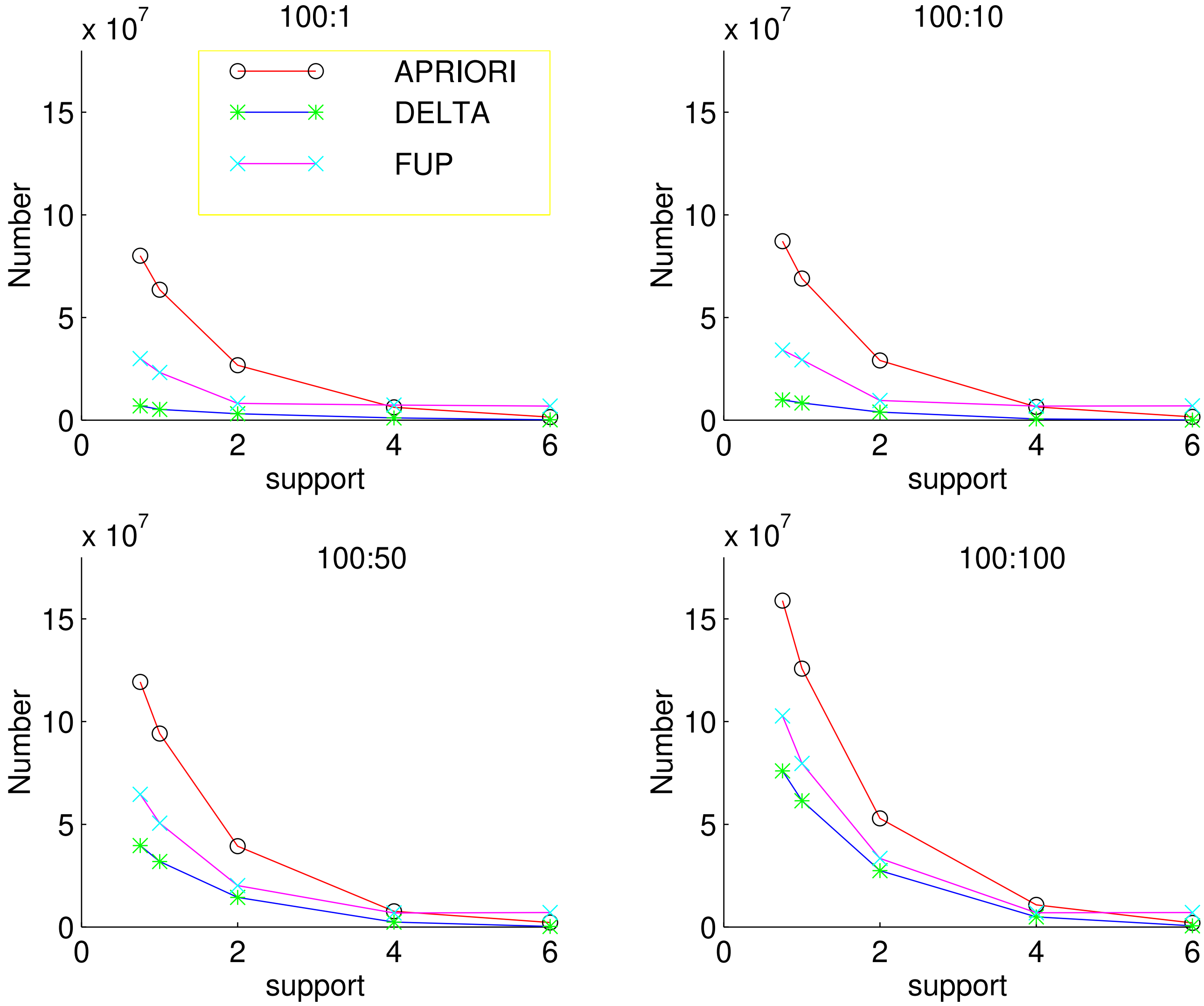

Figure 6.5: High Skew: Number of comparisons vs support

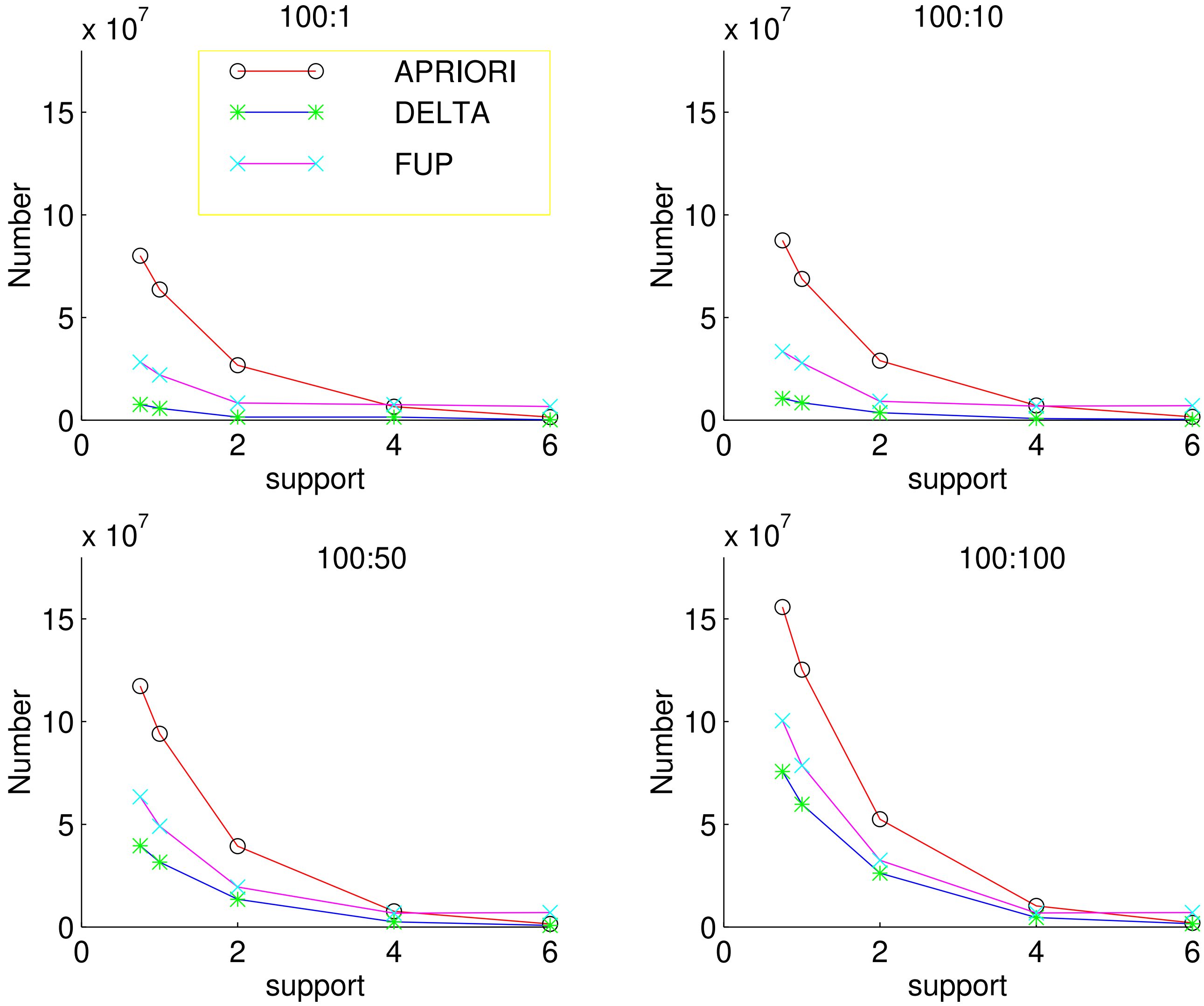

Figure 6.6: Moderate Skew: Number of comparisons vs support

Chapter 7

Implementation of Mining Tool

As mentioned in the introduction, the first phase of the implementation of *MINTO* is done. Here efficient algorithms to find association rules are implemented. *MINTO* is a data miners tool where a miner can choose different types of algorithms, based on the persons requirement. It has got two types of user interface. One is graphical user interface. We have given a SQL like query interface too to the tool. The graphical user interface is given for the ease of use of the tool. Even the people who do not know SQL can use the tool. But on the other hand, people who are familiar with SQL might want to use the tool without having to use the graphical interface, as it is much faster. And if SQL like interface is given, it is easier to interface it within other programs written in some other languages. Moreover, we can run the tool even if X-windows are not supported in the system, as we have this query interface.

MINTO is developed on a Unix platform using C++. The query interface is built using lex and yacc utilities. Graphical user interface makes use of X and motif libraries. The whole effort involved about 15000 lines of C++ code.

7.1 Organization of MINTO

As mentioned in the introduction, only first phase of *MINTO* is implemented now. This facilitates a person to do mining for association rules. The organization of *MINTO* is given in Figure 7.1.

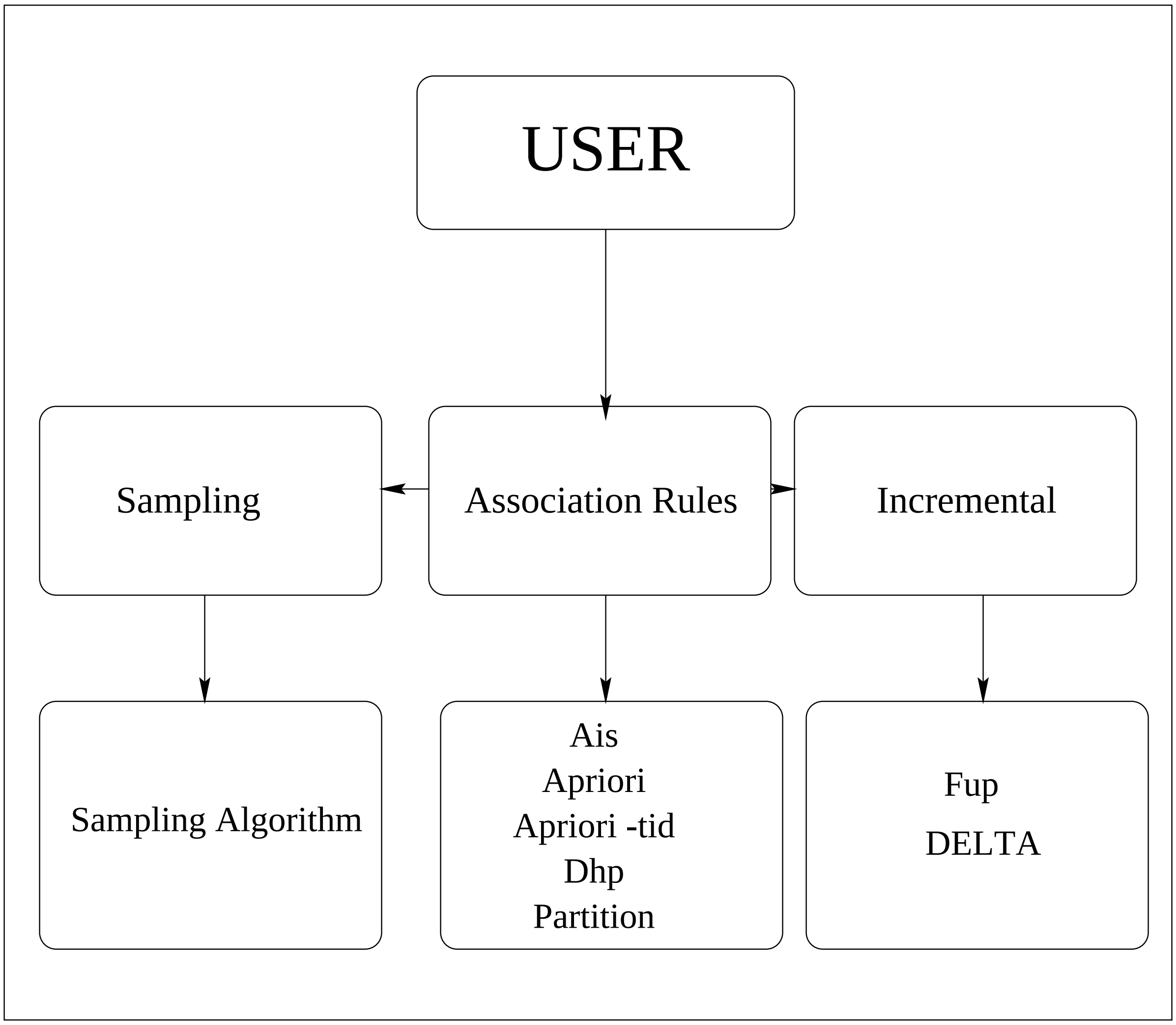

Figure 7.1: Organization of MINTO

There are five general algorithms, namely AIS, APRIORI, APRIORITID, DHP, and PARTITION, implemented to find association rules in *MINTO*. The tool also includes a mining algorithm based on *sampling*, which is explained in Section 4.1. In addition our mining tool gives user the flexibility of doing incremental mining. There are two algorithms to do incremental mining, viz., Fup given in 5.2 and DELTA, which is a new algorithm developed by us and was explained in detail in 5.3.

7.2 Types of Queries

There are three types of queries in data mining as shown in Figure 7.2. These are

- Queries with syntactic constraints.
- Queries with support constraints.
- Queries with syntactic and support constraints.

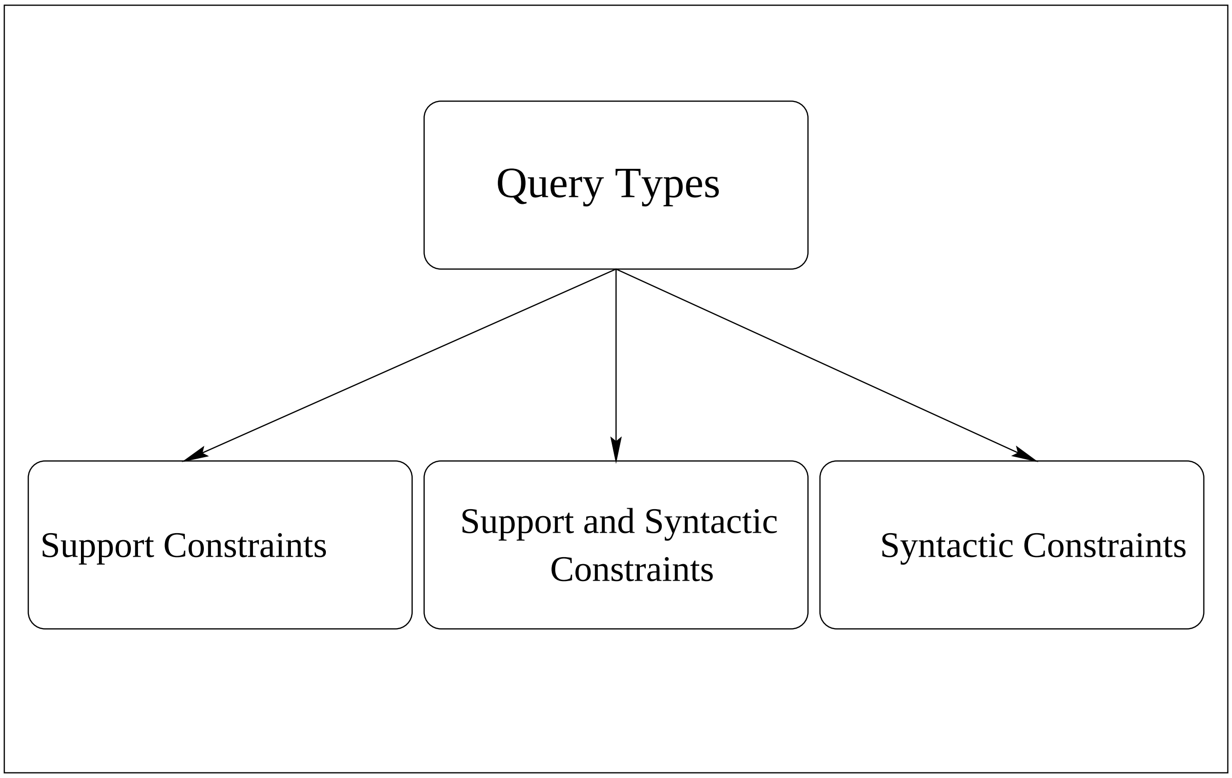

Figure 7.2: Types of Queries

7.2.1 Queries with support constraints

The first type of query is where we are given a rule and asked to find the support and confidence of the rule in the database. This is a simple query to answer as we just have to scan the database once to get the support and confidence in this case, and hence is done in one pass. For example consider the database given in Section 1.2.4. An example query of this type is

find out the confidence and support of the rule

$$potatoes, onions \Longrightarrow tomatoes$$

The answer to this query will be

$$potatoes, onions \Longrightarrow tomatoes \mid (75, 60)$$

Here, the confidence of the rule is 75%, while the support of the rule is 60%.

7.2.2 Queries with syntactic constraints

The second type of query is where we are asked to find the rules which have some minimum support and minimum confidence. Hence, here we say that *support constraints* are given and we are asked to find the rules which satisfy these constraints. An example query of this kind will be

find out all queries in the database given in 1.2.4 with

$support \geq 50\%$ and $confidence \geq 75\%$

Its answer is be given as :

- $potatoes \Longrightarrow onions \mid (89, 80)$
- $onions \Longrightarrow potatoes \mid (100, 80)$
- $onions \Longrightarrow tomatoes \mid (75, 60)$
- $tomatoes \Longrightarrow potatoes \mid (100, 60)$
- $tomatoes \Longrightarrow onion \mid (100, 60)$
- $potatoes, onions \Longrightarrow tomatoes \mid (75, 60)$
- $potatoes, tomatoes \Longrightarrow onions \mid (100, 60)$
- $onions, tomatoes \Longrightarrow potatoes \mid (100, 60)$

7.2.3 Queries with syntactic and support constraints

The third type of query is where syntactic constraints are given along with support constraints. In these sort of queries we are asked to find out all rules with X in the antecedent and Y in consequent with support and confidence greater than the given thresholds. As an example we can say

find out all queries in the database given in 1.2.4 with

$support \geq 50\%$ and $confidence \geq 60\%$ with "potatoes" in the antecedent.

The answer to the above query is :

- $potatoes \Longrightarrow onions \mid (89, 80)$
- $potatoes, onions \Longrightarrow tomatoes \mid (75, 60)$
- $potatoes, tomatoes \Longrightarrow onions \mid (100, 60)$

7.3 Implementation Details

The *MINTO* tool is organized into five basic modules, the data structure module, the algorithm module, the GUI module, the parser module, and a module to interface with the underlying database.

Data Structures All the major data structures used in the tool are present in this module. It has got complicated data structures like hash tree. All the mining algorithms use the data structures given here.

Algorithms In this module all the mining algorithms are placed.

Graphical User Interface This gives the motif and X-toolkit based user interface.

Parser Interface We have added a SQL-like query interface to MINTO. The parser for the query interface can be found in this module.

Link with the underlying Database Management System For the time being, we have our own DBMS, which has the minimum functionality required to do mining.

But with slight modifications to this unit, any DBMS can be used as the underlying DB unit.

7.3.1 Data Structures Module

This module gives all the major data structures used in our data mining tool.

All the items in the database are given distinct consecutive numbers. Set of items is given by a linked list of the numbers corresponding to the items. This class *ITEMSET* is nothing but a sorted linked list, whose elements are numbers. This is the basic unit in which any set of items is represented in MINTO. General set operations like union, intersection, add an element to set, find elements in the set are provided along with other things. Along with the link list, an integer variable *count* is added with each data structure to represent the number of times this itemset is appeared in the database considered. This variable is added since, we are interested in the support for the itemset in the database.

Since, most of the algorithms consider sets of itemsets. A data structure is required to represent sets of itemsets. This again is represented as a linked list of itemsets.

Algorithms like APRIORI and DHP use a *hash tree* data structure. Hash tree is also a way of representing sets of itemsets. The advantage of hash tree data structure is that it is easier to determine the presence or absence of a set of items. Also, finding out all the subsets of a given itemset present in the given sets of itemsets is made significantly easier and faster with the help of this data structure.

An example hash tree is given in Figure 7.4. There are three itemsets shown in this figure: $\{2, 3\}$, $\{2, 4\}$, and $\{2, 10\}$. This is only a part of the entire hash tree.

All the nodes in the hash tree start as leaf nodes. But if the additional itemset cannot be added to this leaf node, the node is converted into an internal hash table node and new leaf nodes are created.

Each leaf node has a set of integers. These integers will represent the set of items along with its support count. Since, leaf node represents these set of items as an array of integers, -1 is the end marker of a set of items and the integer following it will be the support count for the itemset. In the hash tree in Figure 7.4, we have support count for

{2,3} is 100, and that for {2, 4} is 36. The nodes at level 0 and level 1 are hash tables and nodes at level 2 are leaf nodes.

R. Agrawal does not specify a hash function which is used in the hash trees. In our implementation we have used the hash function given by

$$H_{level}(itemset) = \begin{cases} 0 & \mid itemset \mid < level \\ itemset_{level} \ mod(n-1) & otherwise \end{cases}$$

where n is one less than the number of entries in the hash table, as the hash function. The first element in the hash table is reserved for those itemsets, which cannot be hashed further because the number of elements in the itemset is one less than the level of the node.

Which is the exact location of a itemset in the hash tree? First at level 0, i.e., the root of the tree, we go in the branch $H_0(itemset)$. Then at level i, trace the branch $H_i(itemset)$ till we reach a leaf, and the given itemset will belong to that leaf.

To find all the subsets of a given set located in a hash tree, we follow the following principle. Let itemset be $I = I_0, I_1, I_2, \cdots I_k$. At root follow all the subtrees given by $H(I_0), H(I_1), H(I_2) \cdots H(I_k)$. At level i for each subtree got at level i-1 follow all those subtrees given by $H(I_i), H(I_{i+1}), H(I_{i+2}) \cdots H(I_k)$. Once, a leaf is reached, find all those itemsets, which are the subsets of the given set. In this way, only a few of the leaf nodes have to be checked for the presence of subsets.

Aprioritid algorithm explained in Section 3.3 uses a unique data structure. Here each itemset has an associated generator and extension with it. Each itemset is given an unique number. Generator of an k-itemset gives the unique identifiers of the k-1 itemsets from which it is generated. Extension of a k-itemset on the other hand will have identifiers of all those k+1-itemsets which are got as an extension of this itemset.

Finally, in partition algorithm, we maintain a list of transaction ids with the itemsets with single elements.

7.3.2 Algorithm Module

The code for all the different algorithms are incorporated in this module. As mentioned previously we have coded eight different algorithms, namely, AIS [AIS93a], APRIORI [AS94], APRIORITID [AS94], DHP [PCY95], PARTITION [SON95], SAMPLING [Toi96], FUP [CHN+96], and DELTA.

The first five of these algorithms are used for obtaining *association rules*. Sampling algorithm is an algorithm to find out association rules, but it is based on sampling. On the other hand, FUP and DELTA are incremental minining algorithms.

These algorithms use the data structures from data structure module. They interact with the underlying database through the linking module.

7.3.3 Graphical User Interface

In order to make the tool user friendly a graphical user interface (GUI) is provided. This GUI is based on X-Motif and hence can be run on any X-environment. The GUI gives the facility to ask the different types of questions asked by the user. As explained before there are three types of queries asked by the user on a mining tool. GUI has two different forms in which these queries can be asked and the results will be displayed in a user friendly fashion. The main window and some of the forms present in the tool are given in Figures.

There are five general algorithms, namely AIS, APRIORI, APRIORITID, DHP, and PARTITION. The interaction between the different modules of the tool is depicted in the Figure 7.3.

7.3.4 Query Interface

We have given a SQL like query interface to MINTO. This grammar is implemented using lex and yacc. As explained in Section 7.2 there are basically three types of queries asked.

Syntactic Queries

For syntactic queries explained in Section 7.2.2, which are of the sort, where given a rule, one is asked to find the support or confidence or both. For example a query of the sort,

find out the confidence and support of the rule

$potatoes, onions \Longrightarrow tomatoes$

belongs to this category. This query can be specified in our language as

select support confidence
from database
where rule is $potatoes, onions \Longrightarrow tomatoes$

These sort of queries can be given in general as

select *[support] [confidence]*
from *database*
where rule is $[rule]^+$

Support Queries

As seen in Section 7.2.1 there are queries with support constraints. In this case, given a minimum threshold for support and confidence one is asked to find out all the rules in the given database, which satisfy this minimum threshold conditions. The query

find out all queries in the database given in 1.2.4 with

$support \geq 20\%$ and $confidence \geq 60\%$

belongs to this category. This query can be asked as

select rule
from database
where support = 0.2 confidence = 0.6

These sort of queries can be put as

select *rule*
from *database*
where *[support = s] [confidence = c]*

Since, we have implemented eight different algorithms to do the mining, users are given the option of choosing whichever algorithm they want. By default algorithm APRIORI will be chosen to do the mining. Then the query will have the form

select *rule*
from *database*
where *[support = s] [confidence = c]*
using *algorithm*

Support and Syntactic Queries

That leaves us with the queries in which both the above constraints will be combined. These are explained in Section 7.2.3.

find out all queries in the database given in 1.2.4 with
$support \geq 20\%$ and $confidence \geq 60\%$ with "potatoes" in the antecedent.

will be an example of such a query. This query can be expressed as

select *rule*
from *database*
where *[support = 0.2] [confidence = 0.6]*
having *potatoes in antecedent*

A general query in this category can be represented as

select *rule*
from *database*
where *[support = s] [confidence = c]*
having *[item+ in antecedent] [item+ in consequent]*

7.3.5 Interaction with the DBMS

MINTO exists on top of a DBMS. It is built on a layer above that of a DBMS layer. Therefore a unit using which MINTO can interact with the DBMS, to get the DBMS functionalities like finding out the number of attributes, number of transactions, and getting the transactions becomes necessary. This module provides this interaction.

This unit is put as a separate module, because it will be the only module, which is dependent on the underlying DBMS. Other modules do not depend on the underlying DBMS. Hence, if we decide to change the underlying DBMS, it is easy, as only this module has to be changed.

For the time being, we have written a simple DBMS, which will just do the functions needed for MINTO, and does not have the other typical DBMS features. The functionalities included are getting the size of the relation, number of attributes in the relation and also getting the tuples of the relation one by one. But with slight modifications to the module, MINTO can be made to work with any underlying DBMS.

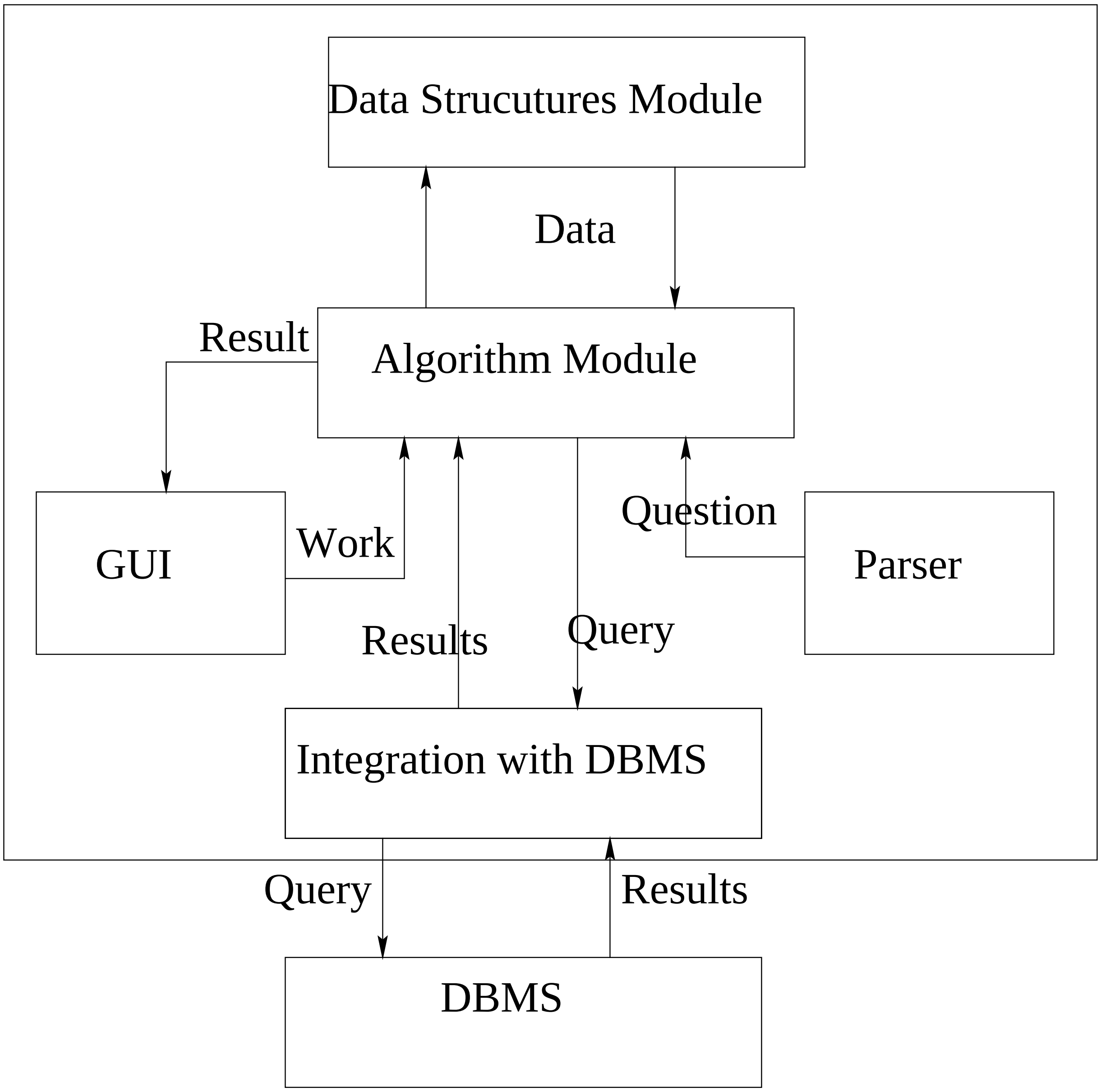

Figure 7.3: Organization of Modules in MINTO

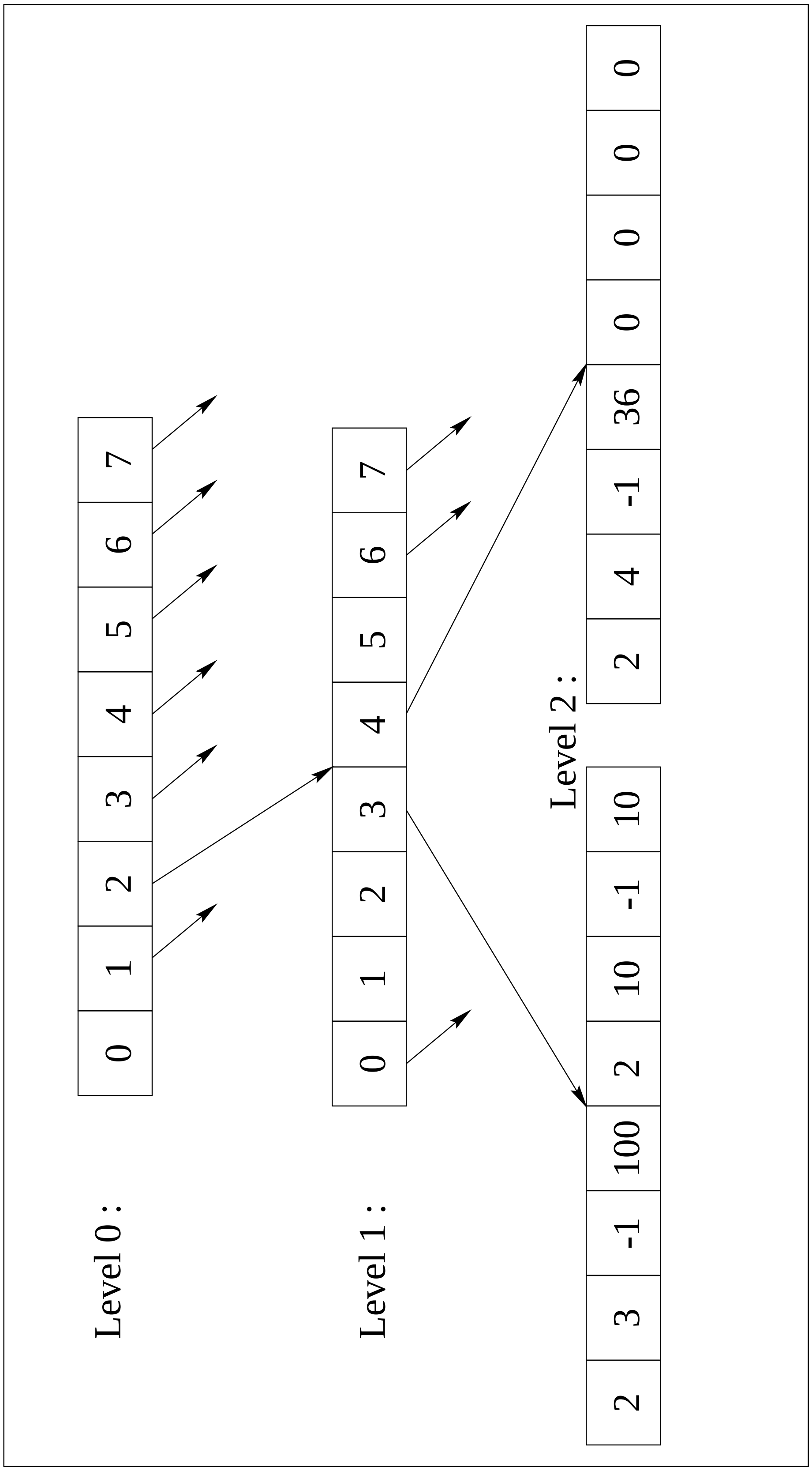

Figure 7.4: An example hash tree

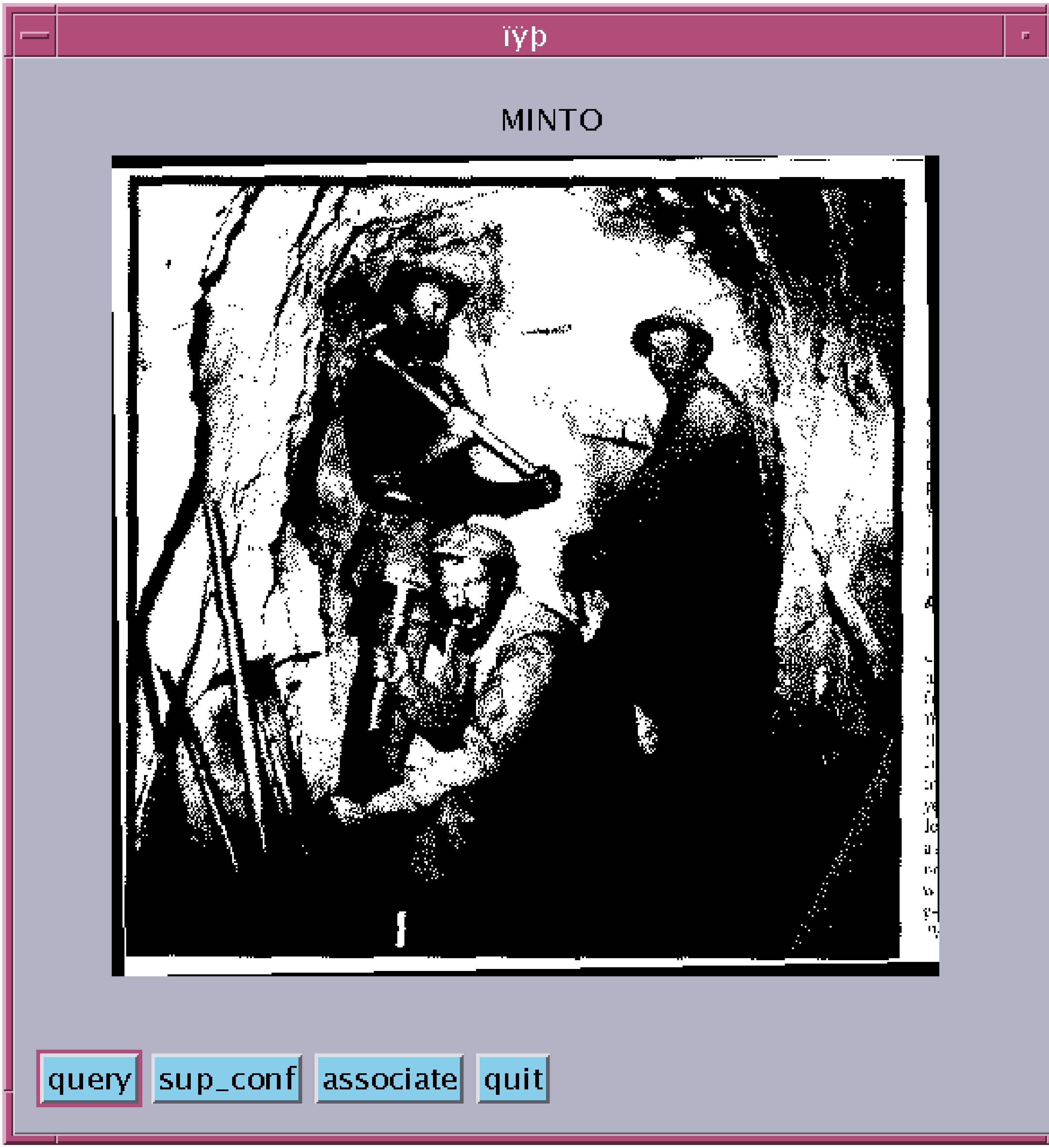

Figure 7.5: Frontview of MINTO

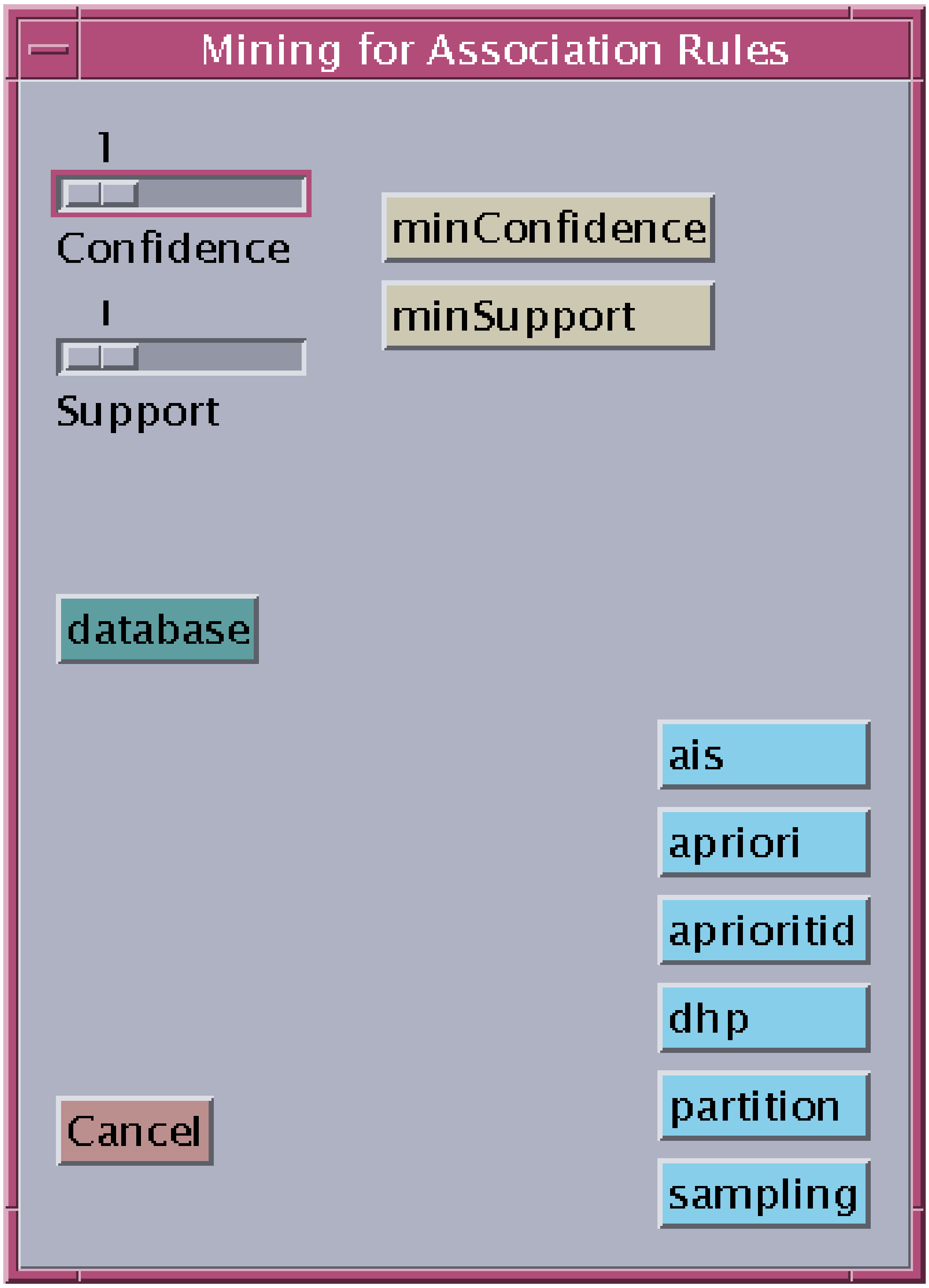

Figure 7.6: Form 1

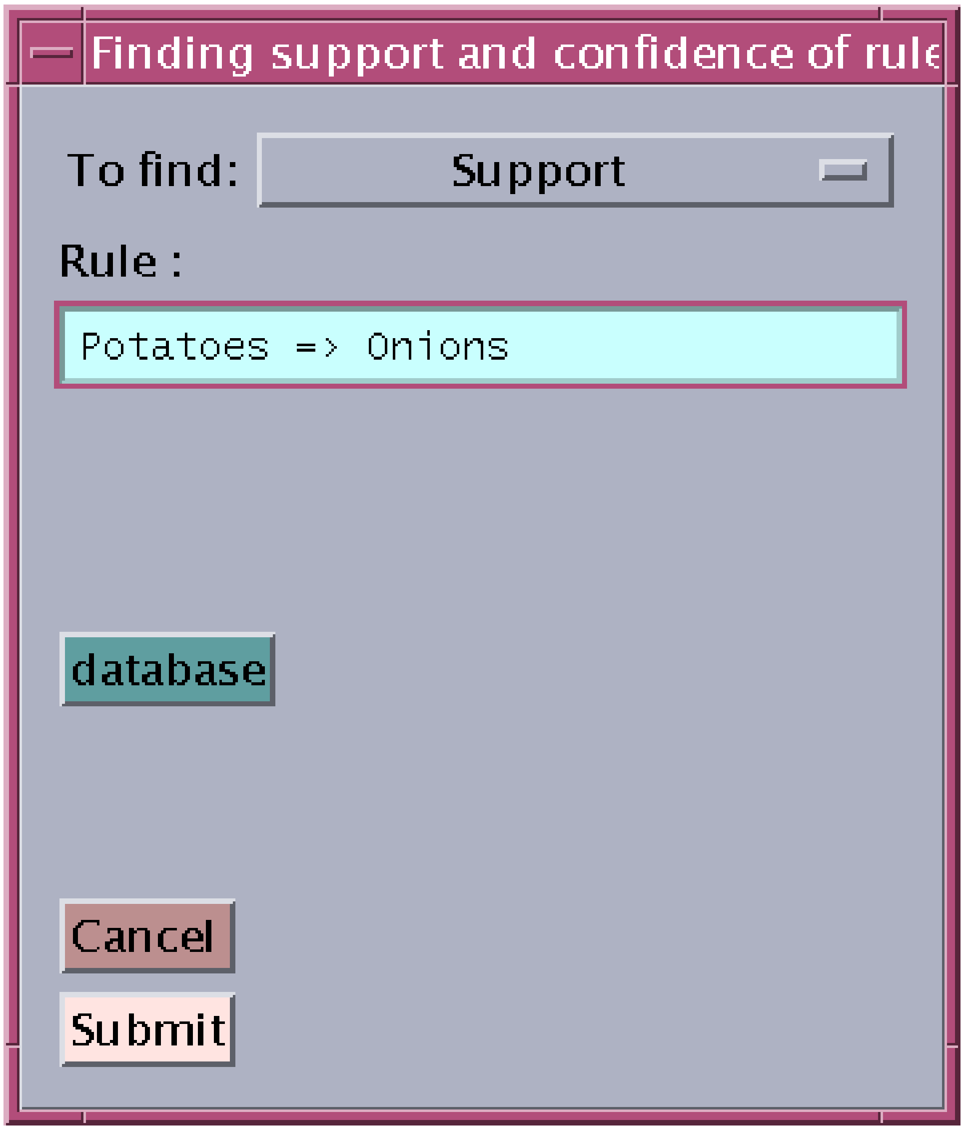

Figure 7.7: Form 2

Chapter 8

Conclusions

The concept of data mining came up because people wanted to excavate useful information from the enormous amount of data collected. There are three types of mining rules, association, classification, and time series.

8.1 Summary

We have developed a mining tool. In particular it does the mining for association rules. It has got two types of interfaces, one is graphical, built using X-toolkit and motif, and the other is a query interface. The query language developed can be used as an extension to SQL.

8.1.1 Algorithms to find Association Rules

AIS, APRIORI, APRIORITID, DHP, and PARTITION algorithms are implemented and can be used to mine the databases. AIS is the first data mining algorithms and is the one, where mining is divided into two parts, to find large itemsets and then to find out the rules from the set of large itemsets. Apriori and Aprioritid are k pass algorithms, where k is the number of elements in the largest large itemset. Dhp is again a k pass algorithm, but it uses the hash table technique to reduce the number of candidate elements, and hence will improve the performance. Partition, uses divide and conquer policy and does mining in two passes.

8.1.2 Sampling

The performance of the mining algorithm depends on the size of the database to be mined. Hence, sampling techniques are thought of to improve the performance. There are two ways to do sampling. In one, only the sample of the database will be used and hence, the results will not be entirely correct. But if the entire database is used at the end to find out the support then we can get a correct result and an sampling algorithm based on the second technique is implemented.

8.1.3 Incremental Mining

If some more elements are added to the database after it is mined, then the database has to be mined again, as these additional elements might cause some change in the large itemsets. Some of the large itemsets, which were originally large might cease to remain so, after the addition. On the other hand, some other itemset may become large after the new elements are added to the database. Hence, mining the database again becomes necessary. Instead of mining the whole database again, we try to use the previous results, that is the itemsets which are large in the original database and their counts, along with the database to find the large itemsets in the changed database. Such a technique is called incremental mining. We have implemented FUP, an algorithm which does incremental mining.

A new algorithm DELTA is also developed to do incremental mining. The performance of DELTA is compared as against apriori and FUP. It is found out that DELTA performs much better than both Apriori and FUP. Moreover, the performance of DELTA does not degrade with the change in distribution of the original database and the addition.

8.2 Future Work

The mining tool at present can find only association rules. We can enhance its functionality by adding the algorithms to find classification rules and time series rules, etc.

Only one type of sampling is considered. One can think of adding the other type of sampling, viz., sampling results can have some percentage error, where the percentage of error is given apriori.

Since mining is a highly time consuming operation, parallelism will be of much importance to improve the efficiency of the mining tool. One can think of parallel mining algorithms and implement them, which will improve the performance of the mining tool.

The DELTA algorithm can be improved further. One can think of combining sampling with incremental mining. That is, in DELTA we keep and use the entire original database when needed. But instead we can keep only a sampled version of the original database and use it. The repercussion of this sampling has to be studied carefully.

The query interface provided is very simple. The grammar can be enhanced to have additional functionalities.

Though the mining tool is developed at a level above a data management system, we have not linked it to any database management system in particular. Using the module which links the mining tool with the DBMS, MINTO can be made to work in correlation with any DBMS.

Bibliography

[Agr96] R. Agrawal, et al, "Quest: A Project on Database Mining",

[AFS93] R. Agrawal, C. Falutsos, and A. Swami, "Efficient Similarity Search in Sequence Databases", *4th International Conference on Foundations of Data Organization and Algorithms*, October 1993.

[AGI+92] R. Agarwal, S. Ghosh, T. Imielinski, H. Iyer, and A. Swami, "An Interval Classifier for Database Mining Applications", *Proceedings of the 18th International Conference on Very Large Data Bases*, September 1992.

[AIS93a] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *Proceedings of 22nd ACM SIGMOD International Conference on Management of Data*, May 1993.

[AIS93b] R. Agrawal, T. Imielinski, and A. Swami, "Database Mining: A Performance Perspective", *IEEE Transactions on Knowledge and Data Engineering*, December 1993.

[ALS+95] R. Agrawal, K. Lin, H. Sawhney, and K. Shim, "Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[AMS+96] R. Agrawal, M. Mehta, H. Shafer, and R. Srikant, "The Quest Data Mining System", *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, September 1996.

[AP95] R. Agrawal, and G. Psaila, "Active Data Mining", *Proceedings of the 1st International Conference on Knowledge Discovery in Databases and Data Mining*, 1995.

[APW+95] R. Agrawal, G. Psaila, E. Wimmers, and M. Zait, "Querying Shapes of Histories.", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[AS96a] R. Agarwal, and J. Shafer, " Parallel Mining of Association Rules: Design, Implementation and Experience", *IEEE Transactions on Knowledge and Data Engineering*, January 1996.

[AS96b] R. Agarwal, and K. Shim, "Developing Tightly-Coupled Data Mining Applications on a Relational Database System", *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[AS94] R. Agrawal, and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proceedings of the 20th International Conference on Very Large Data Bases*, September 1994.

[AS95] R. Agrawal, and R. Srikant, "Mining Sequential Patterns", *Proceedings of the 11th International Conference on Data Engineering*, March 1995.

[AAR96] A. Arning, R. Agrawal, and J. Rissanen, "A Linear Method for Deviation Detection in Large Databases", *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, March 1996.

[BWJ96] X. Bettini, X. Wang, and S. Jajodia, "Mining Event Sequences for Complex Temporal Relationships Involving Multiple Granularities", *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1996.

[CHY96] M. Chen, J. Han, and S. Yu, "Data Mining: An Overview from a Database Perspective", *IEEE Transactions on Knowledge and Data Engineering* , December 1996.

[CHN+96] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu, "A Fast Distributed Algorithm for Mining Association Rules", *4th International Conference on Parallel and Distributed Information Systems*, December 1996.

[FL95] C. Faloutsos, and K. Lin, "FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets", *Proceedings of 24th ACM SIGMOD International Conference on Management of Data*, May 1995.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast Subsequence Matching in Time-Series Databases", *Proceedings of 23rd ACM SIGMOD International Conference on Management of Data*, May 1994.

[FMM+96a] T. Fukuda, Y. Morimoto, S. Morishita, T. Tokuyama, "Mining Optimized Association Rules for Numeric Attributes", *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1996.

[FMM+96b] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, "Data Mining Using Two-Dimensional Optimized Association Rules: Scheme, Algorithms, and Visualization", *Proceedings of 25th ACM SIGMOD International Conference on Management of Data*, June 1996.

[FMM+96c] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, "Constructing Efficient Decision Trees by Using Optimized Numeric Association Rules", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[GHK+96] M. Ganesh, E. Han, V. Kumar, S. Shekhar, and J. Srivastava, "Visual Data Mining: Framework and Algorithm Development", *Technical Report*, University of Minnesota, March 1996.

[Han96] J. Han, "Data Mining Techniques", *Proceedings of 25th ACM SIGMOD International Conference on Management of Data*, June 1996.

[HCC92] J. Han, Y. Cai, and N. Cercone, "Knowledge Discovery in Databases: An Attribute-Oriented Approach", *Proceedings of the 18th International Conference on Very Large Data Bases*, August 1992.

[HF95] J. Han, and Y. Fu, "Discovery of Multiple-Level Association Rules from Large Databases", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[HFW+96] J. Han, Y. Fu, W. Wang, J. Chiang, O. Zaiane, and K. Koperski, "DBMiner: Interactive Mining of Multiple-Level Knowledge in Relational Databases", *Proceedings of 25th ACM SIGMOD International Conference on Management of Data*, June 1996.

[HKK97] E. Han, G. Karypis, and V. Kumar, "Scalable Parallel Data Mining for Association Rules", *Proceedings of 26th ACM SIGMOD International Conference on Management of Data*, June 1997.

[HNK93] J. Han, S. Nishio, and H.Kawano, "Knowledge Discovery in Object-Oriented and Active Databases" *Proceedings of 22nd ACM SIGMOD International Conference on Management of Data*, May 1993.

[HSK+96a] E. Han, S. Shekhar, V. Kumar, M. Ganesh, and J. Srivastava, "Search Framework for Mining Classification Decision Trees", *Technical Report*, University of Minnesota, April 1996.

[HSK96b] E. Han, A. Srivastava, V. Kumar, "Parallel Formulations of Inductive Classification Learning Algorithm", *Technical Report*, University of Minnesota, May 1996.

[Har96] J. Haritsa, "Database Mining", *RESONANCE*, October 1996.

[HS95] M. Houtsma, and A. Swami, "Set-Oriented Mining Association Rules", *International Conference on Data Engineering*, March 1995.

[JL95] G. John, and P. Langley, "Static Versus Dynamic Sampling for Data Mining", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[KMR+94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. Verkamo, "Finding Interesting Rules from Large Sets of Discovered Association Rules", *Third International Conference on Information and Knowledge Management*, Dec 1994.

[LSL95] H. Lu, R. Seiono, and H. Liu, "NeuroRule: A Connectionist Approach to Data Mining", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[MTI94a] H. Mannila, H. Toivonen, and A. Inkeri-Verkamo, "Efficient Algorithms for Discovering Association Rules", *AAAI Workshop on Knowledge Discovery in Databases*, July 1994.

[MTI94b] H. Mannila, H. Toivonen, and A. Inkeri-Verkamo, "Discovering Frequent Episodes in Sequences", *International Conference on Knowledge Discovery in Databases and Data Mining*, August 1995.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen, "SLIQ: A Fast Scalable Classifier for Data Mining", *Proceedings of the fifth International Conference on Extending Database Technology*, March 1996.

[MRA95] M. Mehta, J. Rissanen, and R. Agrawal, "MDL-based Decision Tree Pruning", *1st International Conference on Knowledge Discovery and Data Mining*, August 1995.

[MPC96] R. Meo, G. Psaila, and S. Ceri. "A new SQL-like Operator for Mining Association Rules", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[NRV96] J. Nearhos, M. Rothman, and M. Viviros, "Applying Data Mining Techniques to a Health Insurance Information System", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[NH94] R. Ng, and J. Han, "Efficient and Effective Clustering Methods for Spatial Data Mining", *Proceedings of the 20th International Conference on Very Large Data Bases*, September 1994.

[PCY95] J. Park, M. Chen, and P. Yu, "An Effective Hash-Based Algorithm for Mining Association Rules", *Proceedings of 24th ACM SIGMOD International Conference on Management of Data*, May 1995.

[PYC96] J. Park, P. Yu, and M. Chen, "Mining Association Rules with Adjustable Accuracy", Submitted to VLDB96.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[SSW96] P. Selfridge, D. Srivastava, and L. Wilson, "IDEA: Interactive Data Exploration and Analysis", *Proceedings of 25th ACM SIGMOD International Conference on Management of Data*, June 1996.

[SAM96a] J. Shafer, R. Agrawal, and M. Mehta, "Fast serial and Parallel Classification of Very Large Data Bases", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[SAM96b] J. Shafer, R. Agrawal, and M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[SK96] T. Shintani, and M. Kitsuregawa, "Hash Based Parallel Algorithms for Mining Association Rules", *4th International Conference on Parallel and Distributed Information Systems*, December 1996.

[SA95] R. Srikant, and R. Agrawal, "Mining Generalized Association Rules", *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[SA96a] R. Srikant, and R. Agrawal, "Mining Quantitative Association Rules in Large Relational Tables", *Proceedings of 25th ACM SIGMOD International Conference on Management of Data*, June 1996.

[SA96b] R. Srikant, and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements", *Proceedings of the Fifth International Conference on Extending Database Technology*, March 1996.

[SLW96] S. Sung, T. Ling, and W. Wu, "Mining Association Rules using Fragmentation", Submitted to VLDB96.

[Toi96] H. Toivonen, "Sampling Large Databases for Association Rules", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[TSW96] W. Tong, K. Sankaran, and B. Wuthrich, "DISCOVER: A Database Mining Tool", *Submitted to VLDB96.*

[WCM+94] J. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Shang, "Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results", *Proceedings of 24th ACM SIGMOD International Conference on Management of Data*, May 1994.

[Wan96] Ke Wang, "Generic Association Patterns: An Extensible Data Mining Model", *Submitted to VLDB96.*

[YC95] S. Yen, and A. Chen, "An Efficient Algorithm for Deriving Compact Rules from Databases", *4th International Conference on Database Systems for Advanced Applications*, April 1995.

[YC96] S. Yen, and A. Chen, "An Efficient Approach to Discovering Knowledge from Large Databases", *4th International Conference on Parallel and Distributed Information Systems*, December 1996.

[SGI96] Database Mining and Visualization Group, SGI inc, "MineSetTM: A System for High-End Data Mining and Visualization", *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.