PROJECT TECHNICAL REPORT
for
DST Project III. 5(71) / 96 - ET

Title	**:**	**MINTO: A Software Tool for Mining Manufacturing Databases**
Sponsor	**:**	**Dept. of Science and Technology Government of India**
Principal Investigator	**:**	**Jayant Haritsa Associate Professor Supercomputer Education and Research Centre**
Co-Investigator	**:**	**–**

July 1999

Centre for Sponsored Schemes and Projects

Indian Institute of Science

Bangalore 560012, India

Contents

1 Introduction to Database Mining **1**
1.1 Pattern Discovery . . . 1
1.2 Rules . . . 2
1.3 Rule Discovery . . . 3
1.4 Frequent Itemset Generation . . . 3
1.5 Strong Rule Derivation . . . 4
1.6 Classification and Sequence Rules . . . 4
1.7 Summary . . . 4

2 Mining in the Manufacturing Context **5**
2.1 Current Status of R & D . . . 6
2.2 Mining Model . . . 6
2.2.1 Solution Strategy . . . 7
2.2.2 Generalized and Quantitative Association Rules . . . 7
2.2.3 Report Organization . . . 7

3 First-Time Data Mining Algorithms **8**
3.1 The AIS Algorithm . . . 8
3.2 The Apriori Algorithm . . . 10
3.2.1 Data Structures . . . 11
3.3 The Aprioritid Algorithm . . . 11
3.3.1 Data Structures . . . 11
3.4 The DHP Algorithm . . . 12
3.4.1 Reduction in Database Size . . . 12
3.5 The Partition Algorithm . . . 14
3.6 Speedup through Sampling . . . 16
3.7 The Cumulate Algorithm . . . 17
3.8 The QAR Algorithm . . . 17
3.9 The CountDistribute Parallel Algorithm . . . 18

4 The TWOPASS Algorithm **19**
4.1 Algorithmic Details . . . 19
4.2 Proof of Correctness . . . 21
4.3 Negative Border Closure . . . 21
4.4 Size of Partitions . . . 22
4.5 Number of Candidate Itemsets . . . 22
4.6 Incorporating Sampling . . . 22

5 Incremental Mining **23**
5.1 Incremental Mining . . . 24
5.2 Previous Incremental Algorithms . . . 25
5.2.1 The FUP Algorithm . . . 25
5.2.2 The TBAR Algorithm . . . 25
5.2.3 The Borders Algorithm . . . 25
5.2.4 Other Algorithms . . . 26
5.3 The DELTA Algorithm . . . 26

5.3.1 DELTA Processing Details . . . 27
5.3.2 Intuition Behind the DELTA Design . . . 28
5.3.3 No Scan over DB Scenario . . . 29
5.3.4 Deletion of Tuples . . . 29
5.4 Multi-Support Incremental Mining in DELTA . . . 29
5.4.1 Stronger Support Threshold . . . 29
5.4.2 Weaker Support Threshold . . . 29
5.5 Performance Study . . . 31
5.5.1 Baseline Algorithms . . . 31
5.5.2 Database Generation . . . 31
5.5.3 Itemset Data Structures . . . 32
5.5.4 Overview of Experiments . . . 32
5.6 Experimental Results . . . 33
5.6.1 Experiment 1: Equi-support / Identical Distribution . . . 33
5.6.2 Experiment 2: Equi-support / Skewed Distribution . . . 34
5.6.3 Experiment 3: Multi-Support / Identical Distribution . . . 35
5.7 Handling Generalized Association Rules . . . 36
5.7.1 Performance Evaluation: Generalized / Equi-support / Identical Distribution . . . 36
5.8 Conclusions . . . 37

6 Rule Generation **39**

7 Vertical Mining **40**
7.1 Vertical Mining . . . 41
7.2 The VIPER Algorithm . . . 42
7.2.1 Overview . . . 42
7.2.2 Disk Traffic Reduction . . . 42
7.2.3 Snake Generation and Compression . . . 43
7.2.4 Execution Flow . . . 43
7.3 Snake Merging and Counting . . . 45
7.3.1 DAG Merge Optimization . . . 45
7.3.2 Delayed Write . . . 46
7.3.3 Write Cover and Write Pruning . . . 47
7.3.4 Focussed Writes . . . 48
7.4 Candidate Generation: ClusterGen . . . 48
7.5 Previous Vertical Mining Algorithms . . . 50
7.5.1 The Aggregate Algorithm . . . 50
7.5.2 The DLG Algorithm . . . 50
7.5.3 The Eclat Algorithm . . . 51
7.5.4 The Clique Algorithm . . . 52
7.5.5 The MaxEclat Algorithm . . . 52
7.5.6 The MaxClique Algorithm . . . 52
7.6 Performance Study . . . 53
7.6.1 Database Generation . . . 53
7.6.2 Implementation Details . . . 53
7.6.3 Baseline Experiment . . . 54
7.6.4 Sensitivity Analysis . . . 54
7.6.5 Comparison with Horizontal Algorithms . . . 55
7.6.6 Compression and Pruning . . . 55
7.7 Conclusions . . . 55

8 Implementation of MINTO **58**
8.1 Implementation Details . . . 58
8.1.1 Data Structures Module . . . 58
8.1.2 Algorithm Module . . . 59
8.1.3 Graphical User Interface . . . 59
8.1.4 Query Interface . . . 60

8.1.5 Interaction with the DBMS 60

9 Conclusions **66**

9.1 Future Work 66

Bibliography **67**

Chapter 1

Introduction to Database Mining

Organizations typically collect vast quantities of data relating to their operations. For example, the Income Tax department has several *terabytes* of data stored about taxpayers. In order to provide a convenient and efficient environment for users to productively use these huge data collections, software packages called "DataBase Management Systems" have been developed and are widely used all over the world.

A DataBase Management System, or DBMS, as it is commonly referred to, provides a variety of useful features: Firstly, business rules such as, for example, "the minimum balance to be maintained in a savings account is 100 Rupees", are not allowed to be violated. Secondly, modifications made to the database are never lost even if system failures occur subsequently. Thirdly, the data is always kept consistent even if multiple users access and modify the database concurrently. Finally, friendly and powerful interfaces are provided to ask questions on the information stored in the database.

Users of DBMSs interact with them in basic units of work called *transactions*. Transfer of money from one account to another, reservation of train tickets, filing of tax returns, entering marks on a student's grade sheet, are all examples of transactions. A transaction is similar to the notion of a task in operating systems, the main difference being that transactions are significantly more complex in terms of the functionality provided by them.

For large commercial organizations, where thousands of transactions are executed every second, highly sophisticated DBMSs such as DB2, Informix, Sybase, Ingres and Oracle are available and are widely used. For less demanding environments, such as managing home accounts or office records, there are a variety of PC–based packages, popular examples being FoxPro, Microsoft Access and dBase IV.

DataBase Management Systems typically operate on data that is resident on hard disk. However, since disk capacities are usually limited, as more and more new data keeps coming in, organizations are forced to transfer their old data to tapes. Even if these tapes are stored carefully, the information resident in them is hardly ever utilized. This is highly unfortunate since the historical databases often contain extremely useful information, as described below.

1.1 Pattern Discovery

The primary worth of the historical information is that it can be used to detect *patterns*. For example, a supermarket that maintains a database of customer purchases may find that customers who purchase coffee powder very often also purchase sugar. Based on this information, the manager may decide to place coffee powder and sugar on adjacent shelves to enhance customer convenience. The manager may also ensure that whenever fresh stocks of coffee powder are ordered, commensurate quantities of sugar are also procured, thereby increasing the company's sales and profits. Information of this kind may also be used beneficially in catalog design, targeted mailing, customer segmentation, scheduling of sales, etc. In short, the historical database is a "gold mine" that can be profitably used to make better business decisions.

In the technical jargon, data patterns (of the type described above) are called "rules" and the process of identifying such rules from huge historical databases is called **database mining**. Those familiar with learning techniques will be aware that discovering rules from data has been an area of active research in artificial intelligence. However, these techniques have been evaluated in the context of small (in memory) data sets and perform poorly on large data sets. Therefore, database mining can be viewed as the confluence of machine learning techniques and the performance emphasis of database technology. In particular, it refers to the efficient construction and verification of models of patterns embedded in large databases.

Customer	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: Vegetable Purchase Database

1.2 Rules

In normal usage, the term "rule" is used to denote implications that are always true. For example, Boyle's law *Pressure * Volume = constant* can be inferred from scientific data. For commercial applications, however, this definition is too restrictive and the notion of rule is expanded to denote implications that are *often*, but not necessarily *always*, true. To quantify this uncertainity, a *confidence factor* is associated with each rule. This factor denotes the probability that a rule will turn out to be true in a specific instance. For example, the statement "Ninety percent of the customers who purchase coffee powder also purchase sugar" corresponds to a rule with confidence factor 0.9.

In order for rules of the above nature to be meaningful, they should occur reasonably often in the database. That is, if there were a million customer purchases and, say, only ten of these customers bought coffee powder, then the above example rule would be of little value to the supermarket manager. Therefore, an additional criterion called the *support factor* is used to distinguish "significant" rules. This factor denotes the fraction of transactions in the database that support the given rule. For example, a customer purchase rule with support factor of 0.20 means that twenty percent of the overall purchases satisfied the rule.

At first glance, the confidence factor and the support factor may appear to be similar concepts. However, they are really quite different: Confidence is a measure of the rule's strength, while support corresponds to statistical significance.

Formal Definition

Based on the foregoing discussion, the notion of a rule can be formally defined as:

$$X \Longrightarrow Y \mid (c, s)$$

where X and Y are disjoint subsets of $\mathcal{I}$, the set of all items represented in the database, c is the confidence factor, and s is the support factor. The factors, which are ratios, are usually expressed in terms of their equivalent percentages.

To make the above definition clear, consider a vegetable vendor who sells potatoes, onions, tomatoes, carrots and beans, and maintains a database that stores the names of the vegetables bought in each customer purchase, as shown in Table 1. For this scenario, the itemset $\mathcal{I}$ corresponds to the set of vegetables that are offered for sale. Then, on mining the database we will find rules of the following nature:

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

$$Beans \Longrightarrow Potatoes \mid (80, 40)$$

which translate to "Seventy-five percent of customers who bought potatoes and onions also bought tomatoes. Sixty percent of the customers made such purchases" and "Eighty percent of the customers who bought beans also bought potatoes. Forty percent of the customers made such purchases", respectively.

A word of caution: The rules derived in the above example are not truly valid since the database used in the example is a "toy" database that has only ten customer records – it was provided only for illustrative purposes.

For rules to be meaningful, they should be derived from large databases that have thousands of customer records, thereby indicating consistent patterns, not transient phenomena.

1.3 Rule Discovery

As mentioned in the Introduction, identifying rules based on patterns embedded in the historical data can serve to improve business decisions. Of course, rules may sometimes be "obvious" or "common-sense", that is, they would be known without mining the database. For example, the fact that butter is usually bought with bread is known to every shopkeeper. In large organizations, however, rules may be more subtle and are realized only after mining the database.

Given the need for mining historical databases, we would obviously like to implement this in as efficient a manner as possible since searching for patterns can be computationally extremely expensive. Therefore, the main focus in data mining research has been on designing efficient rule discovery algorithms.

The inputs to the rule discovery problem are $\mathcal{I}$, a set of items, and $\mathcal{D}$, a database that stores transactional information about these items. In addition, the user provides the values for sup_{min}, the minimum level of support that a rule must have to be considered significant by the user, and con_{min}, the minimum level of confidence that a rule must have in order to be useful. Within this framework, the rule mining problem can be decomposed into two sub-problems:

Frequent Itemset Generation
Find all combinations of items that have a support factor of at least sup_{min}. These combinations are called *frequent itemsets*, whereas all other combinations are called *rare itemsets* (since they occur too infrequently to be of interest to the user).

Strong Rule Derivation
Use the frequent itemsets to generate rules that have the requisite strength, that is, their confidence factor is at least con_{min}.

In the following two sections, techniques for solving each of the above sub-problems are presented.

1.4 Frequent Itemset Generation

A simple and straightforward method for generating frequent itemsets is to make a *single pass* through the entire database, and in the process measure the support for every itemset resident in the database. Implementing this solution requires setting up of a measurement counter for each subset of the set of items $\mathcal{I}$ that occurs in the database, and in the worst case, when every subset is represented, the total number of counters required is 2^M, where M is the number of items in $\mathcal{I}$. Since M is typically of the order of a few hundreds or thousands, the number of counters required far exceeds the capabilities of present-day computing systems. Therefore, the "one-pass" solution is clearly infeasible, and several "multi-pass" solutions have therefore been developed.

A simple multi-pass solution works as follows: The algorithm makes multiple passes over the database and in each pass, the support for only certain specific itemsets is measured. These itemsets are called *candidate itemsets*. At the end of a pass, the support for each of the candidate itemsets associated with that pass is evaluated and compared with sup_{min} (the minimum support) to determine whether the itemset is frequent or rare.

Candidate itemsets are identified using the following scheme: Assume that the set of frequent itemsets found at the end of the kth pass is F_k. Then, in the next pass, the candidate itemsets are comprised of all itemsets that are constructed as *one-extensions* of itemsets present in F_k. A one-extension of an itemset is the itemset extended by exactly one item. For example, given a set of items A, B, C, D and E, the one-extensions of the itemset AB are ABC, ABD and ABE. While this scheme of generating candidate itemsets works in general, in order to start off the process we need to prespecify the candidate itemsets for the very first pass ($k = 1$). This is done by making each individual item in $\mathcal{I}$ to be a candidate itemset for the first pass.

The basic idea in the above procedure is simply that "If a particular itemset is found to be rare, then all its extensions are also guaranteed to be rare". This is because the support for an extension of an itemset cannot be more than the support for the itemset itself. So, if AB is found to be rare, there is no need to measure the support of ABC, ABD, $ABCD$, etc., since they are certain to be also rare. Therefore, in each pass, the search space is *pruned* to measure only those itemsets that are potentially frequent and the rare itemsets are discarded from consideration.

Another feature of the above procedure is that in the kth pass over the database, only itemsets that contain exactly k items are measured, due to the one-extension approach. This means that no more than M passes are required to identify all the frequent itemsets resident in the historical database.

1.5 Strong Rule Derivation

In the previous section, we described methods for generating frequent itemsets. We now move on to the second sub-problem, namely that of deriving strong rules from the frequent itemsets. The rule derivation problem can be solved using the following simple method: For every frequent itemset F, enumerate all the subsets of F. For every such subset f, output a rule of the form $f \Longrightarrow (F - f)$ if the rule is sufficiently strong. The strength is easily determined by computing the ratio of the support factor of F to that of the support factor of f. If this value is at least con_{min}, the minimum rule confidence factor, the rule is considered to be strong and is displayed to the user, otherwise it is discarded.

In the above procedure, the only part that is potentially difficult is the enumeration of all the subsets of each frequent itemset.

1.6 Classification and Sequence Rules

The rules that we have discussed so far are called *association rules* since they involve finding associations between sets of items. However, they are only one example of the types of rules that may be of interest to an organization. Other interesting rule classes that have been identified in the literature are *classification rules* and *sequence rules*, and data mining algorithms for discovering these types of rules have also been developed in the last few years.

The classification problem involves finding rules that *partition* the data into disjoint groups. For example, the courses offered by a college may be categorized into good, average and bad based on the number of students who attend each course. Assume that the attendance in a course is primarily based on the qualities of the teacher. Also assume that the college has maintained a database about the attributes of all its teachers. With this data and the course attendance information, a profile of the attributes of successful teachers can be developed. Then, this profile can be used by the college for short-listing the set of good candidate teachers whenever new courses are to be offered. For example, the rule could be "If a candidate has a master's degree, is less than 40 years old, and has more than 5 years experience, then the candidate is expected to be a good teacher".

Organizations quite often have to deal with *ordered* data, that is, data that is sorted on some dimension, usually time. Currency exchange rates and stock share prices are examples of this kind of data. Rules derived from ordered data are called sequence rules. An example rule of this type is "When the value of the US dollar goes up on two consecutive days and the British pound remains stable during this period, the Indian rupee goes up the next day 75 percent of the time".

1.7 Summary

The goal of Database Mining is to discover information from historical organizational databases that can be used to improve their business decisions. Developing efficient algorithms for mining has become an active area of research in the database community in the last few years, including research prototypes such as QUEST, constructed at IBM's Almaden Research Center in San Jose, California, U.S.A., and DISCOVER, available from the Hong Kong University of Science and Technology. We expect that sophisticated database mining packages will be available soon and that they will become essential software for all organizations within a few years.

In this chapter, we gave a broad overview of the mining problem. We move on, in the next chapter, to specifically focusing on the special requirements when mining is applied to manufacturing data.

Chapter 2

Mining in the Manufacturing Context

The problem addressed by the MINTO project is to extend database mining technology, which was discussed in the previous chapter, to the manufacturing domain. In particular, we wish to develop a mining software tool that is customized for processing manufacturing data. The tool must be capable of efficiently representing and processing huge quantities of complex data. Its performance should be able to scale with the number of resources provided and with the accuracy levels specified by the user. The emphasis will be on advanced data modeling and use of techniques such as sampling and parallel computation to support scalability and speedup.

Specifically, we wish to provide the following features in the MINTO package:

Support for complex data and associations: The manufacturing domain is characterized by rich semantic relationships such as "part-of" (containment), "is-a" (inheritance), etc. For example, a manufacturing cell contains machines and transporters, both cranes and trolleys are specializations of manual transporters, etc. In addition, quite often the associations among *attributes* of objects, rather than the objects themselves are interesting to the user. For example, the association pattern may exist that those workpieces which pass through lathe work cells typically also are sent through a quality control test due to customer requirements. Identifying this pattern will indicate to the plant manager that it would be beneficial, with respect to job transport time, to locate the quality control unit near the the lathe work cells.

Support for incremental mining: In many business organizations, the historical database is dynamic in that it is periodically updated with fresh data. For such environments, data mining is not a one-time operation but a *recurring* activity, especially if the database has been significantly updated since the previous mining exercise. Repeated mining may also be required in order to evaluate the effects of business strategies that have been implemented based on the results of the previous mining. In an overall sense, mining is essentially an exploratory activity and therefore, by its very nature, operates as a feedback process wherein each new mining is guided by the results of the previous mining.

In the above context, it is attractive to consider the possibility of using the results of the previous mining operations to minimize the amount of work done during each new mining operation. This is especially required in the manufacturing context since they have data warehouses that are constantly updated with fresh data.

Use of sampling techniques: The data generated in the manufacturing domain is enormous. For example, if there are 1000 sources generating sensor information at the rate of 10 bytes per second, the amount of data generated in a single day itself is of the order of 1 Gigabyte. Mining techniques typically require several passes over the data in order to generate patterns. Even for medium sized plants this will become computationally very expensive. Therefore, we wish to investigate the use of *sampling techniques* to improve performance. The advantage of sampling is that inferences about an entire population can be made based on characteristics exhibited by a representative subset of the population. This is achieved, however, at some cost in the accuracy of the results. We wish to quantify the performance versus accuracy tradeoff explicitly, and thereby determine the sample sizes needed to derive a desired level of accuracy.

Use of Parallel Algorithms: Apart from sampling, an alternative method of achieving acceptable performance for large manufacturing databases is to use parallel computers. For this we need to design parallel mining algorithms. In particular, we desire to design parallel algorithms for the count support phase of data mining. If each processor can hold all the candidate itemset, parallelization is straightforward with

partitioning the given transaction database among the processors. However for large scale transaction data sets, this is not true. We would therefore like to partition the candidate itemsets over the memory space of all the processors. The research problem here is to design partitioning algorithms such that communication costs among the processors are minimized. A complementary objective is to design the partitioning algorithm such that load balancing among the processors is achieved.

Graphical User Interface: The popularity of a software tool is determined to a large extent by the clarity and useability of the user interface. We therefore intend to develop a graphical user interface that facilitates usage of the tool. Users will be given a variety of means to fine-tune and guide the mining process via the graphical interface. These means include selecting and preprocessing of database tables, varying the expressive power of decision functions, declaration of types and variables which may be used to represent decision functions, etc.

In summary, we wish to develop a comprehensive database mining software tool for the manufacturing industry.

2.1 Current Status of R & D

The area of data mining is relatively new with serious research having commenced only in the last five years. In fact, the first major publication in this area appeared in 1993. Since then, data mining has become a hot topic of research in the database community and this year, there is even an international conference that is exclusively devoted to database mining. Further, according to a recent Gartner Group advanced technology research note, data mining is one of the top five key technology areas of the next three years and that data mining is one of the top ten technologies in which companies will invest during the next five years. Therefore, the importance of this topic has been fully realized.

As yet, however, data mining research has focussed primarily on mining commercial databases, and it is only now that researchers have started paying attention to supporting more complex applications such as manufacturing databases. To the best of our knowledge, there is currently no work in progress in India on data mining in general, and data mining for manufacturing in particular. The need for it has however been recognized (for example, see Express Computer, July 29, 1996 issue).

With the recent advent of liberalization, the Indian manufacturing industry has had to come to grips with the highly competitive global marketplace. In this situation, there is an urgent need to develop software tools that can be quickly leveraged by companies in their manufacturing processes to enable them to compete more effectively.

One of the most attractive areas for improving manufacturing processes is to mine the vast amount of historical manufacturing data that is present in the country. We feel that mining this data would result in significantly optimizing plant layouts and activities. Unfortunately, however, no tools currently exist for efficiently achieving this goal. We hope that our project would serve to lead the way and provide Indian manufacturing units with a useful package that would enable them to compete more effectively in the global marketplace.

2.2 Mining Model

We now describe the mining model that we have used in our project. Each transaction is represented as a tuple in the database $t =$ (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 $0 \leq confidence \leq 1$.

Often, only a few types of rules may be of interest, and the aim is to find out only those interesting rules. These interesting rules might have constraints of the following types:

Syntactic Constraints: Syntactic constraints specify all the itemsets in the rule and query the database for the associated support and confidence levels. Alternatively, we may want to find all rules which have a set $\mathcal{X}$ as the antecedent and $\mathcal{Y}$ as the consequent, along with other items.

Support Constraints: One might be interested in only those rules 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.

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

2.2.1 Solution Strategy

The problem of (association rule) 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 each large itemset $\mathcal{X}$, generate all rules of the form $\mathcal{Y} \Longrightarrow \mathcal{X} - \mathcal{Y}$, where $\mathcal{Y} \subset \mathcal{X}$, and compute its support and confidence parameters. The support of the rule is the same as that of the associated itemset while its confidence is computed[1] as $\frac{Support\ for\ \mathcal{X}}{Support\ for\ \mathcal{Y}}$. Output all rules whose support and confidence exceed sup_{min} and con_{min}, respectively.

The solution to the second problem, identifying all subset pairs of the large itemsets is relatively easy compared to the identification of large itemsets themselves. So, the research focus has largely been on developing better and faster algorithms for the identification problem.

2.2.2 Generalized and Quantitative Association Rules

The above type of association rules are simple in that all values are boolean (Y or N) and all items are semantically unrelated to each other. There are, however, often situations, especially in the manufacturing domain, where there is an *is-a classification hierarchy* over the set of items in the database – for example, a sensor may be a production monitor or an environmental monitor. In this case, users may be interested in generating association rules that *span different levels of the classification hierarchy*, since sometimes more interesting rules can be derived by taking the hierarchy into account which otherwise could not be found out. Such rules which operate over a hierarchy are called *generalized association rules* [62].

Similarly, often the values of attributes won't be simply boolean but may have quantitative features – for example, the amount of power consumption by a particular manufacturing process. Rules that operate on such attributes are called *quantitative association rules* [63]. The solution method for finding such rules is to fine-partition the values of the attribute domain and then combine adjacent partitions as necessary.

2.2.3 Report Organization

In the remainder of this report, we discuss the design and implementation of the MINTO software mining tool. MINTO features a variety of mining algorithms catering to simple association rules, generalized association rules, quantitative association rules, parallel mining, sampling-based mining, incremental mining and vertical mining. Some of these algorithms are taken from the literature while the others are new algorithms that we have designed as part of this project.

[1] If $\mathcal{X}$ is large, then any subset $\mathcal{Y}$ of $\mathcal{X}$ has to be also large [1], and hence its support will be known.

Chapter 3

First-Time Data Mining Algorithms

This chapter describes the classical first-time mining algorithms for association rules implemented as part of the MINTO tool. Apart from these algorithms, MINTO also implements a variety of incremental and vertical mining algorithms which are described in later chapters.

3.1 The AIS Algorithm

AIS [1] is among the first data mining algorithms which lends itself to simple implementation. The pseudo code for the algorithm and a detailed description of AIS are given in this section.

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 the frontier set contains a single element, the null set, whereas the candidate set starts off empty at the beginning of reach pass.

AIS is a multiple pass algorithm. The algorithm terminates when the frontier set becomes null. The number of passes depends 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 $f \subset X$ 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 may 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 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*. The question now is 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.

```
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

```
L1 = large 1-item-sets;
for ( k=2; Lk − 1 ≠ 0; k++ ) do begin
    Ck = apriori-gen ( Lk − 1 );            // New Candidate
for all transactions t ∈ D do begin
    Ct = subset (Ck, t );                   // Candidates contained in t
        for all candidates c ∈ Ct do
            c.count++;
    end
    Lk = { c ∈ Ck | c.count ≥ minsup }
end
Answer = ⋃k Lk;
```

Figure 3.4: Algorithm APRIORI

3.2 The Apriori Algorithm

The APRIORI [2] is a k pass algorithm. This algorithm introduced the concept of calculating all large i-itemsets in the ith pass of the database.

The number of candidate sets is much smaller when compared to the algorithm AIS. Moreover, in 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.

The pseudo code for APRIORI is given in Figure 3.4 and Figure 3.5. 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.

```
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

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.

3.2.1 Data Structures

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 are the 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 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 [2] is similar to the APRIORI algorithm in that it 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.

The first pass of AprioriTid is limited to finding out 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 is reduced in each successive scan of the database. In the kth pass of the algorithm all those transactions that do not have any of the candidate (k+1) 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.

3.3.1 Data Structures

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.

```
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

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) [82] 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.

The pseudo code for the algorithm is given in Figure 3.7. 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 3.4.1. 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 of DHP 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.

3.4.1 Reduction in Database Size

DHP reduces the size of the database progressively by trimming each individual transaction size and pruning the number of transactions in the database. This size reduction is implemented in the following way: 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 that do 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*.

/* Part 1 */

s = a minimum support;
set all the buckets of H_2 to zero; /* hash table */
for all transaction $t \in D$ **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
$L_1 = \{c \mid c.count \geq s$, c is in a leaf node of the hash tree $\}$;

/* Part 2 */

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

}

/* Part 3 */

gen_candidate (L_{k-1}, H_k, C_k);
while ($| C_k | > 0$) {
  $D_{k+1} = \phi$;
  for all transactions $t \in D_k$ **do begin**
    count_support (t, $C_k, k, \hat{t}$); /* $\hat{t} \subseteq t$ */
    if ($| \hat{t} | > k$) **then** $D_{k+1} = D_{k+1} \cup \{t\}$;
  end
  $L_k = \{c \in C_k \mid c.count \geq s\}$;
  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)
  $C_k = \phi$;
  for all $c = c_p[1] \cdots c_p[k-2] \cdot c_p[k-1] \cdot c_q[k-1]$,
  $c_p, c_q \in | c_p \cap c_q | = k-2$ **do**
    if ($H_k[h_k(c)] \geq s$) **then**
      $C_k = C_k \cup \{c\}$; /* insert c into a hash tree */
end Procedure

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

3.5 The Partition Algorithm

Unlike the previous algorithms, which are k pass, the PARTITION algorithm [15] is a *two* pass algorithm. It uses the divide and conquer 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 processes is completely independent of the others, this can be done in parallel. Once the local large itemsets are 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.

In Partition, 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 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 the set of tid's of 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. Therefore, only one scan of the database is enough to find out the local large itemsets.

An itemset can 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 all 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.

A practical difficulty with the Partition algorithm is to select the proper partition size. If there are too many partitions, then there might be more number of candidate sets and the final pass will become costly. On the other hand, if the number of partitions is very less, number of elements in each partition will be so large that all the calculations and results may not fit 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 partition size to

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; $C_k^G \neq \phi$; k++) **do begin**
    for all k-itemset $c \in C_k^G$ **do begin**
      templist = $c[1].tidlist \cap c[2].tidlist \cap \cdots \cap c[k].tidlist$
        c.count = c.count + | $templist$ |
    end
  end
end

Figure 3.14: Final Count Generation

```
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 3.15: Algorithm SAMPLING

get the maximum performance by this method.

3.6 Speedup through Sampling

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 partitions 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.

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 in 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 that reflects 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 [48, 66], 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 [66] 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.

The sampling-based mining algorithm implemented in MINTO is due to H. Toivonen [66]. 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.

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 [66]. But if one relies on results from sampling alone, there is a risk of losing valid association rules because of the fact that the frequency of the subset in the sample is below the threshold.

The algorithm operates in 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 using this technique is given in Figure 3.15. 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.

3.7 The Cumulate Algorithm

The Cumulate algorithm [62] has been designed for mining *generalized* association rules. This algorithm is similar in structure to Apriori for boolean association rules, the differences being:

- **Tuple augmentation.** During the scanning of the database, all the ancestors of each item in a tuple are also added to the tuple.
- **Filtering the ancestors added to transactions.** Only ancestors of items in the transaction that are also present in some candidate itemset are added.
- **Pre-computing ancestors.** Rather than finding ancestors for each item by traversing a taxonomy graph, the ancestors for each item are precomputed.
- **Pruning itemsets containing an item and its ancestor.** An itemset that contains both an item and its ancestor may be pruned as it will have the same support as the itemset which doesn't contain that ancestor, hence being redundant. This pruning needs to be done explicitly only once which is before the second pass over the database.

3.8 The QAR Algorithm

In generating quantitative association rules, there are two main problems [63]:

MinSup If the number of intervals for a quantitative attribute is large, the support for any single interval can be low. Hence, without using larger intervals, some rules involving this attribute may not be found because they lack minimum support.

MinConf There is some information loss whenever values are partitioned into intervals. Some rules may have minimum confidence only when an item in the antecedent consists of a single value. This information loss becomes larger as the interval sizes become larger.

The above situation is problematic in that if the intervals are too large, some rules may not have minimum confidence, whereas if they are too small, some rules may not have minimum support. To address this problem, the QAR algorithm was proposed in [63]. Here, ranges over adjacent intervals of quantitative attributes are considered and the extent to which these adjacent intervals are combined is determined by introducing a user-specified "maximum support" parameter. To help decide the number of partitions for each quantitative attribute, a measure of partial completeness that quantifies the information lost due to partitioning is introduced and this measure is used to decide the number of partitions.

A direct application of the above technique may generate too many similar rules. This problem is addressed by using a "greater-than-expected-value" interest measure to identify the interesting rules in the output. This interest measure looks at both generalizations and specializations of the rule to identify the interesting rules.

3.9 The CountDistribute Parallel Algorithm

All the above-mentioned algorithms are meant for mining on single processors. We now describe an algorithm for *parallel* mining. In the Count Distribute algorithm [27], which is a parallelization of the Apriori algorithm, all processors generate the entire candidate hash tree from L_{k-1}. Each processor can thus independently get partial support of the candidates from its local database partition. This is followed by a sum-reduction to obtain the global counts. Note that only the partial counts need to be communicated, rather than merging different hash trees, since each processor has a copy of the entire tree. Once the global L_k has been determined each processor builds C_{k+1} in parallel, and repeats the process until all frequent itemsets are found. This simple algorithm minimizes communication since only the counts are exchanged among the processors. However, since the entire hash tree is replicated, there is a drawback that the aggregate memory is not used very efficiently.

Chapter 4

The TWOPASS Algorithm

As discussed in the previous chapter, there have been been many algorithms proposed in the literature for efficiently mining (for the first time) large historical databases. Except for the Partition algorithm, the other algorithms require making multiple scans over the entire database with the number of scans required being proportional to the number of items in the biggest frequent itemset. In the remainder of this chapter, we present **TWOPASS**, a new algorithm that is guaranteed to provide all frequent itemsets in exactly **two** passes over the database without attracting the problems associated with the Partition algorithm.

4.1 Algorithmic Details

In our scheme, the database is logically divided into partitions, not necessarily of equal size, with the restriction being that each partition should completely fit into main memory. A hashtree data-structure called L is used to store itemsets. Along with each itemset we also maintain two integer fields – a *count* field and a *partition* field. Itemsets are brought into and removed from L dynamically, at each partition. The count field contains the number of occurrences of the itemset in the partitions over which it was counted. If an itemset was brought into L during the nth partition, then its partition field is n. The number of tuples over which such an itemset has been counted so far is denoted by $tuples(n)$. This could be a simple function or a lookup table that is updated after each partition is processed. At any time, the *partial_support* of an itemset is the ratio of its *count* field to *tuples*(*partition* field).

Each partition is read one by one. For each partition, we need to find all its local frequent itemsets. This can be done efficiently as the partition is in main memory. In our implementation, we used the *Apriori* [2] algorithm to find the frequent itemsets in the first partition. For the remaining partitions, we use L, and its negative border N, as a set of candidate itemsets. The counts in N are set to zero. If an itemset in N becomes frequent, then a *miss* has occurred. The negative border closure of $L \cup N$ is generated and its counts over the partition are found. The partition fields of the new itemsets are set to the current partition. If the partial_support of any itemset in L becomes less than sup_{min}, it is removed from L. At the end of the partition, L will contain (atleast) all the local frequent itemsets and N will contain its negative border. Thus we don't need to calculate the negative border of L again for the next partition.

That ends the first pass over the database. L now contains all potentially frequent itemsets. The hashtree N is no longer needed and is discarded.

In the second pass, each partition is again read one by one. Prior to each partition P_n, all itemsets in L whose partition field is n, are output as frequent itemsets, and removed from L. If there are no more itemsets in L, the algorithm halts. Otherwise, the count fields of the remaining itemsets in L are updated over P_n. If the partial_support of any itemset becomes less than sup_{min}, it is removed from L. No new itemsets are added to L during the second pass. By the end of this pass, all frequent itemsets will be output.

In Figure 4.1, we show the generic algorithm, and in Figure 4.2, we show the detailed steps of the TWOPASS

algorithm.

```
1.  L = ∅
2.  For each partition P_n do /* first pass */
3.      Read in the partition to main memory.
4.      Update count fields of L over P_n.
5.      Remove itemsets with partial_support < sup_min from L.
6.      Find local frequent itemsets that are not in L and add them to L.
7.      Set the partition fields of the new itemsets to n.
8.  For each partition P_n do /* second pass */
9.      Output itemsets in L for which partition field is n, and remove them from L.
10.     If L is empty, halt.
11.     Read in P_n to main memory.
12.     Update count fields of L over P_n.
13.     Remove itemsets with partial_support < sup_min from L.
```

Figure 4.1: The Generic Algorithm

```
1.  Read in the 1st partition P_1 to main memory.
2.  Perform Apriori over P_1 to obtain local frequent itemsets L
    and its negative border N.
3.  Set partition fields of itemsets in L to 1.
4.  For each remaining partition P_n do  /* first pass */
5.      Read in the partition to main memory.
6.      Set the count fields of N to zero and their partition fields to n.
7.      Update count fields of L and N over P_n.
8.      Remove itemsets with partial_support < sup_min from L.
9.      M = itemsets of N whose partial_support ≥ sup_min.
10.     L = L ∪ M
11.     C = Negative_Border_Closure( L ∪ N )
12.     Find counts of C over P_n.
13.     L = L ∪ (itemsets of C whose partial_support ≥ sup_min
14.     N = N ∪ (itemsets of C all of whose subsets are in L)
15. Discard N.
16. For each partition P_n do /* second pass */
17.     Output itemsets in L for which partition field is n, and remove them from L.
18.     If L is empty halt.
19.     Read in P_n to main memory.
20.     Update count fields of L over P_n.
21.     Remove itemsets with partial_support < sup_min from L.
```

Figure 4.2: The TWOPASS Algorithm

4.2 Proof of Correctness

We now prove that the algorithm described above correctly generates all the frequent itemsets.

Theorem 1: During the first pass, all itemsets that are frequent within a partition are present in L at the end of that partition.
Proof:

Let $count(i,n)$, $region(i,n)$ and $partial_support(i,n)$ denote the count field, region field and partial_support of itemset i respectively, at the end of handling the nth partition during the first pass. *local_count(i,n)* is the number of occurrences of itemset i within the nth partition and $part_size(n)$ is the size of the nth partition.

If an itemset i is frequent within the nth partition, then either it was present in L at the beginning of the partition or not. If it was present, then,

$$partial_support(i,n\text{-}1) \geq sup_{min} \Leftrightarrow count(i,n\text{-}1) \times sup_{min} \times region(i,n\text{-}1)$$

Since i is frequent in this partition,

$$local_count(i,n) \geq sup_{min} \times part_size(n)$$

Therefore,

$$\begin{aligned} count(i,n) &\geq sup_{min} \times (\ region(i,n\text{-}1) + part_size(n)\ ) \\ &\geq sup_{min} \times region(i,n) \end{aligned}$$

Hence i will be present in L at the end of the partition as well.

If i was not in L at the beginning of the partition, then that would be indicated by a potential miss. It would be present in N or would be generated in the closure step. In either case, it will be counted over the partition. Since its support in the partition exceeds sup_{min}, it will be moved to L. Hence proved.

Theorem 2: During the first pass, let i be an itemset that is not present in (or is removed from) L at the end of some partition P_v. Let R be the region spanning partitions P_u to P_v, for any $u \leq v$. Then i cannot be locally frequent within R.

Proof: It is true when $u = v$, by the converse to Theorem 1. Let it be true when $v - u \leq n$. Then we show it to be true when $v - u = n + 1$.

By the inductive hypothesis, i is not locally frequent within the region spanning partitions P_{u+1} to P_v. If i is not locally frequent in P_u also, then it cannot be frequent within R, hence proving the theorem.

If i is frequent in P_u, then by Theorem 1, it must be present in L after P_u. An itemset is removed from L only if its *partial_support* is less than sup_{min}. Since i is not in L after partition P_v, either it was removed from L only at the end of partition P_v, or it was removed before that. If it was removed from L only at the end of partition P_v, then it can't be locally frequent in R and the theorem holds true.

If i is removed from L at the end of some partition P_w, $w < v$. Then i cannot be frequent in the region of R prior to P_w. If it is, then it wouldn't be removed from L. By the inductive hypothesis, it cannot be frequent in the region spanning partitions P_w to P_v. Thus i cannot be frequent in the entire region. Hence proved.

Corollary: L contains all potentially frequent itemsets at the end of the first pass.

Theorem 3: If an itemset i is removed from L in the second pass, and is not output, then it cannot be frequent.

Proof: Let K be the total number of partitions in the database. The partition field of i cannot be 1, since all such itemsets are output at the start of the second pass. Let its partition field be n. Let i be removed from L after the mth partition during the second pass. Then its local support in the region spanning partitions P_n to P_K and P_1 to P_m must be less than sup_{min}. By Theorem 1, its local support in the region spanning partitions P_{m+1} to P_{n-1} is less than sup_{min}. Since these regions span the entire database, i can't be globally frequent. Hence proved.

4.3 Negative Border Closure

The description of how to efficiently compute the negative border closure is given in the next chapter on incremental mining – please refer there for the details.

4.4 Size of Partitions

One of the design issues for the TWOPASS algorithm is the number of partitions. However, the performance is not sensitive to the size of partitions except for extreme cases. Very small partitions may result in too many candidates from each partition which are not globally frequent. But these candidates will be removed from L after a few more partitions are traversed. Hence the algorithm is *relatively free from skew* as compared to the Partition algorithm [15] which attempted to use the partition idea for *parallel* mining. Very large partitions don't cause any performance degradation, provided they can fit in main memory. Note that the size of a partition must be atleast 1 / sup_{min} since otherwise there will not be enough tuples in each partition for the definition of sup_{min} to hold. A reasonable partition size would be about ten to twenty times this minimum size. However some performance improvement may be noticeable with very large partition sizes.

4.5 Number of Candidate Itemsets

It is easy to see that an itemset will be in L only if it is frequent in atleast one partition. Hence the number of candidates which are generated is not more than that in Partition [15]. However since itemsets are continually removed from L, the number of candidates is actually much less. Another advantage over Partition is due to the way we mine each partition. Here, the information gathered from previous partitions is used to reduce processing during mining each partition. Therefore, no tid-lists are maintained and no level-wise mining of individual partitions is necessary.

4.6 Incorporating Sampling

Instead of performing the Apriori algorithm over the first partition, we could perform *sampling* to obtain the candidate set L. This would be desirable if the cost of sampling is low. To benefit from sampling, the following change needs to be incorporated in the algorithm –

Each itemset has a *pinned* flag associated with it. If an itemset is pinned then it is not removed from L even if its partial_support decreases below sup_{min}. The counts of pinned itemsets in N are not reset to zero at the start of each partition. All the itemsets in L and N at the end of sampling must be pinned. If the sample is accurate, then more than one pass will not be necessary. However, in the event that a second pass is necessary, itemsets in L whose partial support decreases below sup_{min} should be not removed from L during the second pass. The proof of Theorem 3 will not hold any longer.

Sampling cost is not always low. Normally, the sample size required will be much larger than the size of the first partition. Either a database pass or random access is required for sampling – in the latter case, the database cannot be dynamically read from tape. Also, since we cannot remove itemsets from L during the second pass, almost the complete second pass will be necessary. Without sampling, it is likely that all itemsets will be removed from L early in the second pass. The only additional advantage of sampling is that approximate rules can be generated from the sample itself. If this is desired then sampling may be the option of choice.

Chapter 5

Incremental Mining

In many business organizations, the historical database is dynamic in that it is periodically updated with fresh data. For such environments, data mining is not a one-time operation but a *recurring* activity, especially if the database has been significantly updated since the previous mining exercise. Repeated mining may also be required in order to evaluate the effects of business strategies that have been implemented based on the results of the previous mining. In an overall sense, mining is essentially an exploratory activity and therefore, by its very nature, operates as a feedback process wherein each new mining is guided by the results of the previous mining.

In the above context, it is attractive to consider the possibility of using the results of the previous mining operations to minimize the amount of work done during each new mining operation. That is, given a previously mined database DB and a subsequent increment db to this database, to efficiently mine $DB \cup db$. Such "incremental" mining is the focus of this chapter. Practical applications where incremental mining techniques would be especially useful include manufacturing data warehouses since they are constantly updated with fresh data [85], and web mining. On the web, for instance, about one million pages are being added daily to the existing 300 million pages [80].

We consider here the design of incremental mining algorithms for identification of association rules in "market basket" databases [1], including "basic" as well as "generalized" association rules [62]. Generalized rules are applicable to environments where there is an *is-a hierarchy* over the set of items in the database, while basic rules are a special case of generalized rules wherein the hierarchy is flat – all items are leaves.

The design of incremental mining algorithms for basic association rules (BAR) has been considered earlier in [73, 77, 79, 10]. While these studies were a welcome first step in addressing the problem of incremental mining, they also suffer from a variety of limitations. For example, the effect of temporal changes (i.e. skew) in the distribution of database values between DB and db was not considered – as we will show in this chapter, the performance of the algorithms presented in [79, 10] is sensitive to this factor. In fact, their sensitivity is to the extent that, with significant skew and substantial increments, they may do worse than even the naive approach of ignoring the previous mining results and applying the classical Apriori algorithm [2] from scratch on the entire current database. Another limitation in most of the previous performance studies is that only small databases with small increment sizes (relative to the available main memory) were considered.

Similarly, an implicit assumption in almost all the previous algorithms is that the minimum support specified by the user for the current database ($DB \cup db$) is the *same* as that used for the previously mined database (DB). However, in practice, we would expect that user requirements would change with time, resulting in different minimum support levels. Extending the earlier incremental algorithms to efficiently handle such "multi-support" environments is not straightforward. Finally, apart from comparing their performance with that of Apriori, no quantitative assessment was made of the *efficiency* of these algorithms in terms of their distance from the optimal, which would be indicative of the amount of scope if any for further improvement.

All the previous algorithms cited above focus on incremental BAR-mining. No work has so far been done regarding the incremental mining of generalized association rules (GAR). As pointed out in [62], taxonomies are valuable since rules at lower levels may not satisfy the minimum support threshold and hence not many rules are likely to be output if they are not considered. They are also useful to prune away many of the uninteresting rules that would be output at low minimum support thresholds, since many rules at lower levels may be subsumed by those at higher levels.

DB, db, $DB \cup db$	Previous, increment, and current database
I	Set of all items in the database
sup_{min}^{DB}, $sup_{min}^{DB \cup db}$	Previous and New Minimum Support Thresholds
sup_{min}	Minimum Support Threshold when $sup_{min}^{DB} = sup_{min}^{DB \cup db}$
L^{DB}, L^{db}, $L^{DB \cup db}$	Set of large itemsets in DB, db and $DB \cup db$
N^{DB}, N^{db}, $N^{DB \cup db}$	Negative borders of L^{DB}, L^{db} and $L^{DB \cup db}$
L	Set of itemsets in $L^{DB \cup db}$ with known counts (during algorithm execution)
$Small$	Set of itemsets with known counts which are not in L

Table 5.1: **Notation**

Contributions

In this chapter, we present a new incremental mining algorithm called **DELTA** (Differential Evaluation of Large iTemset Algorithm) for the incremental mining of both basic and generalized association rules. While the basic structure of DELTA is similar to that of the algorithms presented in [79, 10], it incorporates a variety of new techniques to address their above-mentioned limitations. For example, a much higher degree of candidate itemset pruning is effected by utilizing negative border [84] information differently. DELTA guarantees that only *three passes* over the increment and *one pass* over the previous database are required in the worst case. For multi-support environments, at most *one* additional pass over the current database may be required to complete the incremental mining process. Finally, for the special case where the new results are a *subset* of the old results, and therefore in principle requiring no processing over the previous database, DELTA is optimal in that it requires only a single pass over the increment to complete the mining process.

Using a synthetic database generator, we analyze the performance characteristics of DELTA on a variety of dynamic databases and compare it with that of Apriori and the previously proposed incremental mining algorithms for basic association rules. For generalized association rules we compare against the Cumulate first-time mining algorithm presented in [62].

A novel feature of our study is that we include in our evaluation suite the performance of an *an oracle* which has *complete apriori* knowledge of the identities of all the large itemsets (and their associated negative border) in the current database and only requires to find their counts. Modeling this performance permits us to characterize the efficiency of practical algorithms in terms of their distance from the optimal. We refer to this algorithm as **ORACLE** in this chapter.

Two kinds of dynamic databases are evaluated in our experiments: *Identical* and *Skewed*. In Identical, the increment db has the same data distribution as that of the previous database DB, whereas in Skewed, there is significant change between the large itemsets of DB and those of db. For each of these workloads, we consider a variety of increment sizes that range from where the increment is a small fraction of the previous database to increments that are of the same size as the previous database. Further, we consider both equi-support and multi-support environments.

The results of our experiments show that DELTA can provide significant improvements in execution times over the previously proposed algorithms in all these environments. In fact, DELTA's efficiency is *close to that obtained by ORACLE* for most of the workloads that were considered in our study. This shows that DELTA is able to extract *most of the potential* for using the previous results in the incremental mining process.

5.1 Incremental Mining

In this section, we overview the incremental approach to mining market basket databases for association rules. The necessary theoretical results on which DELTA and other incremental mining algorithms are based are introduced. For ease of exposition, we use the notation given in Table 5.1 in the following discussion and in the remainder of this chapter.

The input to the incremental mining process is the set of large itemsets L^{DB}, its corresponding negative border N^{DB}, and their associated supports. The output is the updated versions of the inputs, namely, $L^{DB \cup db}$ and $N^{DB \cup db}$, along with their supports. For this system, the following basic observation was identified in [77]:

Theorem 5.1.1 *An itemset can be present in $L^{DB \cup db}$ only if it is present in either L^{DB} or L^{db} (or both).*

From the above theorem, it is straightforward to deduce that the only itemsets for which it is necessary to scan the previous database DB in order to derive their supports are those that are large in db but not in DB,

that is, in $L^{db} - L^{DB}$. This observation can be further augmented by utilizing the negative border information, as proposed in [79, 10]:

Theorem 5.1.2 *If X is an itemset that is not in L^{DB} but is in $L^{DB \cup db}$, then there must be some subset x of X which was in N^{DB} and is now in $L^{DB \cup db}$.*

Itemsets which were in N^{DB} but have now moved to $L^{DB \cup db}$ due to the increment are referred to as *promoted borders* [79]. From Theorem 5.1.2, it follows that itemsets which could potentially be part of the output ($L^{DB \cup db} \cup N^{DB \cup db}$) and whose counts over $DB \cup db$ are currently unknown *must all be extensions of the promoted borders.*

The incremental mining algorithms proposed in the literature are based on the above observations, as described in the following section.

5.2 Previous Incremental Algorithms

In this section, we provide an overview of the algorithms that have been developed over the last two years for incremental mining of basic association rules on market basket databases.

5.2.1 The FUP Algorithm

The **FUP** (Fast UPdate) algorithm [77] represents the first work in the area of incremental mining. It operates on an iterative basis and in each iteration makes a *complete scan of the current database.* In each scan, the *increment is processed first* and the results obtained are used to guide the mining of the original database DB. An important point to note about the FUP algorithm is that it requires k- passes over the entire database, where k is the cardinality of the longest large itemset.

In the first pass over the increment, all the 1-itemsets are considered as candidates. At the end of this pass, the complete supports of the candidates that happen to be also large in DB are known. Those which have the minimum support are retained in $L^{DB \cup db}$. Among the other candidates, only those which were large in db can become large overall due to Theorem 5.1.1 (Section 5.1). Hence they are identified and the previous database DB is scanned to obtain their overall supports, thus obtaining the set of all large 1-itemsets. The candidates for the next pass are calculated using the AprioriGen function [2], and the process repeats in this manner until all the large itemsets have been identified.

After FUP, algorithms that utilized the *negative border* information were proposed independently in [10] and [79] with the goal of achieving more efficiency in the incremental mining process. In the sequel we will use TBAR to refer to the algorithm in [10], and Borders to refer to the algorithm in [79].

5.2.2 The TBAR Algorithm

In the TBAR algorithm, first *all the large itemsets* in the increment db (i.e. L^{db}) are found by fully applying the Apriori algorithm on the increment. Simultaneously, the counts of all itemsets in $L^{DB} \cup N^{DB}$ are updated over db and, among these, the itemsets that turn out to be large are added to L. After this, if L is found to be different from L^{DB}, the new negative border N is calculated. Now, if there are itemsets in $L \cup N$ whose supports w.r.t $DB \cup db$ are still unknown, new candidates are generated by computing the *negative border closure* of L. During the computation of the closure, any itemsets that are not large in db are pruned. The counts of the remaining candidates are then obtained by making a single pass over $DB \cup db$. Thus all itemsets in $L^{DB \cup db}$ and $N^{DB \cup db}$ and their supports are finally obtained.

The TBAR approach is attractive by virtue of requiring only *one scan* over the previous database, as compared to FUP which requires k passes. However, a significant drawback is that *too many unnecessary candidates* may be generated in the negative border closure resulting in an increase in the overall mining time. Further, the increment db is initially *completely mined* – this could be inefficient for large increments since none of the previous mining results are used in this process.

5.2.3 The Borders Algorithm

The Borders algorithm is similar in design to TBAR but differs in the manner in which extensions of promoted borders are pruned. Here, an extension of length k is retained as a candidate for counting if each of its $k-1$ subsets is either: (a) also a candidate, or (b) a promoted border, or (c) is in L_{inc} (where L_{inc} refers to those itemsets in $L^{DB} \cup N^{DB}$ which have currently been determined to be in $L^{DB \cup db}$ and appear *at least once* in the

increment db). The candidates thus identified are counted by a scan over $DB \cup db$ and the overall set of large itemsets is determined along with its negative border.

The Borders algorithm differs from TBAR in that a complete initial mining of the increment db is not made. Further, the full mining process requires only *two passes* over the increment in the worst case, whereas TBAR requires k passes. However, a potential drawback of Borders is that it could generate *a very large number of candidates* if the promoted borders contain more than a few 1-itemsets. This is because if p_1 is the number of 1-itemsets in the promoted borders, a lower bound on the number of candidates is $2^{p_1}(|L_{inc}| - p_1)$. This arises out of the fact that every combination of the p_1 1-itemsets is a possible extension, and all of them can combine with any other large itemset in L_{inc} to form candidates. Therefore, even for moderate values of p_1, the number of candidates generated could be extremely large.

Another point to note is that the pruning optimization used in the candidate generation, wherein extensions of itemsets in L^{DB} which are *completely absent* in the increment are removed (clause (c) above), may not be useful in practice. This is because with increments of non-trivial sizes, we could expect that most or all of the itemsets in L^{DB} would be present *at least once* in the increment.

A new version of the Borders algorithm which attempts to overcome some of the above drawbacks was recently proposed in [73]. A problem, however, is that the new algorithm has *no fixed bound* on the number of passes made over the database. Further, a variant of the new algorithm was proposed to handle multi-support mining. The applicability of this algorithm, however, is limited to the very special case of *zero-size* increments, that is, where the database has not changed at all between the previous and the current mining. Due to these limitations, we consider only the original Borders algorithm in the remainder of this chapter – in our future work, we propose to also evaluate the new version.

5.2.4 Other Algorithms

Recently a new algorithm for first-time mining called CARMA was proposed in [81] where its applicability to incremental mining was also briefly mentioned. Although the algorithm is a novel and efficient approach to first-time mining, we note that it suffers from the following drawbacks when applied to incremental mining: (1) It does not maintain negative border information and hence will need to access the original database DB if there are any locally large itemsets in the increment, even though these itemsets may not be globally large. (2) The shrinking support intervals which CARMA maintains for candidate itemsets are not likely to be tight for itemsets that become potentially large while processing the increment. This is because the number of occurrences of such itemsets in DB will be unknown and could be as much as $sup_{min} * |DB|$.

5.3 The DELTA Algorithm

In this section, we present our new incremental algorithm, **DELTA** (Differential Evaluation of Large iTemset Algorithm), for identifying both basic and generalized association rules in basket databases. The DELTA algorithm is similar in overall structure to the TBAR and Borders algorithms, but functionally differs from these algorithms in a variety of ways:

- While DELTA does generate, like Borders, the extensions of promoted borders, it does not generate *all* such extensions at the same time. Instead, only the *immediate* extensions are first generated and counted over the increment db. By immediate extensions we mean the 1-extensions corresponding to the first layer of the negative border closure. Now, from Theorem 5.1.1, we can conclude that if an immediate extension is small over the increment, then none of its extensions need to be considered. Therefore, identifying such small immediate extensions early on helps to prune away many of the candidates that are generated by the original Borders algorithm.

- DELTA guarantees completion in the worst case in *three passes* over the increment db and *one* over DB. Overall this might seem less efficient than the original Borders algorithm which requires only two passes over the increment db (as described in Section 5.2). However, this is really not the case since the *number of candidates* that are counted during each of the DELTA passes is much fewer. That is, DELTA makes a conscious choice of increasing the number of passes over the increment to reduce the number of candidates counted in each such pass.

- While both TBAR and Borders are sensitive to skew in the temporal distribution of database contents, DELTA is comparatively robust.

- DELTA can handle multi-support environments, whereas this issue is not addressed in detail by any of the previously proposed algorithms. It requires only *one* additional pass over both the increment and the previous database to achieve this functionality.
- Unlike any of the previously proposed algorithms, DELTA can also handle generalized association rules.

5.3.1 DELTA Processing Details

DELTA $(DB, db, L^{DB}, N^{DB}, sup_{min})$
Input: Previous Database DB, Increment db, Previous Large Itemsets L^{DB},
Previous Negative Border N^{DB}, Minimum Support Threshold sup_{min}
Output: Updated Set of Large Itemsets $L^{DB \cup db}$, Updated Negative Border $N^{DB \cup db}$
begin
1. UpdateCounts$(db, L^{DB} \cup N^{DB})$; // **first pass over** db
2. L = GetLarge$(L^{DB} \cup N^{DB}, sup_{min} * |DB \cup db|)$;
3. $Small = (L^{DB} \cup N^{DB}) - L$ // used later for pruning
4. **if** $(L^{DB} == L)$
5.   **return**(L^{DB}, N^{DB});

6. N = NegBorder(L);
7. **if** $(N \subseteq Small)$
8.   get supports of itemsets in N from $Small$
9.   **return**(L, N);

10. $N = N - Small$;
11. UpdateCounts(db, N); // **second pass over** db
12. C = GetLarge$(N, sup_{min} * |db|)$;
13. $Small^{db} = N - C$ // used later for pruning
14. **if** $(|C| > 0)$
15. $C = C \cup L$
16.   ResetCounts(C);
17.   **do** // compute negative border closure
18. $C = C \cup$ NegBorder(C);
19. $C = C - (Small \cup Small^{db})$ // prune
20.   **until** C does not grow
21. $C = C - (L \cup N)$
22.   **if** $(|C| > 0)$
23.   UpdateCounts(db, C); // **third and final pass over** db

24. $ScanDB$ = GetLarge$(C \cup N, sup_{min} * |db|)$;
25. N' = NegBorder$(L \cup ScanDB) - Small$;
26. get supports of itemsets in N' from $(C \cup N)$
27. UpdateCounts$(DB, N' \cup ScanDB)$; // **first (and only) pass over** DB
28. $L^{DB \cup db} = L \cup$ GetLarge$(ScanDB, sup_{min} * |DB \cup db|)$;
29. $N^{DB \cup db}$ = NegBorder$(L^{DB \cup db})$;
30. get supports of $N^{DB \cup db}$ from $(Small \cup N')$
31. **return**$(L^{DB \cup db}, N^{DB \cup db})$;

end

Figure 5.1: **The DELTA Incremental Mining Algorithm**

We now present a detailed description of the DELTA algorithm and how it achieves the above-mentioned features and improvements. For ease of exposition, we first present the "equi-support" case, and then in Section 5.4, we present the modifications required to handle the multi-support environment. In these sections we present the algorithm as it applies to basic association rules. Then in Section 5.7, we present the modifications required to handle generalized association rules.

The pseudo-code of the (equi-support) DELTA algorithm is shown in Figure 5.1, and we explain below the steps taken in each of the passes that it makes over the increment and the previous database.

First Pass over the Increment

In the first pass, the counts of itemsets in L^{DB} and N^{DB} are updated over the increment db, using the function UpdateCounts (line 1 in Figure 5.1). By this, some itemsets in N^{DB} may become large and some itemsets in L^{DB} may become small. Let the resultant set of large itemsets be L. These large itemsets are extracted using the function GetLarge (line 2). The remaining itemsets are put in *Small* (line 3), and will be later used for pruning candidates. The algorithm terminates if no itemsets have moved from N^{DB} to L (lines 4–5). This is valid due to Theorem 5.1.2 in Section 5.1.

Second Pass over the Increment

On the other hand, if some itemsets do move from N^{DB} to L, then any extension of these itemsets may also turn out to be large. So, a second pass over the increment is made to find the counts of the immediate extensions for which counts are not already available, i.e. NegBorder(L) $-$ *Small* (line 11). We refer to this set as N (lines 6 and 10). The negative border of a set of itemsets is computed using the AprioriGen function, as in [10].

Now, by Theorem 5.1.1, any itemset in N that is not locally large in db cannot be large in $DB \cup db$. Further, none of its extensions can be large as well. We make use of these properties by storing all such itemsets in a set called $Small^{db}$ (line 13), which is later used for pruning candidates.

Third (and Final) Pass over the Increment

We then form all possible extensions of L which could be in $L^{DB\cup db} \cup N^{DB\cup db}$ and store them in set C. This is done by computing the remaining layers of the negative border closure of L (lines 15–20). (We expect that the remaining layers can be generated together since the number of 2-itemsets in L is typically much smaller than the overall number of all possible 2-itemset pairs.) At the start of this computation, the counts of itemsets in C are reset to zero using the function ResetCounts (line 16). Then at every stage during the computation of the closure, those itemsets that are in *Small* and $Small^{db}$ are removed so that none of their extensions are generated (line 19). After all the layers are generated, the first two (i.e. L and N) are removed from C since the counts of their itemsets are already available (line 21). The third and final pass over db is then made to find the counts of the remaining itemsets in C (line 23).

First (and Only) Pass over the Previous Database

Those itemsets of the closure which turn out to be locally large in db need to be counted over DB as well to establish whether they are large overall. We refer to these itemsets as *ScanDB* (line 24). Since the counts of $N^{DB\cup db}$ need to be computed as well, we evaluate NegBorder($L \cup ScanDB$). From this the itemsets in *Small* are removed since their counts are already known. The counts of the remaining itemsets (i.e. N' in line 25) are then found by making a pass over DB (line 27).

After the pass over DB, the large itemsets from *ScanDB* are gathered to form $L^{DB\cup db}$ (line 28) and then its negative border $N^{DB\cup db}$ is computed (line 29). The counts of $N^{DB\cup db}$ are obtained from *Small* and N' (line 30). Thus we obtain the final set of large itemsets $L^{DB\cup db}$ and its negative border $N^{DB\cup db}$.

5.3.2 Intuition Behind the DELTA Design

In this subsection we informally motivate as to why the DELTA algorithm was designed in the above fashion and why we expect it to perform better than the previously proposed algorithms.

The purpose of the second pass over the increment is to prune away most of the unnecessary candidates that would have been generated had we computed all possible extensions of the promoted borders at that stage. By performing the second pass, any itemsets in the first layer of the negative border closure that are not large in the increment can be excluded from further candidate generation.

The above pruning strategy can be taken to its limit by making a separate pass over the increment after computing each layer of the negative border closure. However, this could be inefficient for large increment sizes. Besides, the amount of pruning in these later passes may not be substantial since the number of large 2-itemsets (in the increment) would typically be negligible compared to the total number of candidate 2-itemsets. Moreover, if the number of known potentially large 2-itemsets is small, the AprioriGen function is known to be efficient in its pruning [82]. Hence the computed negative border closure after the second pass would not be much larger than what would be obtained by performing multiple passes. Therefore, DELTA does not follow this approach and chooses to perform only one additional pass over the increment.

Finally, DELTA performs a third pass over the increment to prune away itemsets in the computed negative border closure which cannot be in $L^{DB \cup db} \cup N^{DB \cup db}$.

In the design of the DELTA algorithm, we have thus endeavored to achieve a reasonable compromise between the number of candidates counted and the number of database passes. As these two factors represent the primary bottle-neck in most association rule generating algorithms, we expect DELTA to perform better than previous algorithms for most database workloads – this is confirmed in our experimental study described in Section 5.6.

In the remainder of this section, we discuss how DELTA handles some special situations that may arise in the course of incremental mining.

5.3.3 No Scan over DB Scenario

An interesting situation arises when $L^{DB} \cup N^{DB}$ is exactly equal to or a superset of $L^{DB \cup db} \cup N^{DB \cup db}$. In this case, note that in principle it is not necessary to make *any* passes over the previous database DB since all the required counts are already available.

In the above situation, FUP will still make passes over DB since it does not utilize negative border information (note that this would happen even in the more restrictive situation where $L^{DB} \supseteq L^{DB \cup db}$). In contrast, TBAR would correctly recognize the situation and not require any passes over DB. However, it would still need to do a complete mining over the increment, requiring k-passes.

The Borders algorithm, on the other hand, does not feature this optimization and may therefore have to scan DB. Finally, in DELTA, we ensure like TBAR that no pass is required over the old database. Further, only *one pass* over the increment is required. Note that this is also the minimal that is required, showing that DELTA is *optimal* in the above scenario.

5.3.4 Deletion of Tuples

While typically the increments to historical databases consist of fresh insertions, it is also conceivable that there may be occasional deletion of tuples as well [78]. This is handled in DELTA similar to the manner suggested in Borders: While counting, the count of each subset of a deleted tuple is decremented. Besides this, the two cases of addition and deletion of tuples need not be considered separately since Theorem 5.1.2 (Section 5.1) is independent of how an itemset of the negative border becomes large.

5.4 Multi-Support Incremental Mining in DELTA

In the previous section, we considered incremental mining in the context of "equi-support" environments. As mentioned in the Introduction, however, we would expect that user requirements would typically change with time, resulting in different minimum support levels across mining operations. In DELTA, we address this issue which has not been considered in the literature. We feel that this is an important value addition given the inherent exploratory nature of mining.

For convenience, we break up the multi-support problem into two cases: *Stronger*, where the current threshold is higher (i.e., $sup_{min}^{DB \cup db} > sup_{min}^{DB}$), and *Weaker*, where the current threshold is lower (i.e., $sup_{min}^{DB \cup db} < sup_{min}^{DB}$). We now address each of these cases separately:

5.4.1 Stronger Support Threshold

The stronger support case is handled almost exactly the same way as the equi-support case, that is, as though the threshold has *not changed*. The only difference is that the following modification is incorporated:

Initially, all itemsets which are not large w.r.t. $sup_{min}^{DB \cup db}$ are removed from L^{DB} and the corresponding negative border is then calculated. The itemsets that are removed are not discarded completely, but are retained separately since they may become large after counting over the increment db. After computing the negative border closure, if the counts of all the itemsets in the closure are already known, *the pass over DB becomes unnecessary*; otherwise, the itemsets for which the count information is unavailable are counted over DB.

5.4.2 Weaker Support Threshold

The weaker support case is much more difficult to handle since the L^{DB} set now needs to be *expanded* but the identities of these additional sets cannot be deduced from the increment db. The pseudo-code for the way DELTA handles this case is given as function **DeltaLow** in Figure 5.2, and operates as described below.

```
DeltaLow (DB, db, L^DB, N^DB, sup_min^DB, sup_min^{DB∪db})
Input: Previous Database DB, Increment db, Previous Large Itemsets L^DB,
       Previous Negative Border N^DB, Previous Minimum Support Threshold sup_min^DB,
       Present Minimum Support Threshold sup_min^{DB∪db}
Output: Updated Set of Large Itemsets L^{DB∪db}, Updated Negative Border N^{DB∪db}
begin
1.      UpdateCounts(db, L^DB ∪ N^DB);      // pass over db
2.      L' = GetLarge(L^DB ∪ N^DB, sup_min^{DB∪db} * |DB ∪ db|);
3.      NBetween = NegBorder(L') − (L^DB ∪ N^DB);
4.      // perform lines 2–31 of DELTA for equi-support case using sup_min^DB with
        // the following modification: find the counts of itemsets in NBetween also
        // over (DB ∪ db). Let (L, N) be the output obtained by this process.
5.      L = L ∪ GetLarge(NBetween, sup_min^{DB∪db} * |DB ∪ db|);
6.      Small = N ∪ (NBetween − L);
7.      if (NegBorder(L) ⊆ Small)
8.          get supports of itemsets in NegBorder(L) from Small
9.          output(L, NegBorder(L));
10.     C = L;
11.     ResetCounts(C);
12.     do      // compute negative border closure
13.         C = C ∪ NegBorder(C);
14.         C = C − Small // prune
15.     until C does not grow
16.     C = C − (L ∪ Small)
17.     UpdateCounts(DB ∪ db, C);      // additional pass over DB ∪ db
18.     L^{DB∪db} = L ∪ GetLarge(C, sup_min^{DB∪db} * |DB ∪ db|);
19.     N^{DB∪db} = NegBorder(L^{DB∪db});
20.     get supports of itemsets in N^{DB∪db} from (C ∪ Small)
21.     output(L^{DB∪db}, N^{DB∪db});
end
```

Figure 5.2: **DELTA for Weaker Support Threshold (DeltaLow)**

After updating L^{DB} and N^{DB} over the increment (line 1 in Figure 5.2), some itemsets may move from N^{DB} to L^{DB} due to the lowered threshold. By Theorem 5.1.2 (Section 5.1), we need to only consider itemsets that are extensions of these itemsets. Let L' be the resultant set of large itemsets after updating over the increment (line 2). The immediate extensions are computed as $\mathsf{NegBorder}(L') - (L^{DB} \cup N^{DB})$. We refer to this set as $NBetween$ since these itemsets are likely to have supports *between* sup_{min}^{DB} and $sup_{min}^{DB \cup db}$ (line 3). An important point to note here is that for these itemsets Theorem 5.1.1 *does not hold* due to the lowered support threshold.

We now execute DELTA as if the support *had not changed*. A difference, however, is that we simultaneously find the counts of itemsets in $NBetween$ over $DB \cup db$ (line 4). Following this, the counts of all large 1-itemsets over $DB \cup db$ are available. This is because $L^{DB} \cup N^{DB}$ contains all possible 1-itemsets [10]. We also have the counts of all 2-itemsets of $L^{DB \cup db} \cup N^{DB \cup db}$. This is because L' contains all large 1-itemsets in $DB \cup db$ and $NBetween$ contains the immediate extensions of L' that are not already in $(L^{DB} \cup N^{DB})$.

Let L be the set of all large itemsets whose counts are known (line 5), and let $Small$ be the set of itemsets with known counts which are not in L (line 6). If the counts of the negative border of L are already known, then the algorithm terminates (lines 7–9). Otherwise, all the remaining extensions of L that could become large are determined by computing the negative border closure (lines 10–16). (As in the equi-support case, we expect that the remaining layers of the closure can be generated together since the number of 2-itemsets in L is typically much smaller than the overall number of all possible 2-itemset pairs.) The itemsets of the closure are counted over the entire database (line 17), and the final set of large itemsets and its negative border are determined (lines 18–20).

5.5 Performance Study

In the previous sections, we presented the **FUP**, **TBAR** and **Borders** incremental mining algorithms, apart from our new **DELTA** algorithm. To evaluate the relative performance of these algorithms and to confirm the claims that we had informally made about their expected behavior, we conducted a series of experiments that covered a range of database and mining workloads. The performance metric in these experiments is the *total execution time* taken by the mining operation.

5.5.1 Baseline Algorithms

We include the **Apriori** algorithm also in our evaluation suite to serve as a baseline indicator of the performance that would be obtained by directly using a "first-time" algorithm instead of an incremental mining algorithm. This helps to clearly identify the utility of "knowing the past".

Further, as mentioned in the Introduction, it is extremely useful to put into perspective *how well* the incremental algorithms make use of their "knowledge of the past", that is, to characterize the *efficiency* of the incremental algorithms. To achieve this objective, we also evaluate the performance achieved by the **ORACLE** algorithm, which "magically" knows the identities of all the large itemsets (and the associated negative border) in the current database and only needs to gather the current supports of these itemsets. Note that this idealized incremental algorithm represents the *absolute minimal amount* of processing that is necessary and therefore represents a lower bound[1] on the (execution time) performance.

The ORACLE algorithm operates as follows: For those itemsets in $L^{DB \cup db} \cup N^{DB \cup db}$ whose counts over DB are currently unknown, the algorithm first makes a pass over DB and determines these counts. It then scans db to update the counts of all itemsets in $L^{DB \cup db} \cup N^{DB \cup db}$. So, in the worst case, it needs to make one pass over the previous database and one pass over the increment.

5.5.2 Database Generation

The databases used in our experiments were synthetically generated using the technique described in [2] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator are described in Table 5.2. These are similar to those used in [2] except that the size and skew of the increment are two additional parameters. Since the generator of [2] does not include the concept of an increment, we have taken the following approach, similar to [77]: The increment is produced by first generating the entire $DB \cup db$ and then dividing it into DB and db.

[1] Within the framework of the data and storage structures used in our study.

Parameter	Meaning	Values
N	Number of items	1000
T	Mean transaction length	10
L	Number of potentially large itemsets	2000
I	Mean length of potentially large itemsets	4
D	Number of transactions in database DB	4 M (200 MB disk occupancy)
d	Number of transactions in increment db	1%, 10%, 50%, 100% of D
S	Skew of increment db (w.r.t. DB)	Identical, Skewed
p_{is}	Prob. of changing large itemset identity	0.33 (for Skewed)
p_{it}	Prob. of changing item identity	0.50 (for Skewed)

Table 5.2: **Parameter Table**

Data Skew Generation

The above method will produce data that is *identically* distributed in both DB and db. However, as mentioned earlier, databases often exhibit temporal trends resulting in the increment perhaps having a different distribution than the previous database. That is, there may be significant changes in both the number and the identities of the large itemsets between DB and db. To model this "skew" effect, we modified the generator in the following manner: After D transactions are produced by the generator, a certain percentage of the potentially large itemsets are changed. A potentially large itemset is changed as follows: First, with a probability determined by the parameter p_{is} it is decided whether the itemset has to be changed or not. If change is decided, each item in the itemset is changed with a probability determined by the parameter p_{it}. The item that is used to replace the existing item is chosen uniformly from the set of those items that are not already in the itemset. After the large itemsets are changed in this manner, d number of transactions are produced with the new modified set of potentially large itemsets.

5.5.3 Itemset Data Structures

In our implementation of the algorithms, we generally use the *hashtree* data-structure [2] as a container for itemsets. However, as suggested in [76], the 2-itemsets are not stored in hashtrees but instead in a 2-dimensional triangular array which is indexed by the large 1-itemsets. It has been reported (and also confirmed in our study) that adding this optimization results in a considerable improvement in performance. All the algorithms in our study are implemented with this optimization.

5.5.4 Overview of Experiments

We conducted a variety of experiments to evaluate the relative performance of DELTA and the other mining algorithms. Due to space limitations, we report only on a representative set here. In particular, the results are presented for the workload parameter settings shown in Table 5.2.

The experiments were conducted on an UltraSparc 170E workstation running Solaris 2.6 with 128 MB main memory and a 2 GB local SCSI disk. A range of rule support threshold values between 0.33% and 2% were considered in our equi-support experiments.

The previous database size was always kept fixed at 4 million transactions. Along with varying the support thresholds, we also varied the size of the increment db from 40,000 transactions to 4 million transactions, representing an increment-to-previous database ratio that ranges from 1% to 100%.

Two types of increment distributions are considered: *Identical* where both DB and db have the same itemset distribution, and *Skewed* where the distributions are noticeably different. For the *Skewed* distribution for which results are reported in this chapter, the p_{is} and p_{it} parameters were set to 0.33 and 0.5 as mentioned in Table 5.2. With these settings, at the 0.5 percent support threshold and a 10% increment, for example, there are over 700 large itemsets in db which are not large in DB, and close to 500 large itemsets in DB that are not large in db.

We also conducted experiments wherein the new minimum support threshold is different from that used in the previous mining. The previous threshold was set to 0.5% and the new threshold was varied from 0.2% to 1.5%. Therefore, both the Stronger Threshold and Weaker Threshold cases outlined in Section 5.3 are considered in these experiments.

5.6 Experimental Results

In this section, we report on the results of our experiments comparing the performance of the various incremental mining algorithms for the dynamic basket database model described in the previous section.

5.6.1 Experiment 1: Equi-support / Identical Distribution

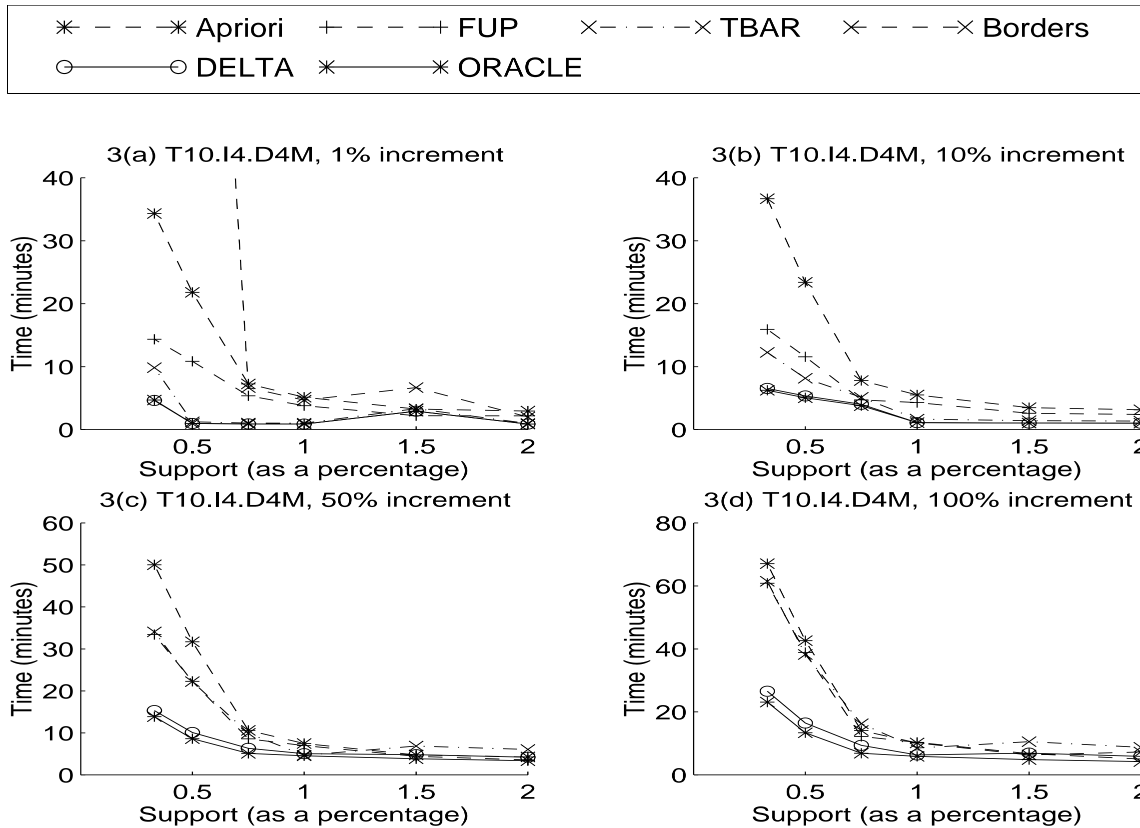

Figure 5.3: **Equi-support / Identical Distribution**

Our first experiment considered the equi-support situation with identical distribution between DB and db. For this environment, the execution time performance of all the mining algorithms is shown in Figures 5.3a–d for increment sizes ranging from 1% to 100%.

Focusing first on FUP, we see in Figure 5.3 that for all the increment sizes and for all the support factors, FUP performs better than or almost the same as Apriori. Moving on to TBAR, we observe that it outperforms both Apriori and FUP at small increment sizes and low supports. At high supports, however, it is slightly worse than Apriori due to the overhead of maintaining the negative border information. As the increment size increases, TBAR's performance becomes progressively degraded. This is explained as follows: Firstly, TBAR updates the counts of itemsets in $L^{DB} \cup N^{DB}$ over db – these itemsets are precisely the same as the set of all candidates generated in running Apriori over DB. Secondly, it performs a complete Apriori-based mining over db. When $\mid db \mid = \mid DB \mid$, the total cost of these two factors is the same as the total cost incurred by the Apriori algorithm. However, TBAR finally loses out because it needs to make a further pass over DB.

Turning our attention to Borders, we find in Figure 5.3a, which corresponds to the 1 percent increment, that while for much of the support range its performance is similar to that of FUP and TBAR, there is a sharp degradation in performance at a support of 0.75 percent. The reason for this is the "candidate explosion" problem described earlier in Section 5.2. This was confirmed by measuring the number of candidates for supports of 1 percent and 0.75 percent – in the former case, it was a little over 1000 whereas in the latter, it had jumped to over 30000!

The above candidate explosion problem is further intensified when the increment size is increased, to the extent that its performance is an order of magnitude worse than the other algorithms – therefore we have not shown Borders performance in Figures 5.3b–d.

Finally, considering DELTA, we find that it significantly outperforms all the other algorithms at lower support thresholds for all the increment sizes. In fact, in this region, *the performance of DELTA almost coincides with that of ORACLE*. The reason for the especially good performance here is the following – low support values result in tighter values of k, the maximal large itemset size, leading to correspondingly more iterations for FUP over the previous database DB, and for TBAR over the increment db. In contrast, DELTA requires only three passes over the increment and one pass over the previous database. Further, because of its pruning optimizations, the number of candidates to be counted over the previous database DB is significantly less as compared to TBAR – for example, for a support threshold of 0.5 percent and a 50% increment (Figure 5.3c), it is smaller by a factor of *two*.

We note that the marginal non-monotonic behavior in the curves of TBAR, Borders, DELTA and ORACLE at low increment sizes is due to the fact that only sometimes do they need to access the original database DB and this is not a function of the minimum support threshold.

5.6.2 Experiment 2: Equi-support / Skewed Distribution

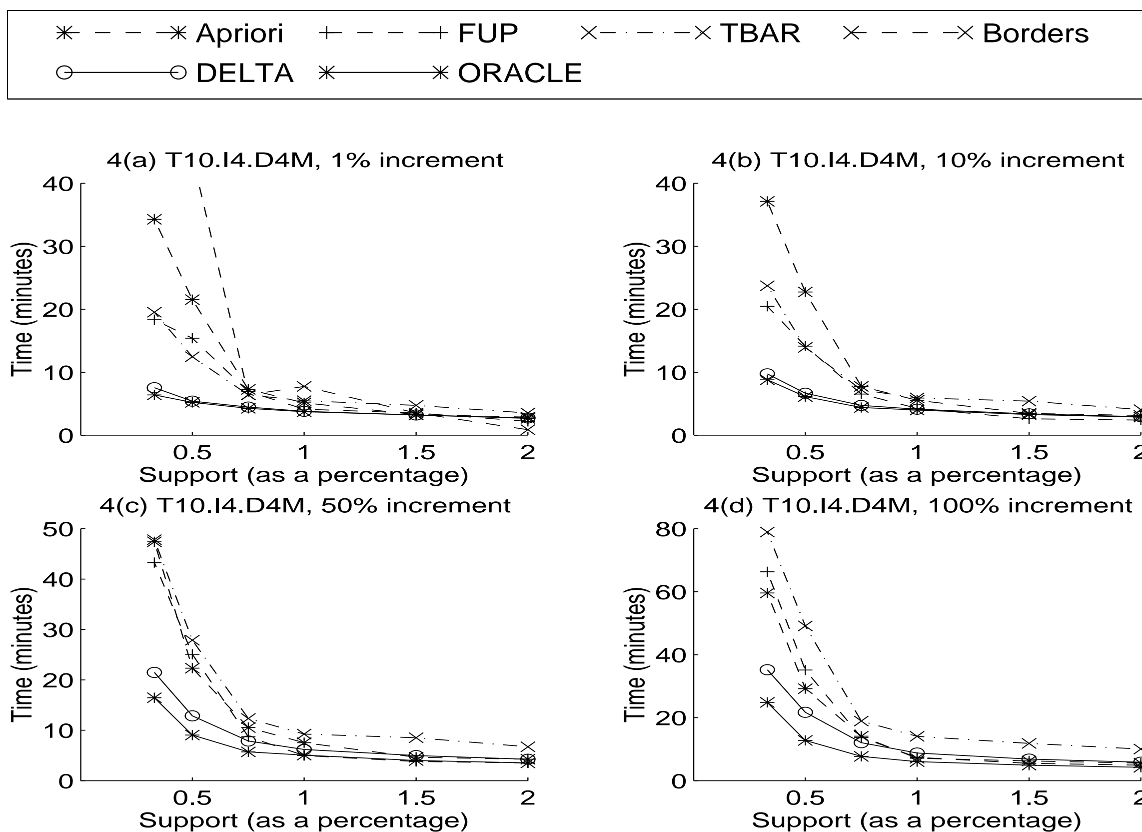

Figure 5.4: **Equi-support / Skewed Distribution**

Our next experiment considered the Skewed workload environment, all other parameters being the same as that of the previous experiment. The execution time performance of the various algorithms for this case is shown in Figures 5.4a–d. We see here that the effect of the skew is pronounced in the case of both TBAR and Borders, whereas the other algorithms (including DELTA) are relatively unaffected.

The effect of skew is noticeable in the case of TBAR since it relies solely on the increment to prune candidates from its computation of the closure and therefore many unnecessary candidates are generated which later prove to be small over the entire database. Borders, on the other hand, is affected because the number of 1-itemsets that are in the promoted border tends to increase when there is skew. For instance, for a minimum support of 0.33% and an increment of 10%, there were nine 1-itemsets among the promoted borders and the number of large itemsets was 4481, resulting in over 2 million candidates.

In contrast to the above, Apriori and FUP are not affected by skew since the candidates that they generate in each pass are determined only by the *overall* large itemsets, and not by the large itemsets of the increment.

DELTA is not as affected by skew as TBAR, since it utilizes the *complete* negative border information to prune away candidates. That is, all itemsets which are known to be small either over $DB \cup db$ or over db are pruned away during closure generation, and not merely those candidates which are small over db. Hence, DELTA is relatively stable with respect to data skew. As in the Identical distribution case, it can be seen in Figures 5.4a–b that for small increment sizes, its performance almost coincides with that of ORACLE. It however degrades to some extent for large skewed increments because of two reasons: (1) the number of itemsets in $L^{DB} - L^{DB \cup db}$ increases, resulting in more unnecessary candidates being updated over db, and (2) the number of itemsets in $L^{DB \cup db} - L^{DB}$ increases, resulting in more promoted borders followed by more candidates over DB. Even in these latter cases it is seen to perform considerably better than other algorithms. For example, for a minimum support of 0.33% and an increment of 100%, its performance is more than twice as good as that of TBAR.

5.6.3 Experiment 3: Multi-Support / Identical Distribution

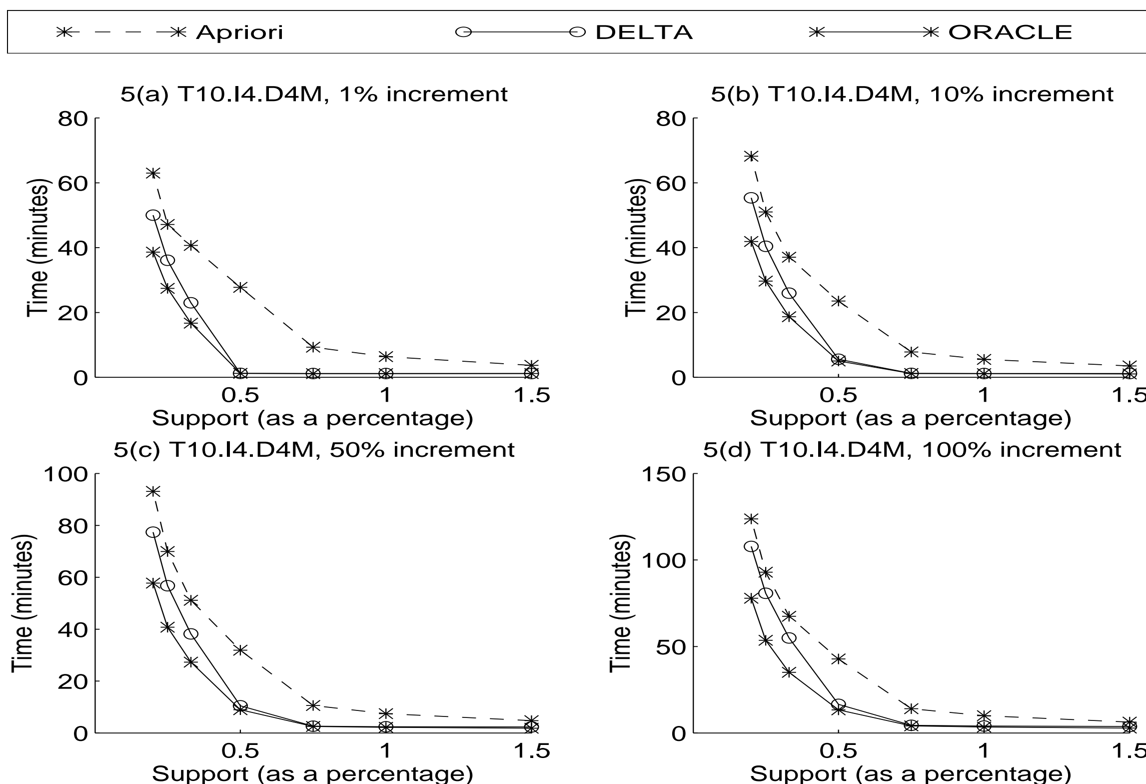

Figure 5.5: **Multi-Support / Identical Distribution [Previous Support = 0.5%]**

The previous set of experiments modeled equi-support environments. We now move on to considering *multi-support* environments. In these experiments, we compare the performance of DELTA with that of Apriori and ORACLE only since, as mentioned earlier, FUP, TBAR and Borders do not handle the multi-support case.

In this experiment, we fixed the initial support to be 0.5% and the new support was varied between 0.2% and 1.5%, thereby covering both the Weaker Threshold and Stronger Threshold possibilities. For this environment, Figures 5.5a–d show the performance of DELTA relative to that of Apriori for the databases where the distribution of the increments is Identical to that of the previous database. The corresponding graphs for the Skewed case are not shown here due to space constraints but are available in [83] and are similar in nature to those in Figures 5.5a–d.

We note here that at either end of the support spectrum, DELTA performs very similarly to Apriori whereas in the "middle band" it does noticeably better, especially for moderate increment sizes (Figures 5.5a–b). In fact, the performance gain of DELTA is maximum when the new minimum support threshold is the same as the previous threshold and tapers off when the support is changed in either direction. At very low support thresholds, the number of large itemsets increases exponentially, and therefore the number of candidates generated in the negative border closure in DELTA will be a few more than the number of candidates generated in Apriori. Most of the candidates will have support less than the previous minimum threshold, and hence all of them have to

Parameters	Meaning	Values
R	Number of roots	250
L	Number of levels	4
F	Fanout	5
D	Depth-ratio	1

Table 5.3: **Taxonomy Parameter Table**

be counted over the previous database. Therefore, the performance of DELTA approaches that of Apriori in the low support region. In the high support region, on the other hand, most of the candidates do not turn out to be large and hence both algorithms perform almost the same amount of processing.

5.7 Handling Generalized Association Rules

Having established DELTA's good performance characteristics for BAR-mining, we move on in this section to describing how DELTA can be extended to handle *generalized* association rules. In [62], three first-time algorithms to mine generalized rules were presented — Basic, Cumulate and Stratify. We chose Cumulate as a representative of these algorithms since it performed the best on most of our workloads. Cumulate is an extension of the Basic algorithm which merely consists of augmenting each tuple with all ancestors of items in that tuple. In particular, Cumulate includes the following optimizations:

- **Filtering the ancestors added to transactions.** Only ancestors of items in the transaction that are also present in some candidate itemset are added.
- **Pre-computing ancestors.** Rather than finding ancestors for each item by traversing a taxonomy graph, the ancestors for each item are precomputed.
- **Pruning itemsets containing an item and its ancestor.** An itemset that contains both an item and its ancestor may be pruned as it will have the same support as the itemset which doesn't contain that ancestor, hence being redundant. This pruning needs to be done explicitly only once which is before the second pass over the database.

The above optimizations of Cumulate were also included in our implementations of DELTA and ORACLE for generalized rules. The third optimization in Cumulate above is performed in DELTA at the end of the first pass over the increment, since at this stage the identities of potentially large 2-itemsets (over $DB \cup db$) are known, and hence no further candidate 2-itemsets would be generated. Since it is sufficient to perform the third optimization to 2-itemsets only [62], this optimization does not have to be applied again.

5.7.1 Performance Evaluation: Generalized / Equi-support / Identical Distribution

For evaluating the performance of DELTA on GAR-mining, we compared it with Cumulate and ORACLE as no previous incremental algorithms are available for comparison. The generalized databases were generated using the same technique as in [62]. The parameters used and their specific values are identical to those in Section 5.5 except for the number of items (N) and the number of potentially large itemsets (L) which were both set to 10000. Additional parameters required for the taxonomy and their specific values are shown in Table 5.3.

The execution time performance of the Cumulate, DELTA and ORACLE mining algorithms for the equi-support identical distribution environment is shown in Figures 5.6a–d for increment sizes ranging from 1% to 100%. The performance was measured only for supports between 0.75% and 2% since for lower supports, the running time of all the algorithms was in the range of several hours.

For all support thresholds and database sizes, we find that DELTA significantly outperforms Cumulate, and is in fact very close to ORACLE. We see that DELTA exhibits a huge performance gain over Cumulate, upto *as much as 9 times* at the 1% increment and 0.75% support threshold, and as much as 3 times on average. In fact the performance of DELTA is seen to overlap with that of ORACLE for small increments (Figures 5.6a–b). The reason for this is the number of candidates in DELTA over both db and DB were only marginally more than that in ORACLE. This is again owing to the fact that the set of large itemsets with its negative border is relatively stable, and that DELTA prunes away most of the unnecessary candidates in its second pass over the increment.

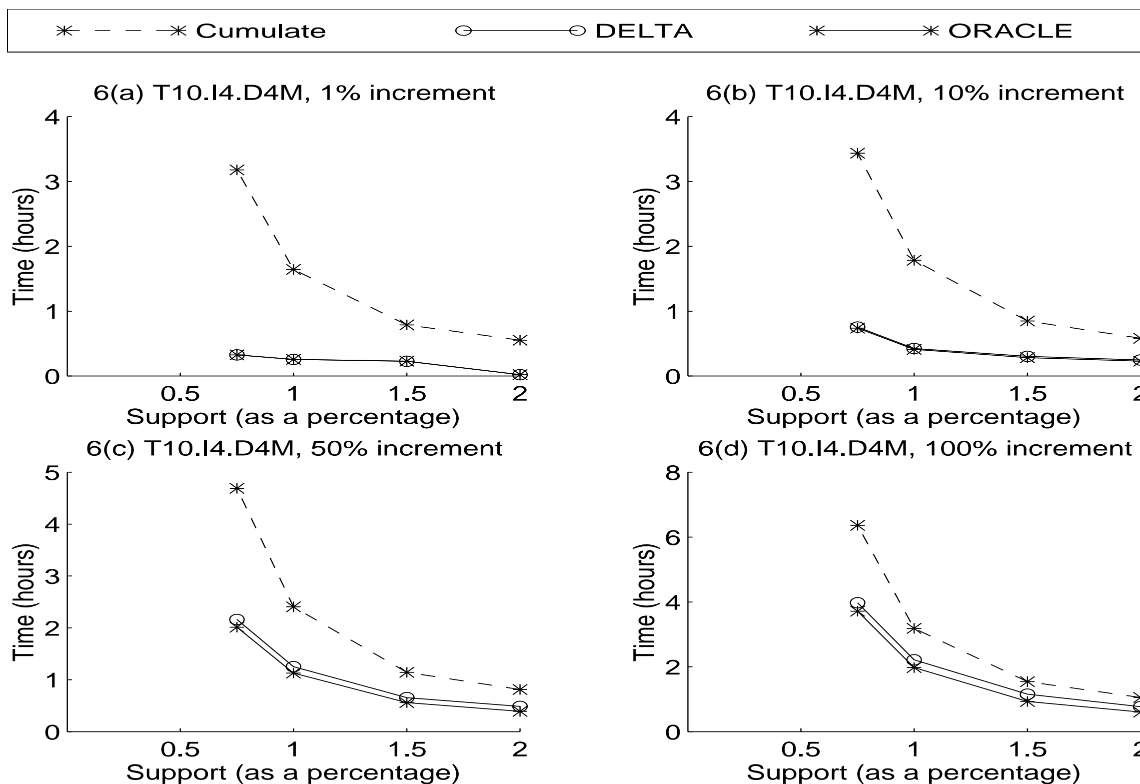

Figure 5.6: **Equi-support / Identical Distribution (Generalized Rules)**

Owing to space constraints, the results of experiments where the distribution is Skewed, as also the multi-support experiments are not presented here. They are however available in [83], and are similar in nature to those presented for BAR-mining in Section 5.6.

5.8 Conclusions

We considered the problem of incrementally mining association rules on market basket databases that have been subjected to a significant number of updates since their previous mining exercise. Instead of mining the whole database again from scratch, we attempt to use the previous mining results, that is, knowledge of the itemsets which are large in the previous database, their negative border, and their associated supports, to efficiently identify the same information for the updated database.

We proposed a new algorithm called DELTA that guarantees completion of mining in three passes over the increment and one pass over the previous database. This compares favorably with previously proposed incremental algorithms like FUP and TBAR wherein the number of passes is a function of the length of the longest large itemset. Also, DELTA does not suffer from the candidate explosion problem associated with the Borders algorithm since it initially computes only the immediate extensions of the promoted borders, not their entire closure.

DELTA's design was extended to handle multi-support environments, an important issue not previously addressed in the literature, at a cost of only one additional pass over the current database.

Using a synthetic database generator, the performance of DELTA was compared against that of FUP, TBAR and Borders, and also the two baseline algorithms, Apriori and ORACLE. Our experiments showed that for a variety of increment sizes, increment distributions and support thresholds, DELTA performs significantly better than the previously proposed incremental algorithms. *In fact, for many workloads its performance approached that of ORACLE, which represents a lower bound on achievable performance, indicating that DELTA is quite efficient in its candidate pruning process.* Also, while the TBAR and Borders algorithms were sensitive to skew in the data distribution, DELTA was comparatively robust.

In the special scenario where no pass over the previous database is required since the new results are a subset of the previous results, DELTA's performance is optimal in that it requires only one pass over the increment

whereas all the other algorithms either are unable to recognize the situation or require multiple passes over the increment.

Finally, DELTA was shown to be easily extendible to generalized association rules, while maintaining its performance close to ORACLE. No prior work exists on extending incremental mining algorithms to handle generalized rules.

In summary, DELTA is a practical, robust and efficient incremental mining algorithm. Our current work includes extending the DELTA algorithm to handle quantitative rules [63] and also to develop incremental algorithms for sequence [74] and classification rules [75].

Chapter 6

Rule Generation

In the previous two chapters, we showed how to efficiently generate frequent itemsets for both fresh databases and incremental databases. We now move on to the second phase of mining, that is *rule generation* from the frequent itemsets.

For each frequent itemset I, rules are normally generated as follows: Consider all possible subsets of this itemset. If A is any such subset, then

$$A \Longrightarrow (I - A)$$

is a rule if it has enough confidence. There could be many such rules. However, not all such rules are interesting. Specifically, if the above ruls has minimum confidence, then any rule

$$B \Longrightarrow (I - B)$$

also has minimum confidence (and minimum support), for any B that is a superset of A. Thus, it is necessary to find *only* all the *minimal* rules: A rule is *minimal* if there is no subset of its LHS which can also form a rule with minimum confidence. That is, rules such as the second one above are not minimal – instead, they can be inferred. Among all rules which can be inferred from a *minimal* rule, the *minimal* rule has the longest RHS and the shortest LHS.

Further, it pays to find only minimal rules because, given a set of frequent itemsets with m being the length of the biggest itemset in this set, the number of rules that can be generated is of order *exponential* in m. However, the number of minimal rules is **linear** in the size of m. Such a drastic reduction in the number of uninteresting rules is certainly worthwhile.

To address the above issue, we present the **RULEGEN** algorithm in Figure 6.1 – this algorithm efficiently finds all *minimal* rules, given a set of frequent itemsets, L. The RULEGEN algorithm closely resembles the *Apriori* algorithm in its structure.

1.	For each itemset I in L do
2.	C_1 = set of all 1-itemsets of I
3.	$k = 1$
4.	$R = \emptyset$ /* set of interesting rules */
5.	while ($C_k \neq \emptyset$) do
6.	for each itemset A in C_k do
7.	if $A \Longrightarrow (I - A)$ has enough confidence, then
8.	remove A from C_k and add it to R.
9.	C_{k+1} = AprioriGen(C_k)
10.	$k++$

Figure 6.1: The RULEGEN Algorithm

Chapter 7

Vertical Mining

A feature common to the mining algorithms discussed in the previous chpaters is that they are designed for use on databases where the data layout is *horizontal*. In a horizontal layout, the database is organized as a set of rows with each row consisting of a transaction identifier (TID) and a list of the identifiers of the items present in the transaction.

A plausible alternative database structure is a *vertical* layout wherein each *item* is associated with a column that contains the list of tuples in which it appears – this list could be maintained either in *bit-vector* format or in *tid-list* format. Since association rule mining's objective is to discover correlated *columns*, the item-based vertical layout appears to be a *natural* choice for achieving this goal. Further, vertical partitioning opens up possibilities for fast and simple support counting, for reducing the effective database size, for compact storage of the database, for better support of dynamic databases, and for asynchrony in the counting process. These features are described in detail in Section 7.2.

Previous Work

Although the vertical layout has several positive features, as described above, it has received comparatively little attention in the data mining literature. In fact, to the best of our knowledge, it has been considered only in [13, 16] and a series of papers by Zaki et al [18, 20, 19].

The vertical layout was first proposed in [13] which described how, with this layout, itemset supports can be counted using only the simple set operations of union and intersection. Subsequently, a graph-based algorithm called DLG (Direct Large itemset Generation) was proposed in [16]. These ideas were further developed in the Eclat and Clique algorithms of [20], wherein lattice-theoretic concepts are employed to cluster related itemsets. The Eclat and Clique algorithms incorporate a bottom-up search strategy – replacing this with a hybrid top-down/bottom-up search strategy results in the MaxEclat and MaxClique variants [20]. Performance evaluation of these algorithms against classical horizontal layout algorithms such as Apriori [2] and Partition [15] indicated the potential for significant performance improvements. Finally, extensions of Eclat and Clique to shared-memory parallel (SMP) database architectures were discussed in [18, 19].

While the above studies served to highlight the utility of the vertical approach, each has one or more of the following limitations (discussed in detail in Section 7.5): First, the candidate itemset pruning strategies are coarse-grained. Second, the ability to handle large databases, especially those where the current "active segment" or "working set" of the database may not completely fit into main memory, is unclear. Third, dynamic databases where the database may be augmented between successive mining operations are not handled efficiently. Fourth, the benefits obtainable from data compression are not considered. Finally, there is significant repetition of previously completed processing.

Contributions

In this chapter, we present a new vertical mining algorithm called **VIPER** (Vertical Itemset Partitioning for Efficient Rule-extraction). The focus of VIPER's design is on simultaneously achieving *scalable* performance with respect to database size and aggressively exploiting the benefits obtainable from the vertical database layout. VIPER combines efficient disk usage with several optimizations to reduce computation, in order to obtain high performance gains and close to linear scalability.

VIPER is based on a *bit-vector* layout wherein each vector is stored in *compressed* format – in the remainder of this chapter, we will use the term "snake" to refer to an itemset and its associated compressed bit vector.

A special compression technique based on Golomb encoding [12] is used to ensure a compact representation of the snakes that is amenable to easy processing. Efficient generation of candidate itemsets is ensured by capitalizing on the inherent clustering of the frequent itemsets. Further, candidate itemsets of *multiple* levels can be counted in each pass at minimal additional cost using a novel scheme for merging snakes. Finally, the number of snakes that are read and written during each pass of the mining process is minimized using a variety of pruning techniques. Most of these optimizations are possible only due to the vertical layout of the database.

Using a synthetic database generator, we analyze the response time performance of VIPER for a variety of database workloads and compare it with that of MaxClique, which represents the best among the previously proposed vertical mining algorithms. To serve as a baseline, we also include the horizontal-layout-based classical Apriori algorithm in our evaluation suite. An important feature of our experiments is that they include workloads where the database is large enough that the working set of the database cannot be completely stored in memory. This situation may be expected to frequently arise in data mining applications since they are typically executed on huge historical databases. In this respect, our study evaluates environments not considered in previous vertical mining papers.

Our experimental results indicate that VIPER provides significant performance gains over both MaxClique and Apriori, especially for large databases. Most importantly, it shows close to linear scaleup with database size, unlike MaxClique whose performance deteriorates for large databases.

Organization

The remainder of this chapter is organized as follows: A discussion on the suitability of the vertical database layout for association rule mining is presented in Section 7.1. Our new VIPER algorithm is presented in Sections 7.2 through 7.4. An overview of the previous vertical mining algorithms, especially MaxClique, and their limitations is provided in Section 7.5. The performance model and the experimental results are highlighted in Section 7.6. Finally, in Section 7.7, we summarize the conclusions of our study and outline future avenues to explore.

7.1 Vertical Mining

The vertical database layout appears to have several advantages over the horizontal layout, including the following:

- Given the association rule mining objective of finding correlated *items*, the item-based vertical layout appears to be a *natural* choice for efficiently achieving this goal.

- Computing the supports of itemsets is much simpler and faster with the vertical layout since it involves only the intersections of tid-lists or bit-vectors, whereas complex hash-tree data structures [2] are required to perform the same function for horizontal layouts.

  This optimization is especially important since earlier studies [2] have found that data mining is a CPU-intensive process and that hash-tree lookups are responsible for a substantial part of this load. It is also important for a completely different reason – it is amenable to implementation on general purpose DBMS since they already efficiently support the standard set operations.

- With the vertical layout, there is an automatic "reduction" of the database before each scan in that only those itemsets that are *relevant* to the following scan of the mining process are accessed from disk.

  In the horizontal layout, however, extraneous information that happens to be part of a row in which useful information is present is *also transferred* from disk to memory. This is because database reductions are comparatively hard to implement in the horizontal layout. Further, even if reduction were possible, the extraneous information can be removed only in the scan *following the one* in which its irrelevance is discovered. Therefore, there is always a *reduction lag* of at least one scan in the horizontal layout.

- Compression of columns *scales* with the database size whereas compression of rows is limited by the average transaction length and is independent of the number of rows.

- Finally, the vertical layout permits *asynchronous* computation of the frequent itemsets. For example, once the supports of items A and B are known, counting the support of their combination AB can commence even before the other items have been completely counted. This is in marked contrast to the horizontal approach where the counting of all itemsets has to proceed synchronously with the scan of the database.

We believe that asynchrony will prove to be an especially important advantage in *parallel* implementations of the mining process.

While, as described above, the vertical layout appears to have several advantages, a careful algorithmic design is required to ensure that these advantages are materialized – we attempt this in the VIPER algorithm, which is described in the following section.

For ease of exposition, we will use the following notation in the remainder of this chapter:

$\mathcal{I}$	Set of items in the database
D	Database of customer purchases
$\mid D \mid$	Cardinality of the database
$minSup$	User-specified minimum rule support
F_k	set of frequent k-itemsets in D
F	set of frequent itemsets in D
C_k	Set of candidate k-itemsets in D

7.2 The VIPER Algorithm

In this section, we overview the main features of the VIPER algorithm and describe its primary components in detail in the following sections.

7.2.1 Overview

VIPER uses a *bit-vector* data format for representing an itemset's occurence in the database. Further, the bit-vector is stored in a *compressed* format. Our compression technique takes advantage of the *sparseness* that is typically exhibited in itemset bit-vectors. We will hereafter refer to an itemset and its associated compressed bit-vector as a **"Snake"**. Since each snake is associated with a unique itemset and a compressed bit-vector, we shall use the term "frequent snake" to mean that the corresponding itemset is frequent. Finally, the term i-length snake will be used to refer to a snake corresponding to an itemset comprised of i items.

VIPER is a multi-pass algorithm, wherein data (in the form of snakes) is read from and written to the disk in each pass. It proceeds in a bottom up manner and at the end of the data mining, the supports of all frequent itemsets are available. Each pass involves the simultaneous counting of *several levels* of candidates via intersections of the input snakes. While the writing of data may appear to represent an additional overhead as compared to "read-only" algorithms like Apriori, we expect that this is a *positive tradeoff* since the data that is written back may help to considerably speed up the subsequent mining process. Further, the disk traffic is minimized in a variety of ways, described below.

7.2.2 Disk Traffic Reduction

Single Scan per Snake: We propose a novel DAG-based counting scheme that does a *simultaneous merge* of all the snakes read in a particular pass, and ensures that any given snake is read *at most once*. This is a substantial gain compared to other vertical algorithms, wherein a given itemset may be read multiple times.

Fewer Passes over the Database: Our Snake-Merge scheme is capable of counting *several* levels of candidate itemsets within a single pass. More specifically, by merging i-snakes, we compute the support for candidates of length $i+1$ through $2i$. This results in greatly reducing the number of passes over the database.

Database Size Reduction: The snakes written to disk are a carefully selected *subset* of the overall set of snakes that are processed – in particular, only snakes that are relevant to future passes are saved.

Data Compression: Snakes are always maintained in *compressed* format on the disk. Our compression technique gives us significant compression ratios on both the bit-vector representation of the horizontal database, and on the intermediate snakes. In fact, the compression is to an extent that although the bit-vector representation is typically much larger than the original horizontal database (since it explicitly represents the *absence* of an item in a transaction), the compressed set of snakes occupy less space than the horizontal representation.

In section 7.6, we show that with the above optimizations, VIPER actually has *less* overall disk traffic than approaches that do not do any disk writes.

Processing Cost Reduction

VIPER also incorporates optimizations to minimize the *in-memory processing costs*: Firstly, it uses simple data-structures that are not memory-intensive. Secondly, it uses efficient snake merging techniques that eliminate the need for detection of item subsets in a transaction. Thirdly, the candidate generation scheme has been tuned to impose only a relatively small CPU overhead.

7.2.3 Snake Generation and Compression

Snakes are generated in the following manner: A main-memory buffer is maintained for each itemset whose snake is currently being "materialized". The snake portions corresponding to these itemsets are first accumulated in these buffers before being written out to disk. Each time an itemset occurs in a transaction, the TID is passed to a compression routine (described below) which generates the compressed version on the fly and adds it to the buffer associated with the itemset. Once a buffer is full, it is written to a disk-resident *common* file.[1] The pages associated each itemset are chained together through a linked list of pointers.

At first glance it may seem that the classical and simple to implement *Run-Length Encoding (RLE)* would be the appropriate choice to compress the bit-vectors. However, we expect that while there may be long runs of 0's, runs of 1's which imply *consecutive* customers making the same purchase may be uncommon for most itemsets in transactional databases. In the worst-case, where all 1's occur in an isolated manner, the RLE vector will have *two* words for each 1 – one word for each 0 run and its following 1. This means that the resulting database will be *double* the size of the original horizontal database, which would have only one word associated with each 1 (since 0's are not explicitly represented). In short, it would result in an *expansion*, rather than compression !

We have, therefore, developed an alternative snake compression technique, called **"Skinning"**, that is based on the classical Golomb encoding scheme [12]. Here, runs of 0's and runs of 1's are divided into groups of size W_0 and W_1, respectively – the W's are referred to as "weights". Each such full group is represented in the encoded vector by a single "weight" bit set to 1. If the division is not exact, the last group (of length $R \ mod \ W_i$), where R is the length of the run, is represented by a count field that stores the binary equivalent of the remainder length, expressed in $log_2 W_i - 1$ bits. A "field separator" 0 bit is placed between the last weight-bit field and the count field to indicate the transition from the former to the latter. Note that a "run separator" for distinguishing between a run of 0's and a run of 1's, is *not* required since it is implicitly known that the run symbol changes after the count field and the number of bits used for the count field ($log_2 W_i - 1$) is fixed.[2]

Example: To illustrate the Skinning technique, consider the 30-bit vector

$$(1)^A(000)^B(1)^C(0000)^D(0000)^E(0)^F(1)^G(0000)^H(0000)^I(0)^J(1)^K(0000)^L(0)^M$$

to be encoded using weights $W_0 = 4$ and $W_1 = 1$. The alphabetic superscripts are not part of the bit vector but are included to indicate the groups associated with these weight settings.

After skinning, the resultant compressed vector is of length 25 bits:

$$(1)^A\ 0\ 0\ (11)^B\ (1)^C\ 0\ (1)^D(1)^E\ 0\ (01)^F\ (1)^G\ 0\ (1)^H(1)^I\ 0\ (01)^J\ (1)^K\ 0\ (1)^L\ 0\ (01)^M$$

where the alphabetic superscripts indicate the correspondence between the group in the original bit vector and its encoded version. The rest of the 0 bits are the field separators.

In the above "toy" example, the compression is only from 30 bits to 25 bits – however, for practical values of W_0 and W_1, much higher compression ratios are achieved. In fact, in [14], we have derived analytical formulas that indicate compression factors of approximately 3 times over the horizontal representation – this was also confirmed in our experiments. Further, note that the space occupied by the *tid-list* vertical format, which is employed by algorithms like MaxClique, is *identical* to that of the horizontal representation – therefore, we can deduce that VIPER generates significantly smaller database sizes as compared to these vertical mining algorithms also.

7.2.4 Execution Flow

We now move on to discuss the execution flow in the VIPER mining process. The pseudocode for the VIPER algorithm is given in Figure 7.1. Assuming the most general case wherein the original database is stored in *horizontal* format, the following sequence of passes (over continually shrinking databases) is executed:[3]

---

[1] The option of writing each snake into a separate file is presently not feasible since current operating systems do not permit applications to have more than a limited number of file descriptors simultaneously open.

[2] Without loss of generality, we assume that every bit vector starts with a run of 1's, possibly of zero length.

[3] If the original database is already in the vertical format, Pass 1 consists of simply counting the number of occurrences of the 1-itemsets.

```
Algorithm VIPER(D, I, minSup){
    Input: Horizontal Database : D, Set of items : I, Minimum Support : minSup
    Output: The Set of Frequent Itemsets, F

    F = countLevelOne(D);         // Pass1: Count F1 and write 1-snakes to disk
    F = F ∪ countPairs(F_1, D); // Pass2: Count F2
    i = 2 ; until (isEmpty(F_i)) {
        candDAG = createDAG(level = i); // Create a new DAG for this level
        C_i = F_i
        // Candidate Generation for levels k+1 through 2k
        for k in i + 1 to 2i do {
            C_{k+1} = ClusterGen(C_k);
            if (isEmpty(C_{k+1})) break;
            candDAG = candDAG ∪ C_{k+1};
        }
        if (isEmpty(candDAG)) break;  // VIPER complete

        readList = findReadList(candDAG);       // List of snakes to be read
        writeList = writePrune(candDAG);         // Trim the list of snakes to write

        // The snake merge procedure – also writes F_i snakes to disk
        F_{2i} = mergeCount(candDAG, readList, writeList);

        for k in i + 1 to 2i do
            F = F ∪ frequentItems(candDAG,k);
        DeleteSnakes(F_i);
        k = k * 2;
    }
}
```

Figure 7.1: **The VIPER Algorithm.**

Pass 1 and 2: In the first pass over the database, the snakes for all the 1-itemsets (i.e. individual items) are created and stored on the disk. That is, the database is converted from the horizontal format to the compressed vertical format. During this process, the supports of these 1-itemsets are counted and F_1 is determined. In the second pass, F_2 is determined through a special approach since the direct method of computing the supports of all $\binom{\mid F_1 \mid}{2}$ pairs is expected to be extremely expensive. In this special approach [20], horizontal tuples are dynamically created *on the fly in main memory* from the frequent snakes and the supports are counted based on this format using a 2-D triangular array (dimension F_1) of counters.

The important point to note in the above is that the snakes corresponding to the combination 2-itemsets which are counted in the second pass are *not* written to disk. This is because writing out the snakes of all $\binom{|F_1|}{2}$ combinations would not only be extremely expensive, but also quite wasteful given that many of the combinations may eventually turn out to be infrequent.

Generalizing the above observation, there are two features of VIPER's snake writing:

- Only a subset of frequent snakes are written to disk.
- The writing of snakes always *lags one pass behind.* If and when these snakes are required (and only snakes for frequent itemsets will be required), instead of being read from disk, they are *dynamically generated* using snakes written out in the previous pass.

Subsequent Passes: In each subsequent pass P ($P > 2$), the first step is to generate the candidate itemsets of the current level, based on the immediately previous level's frequent itemsets, using the *ClusterGen* candidate generation procedure described in Section 7.4. That is, C_{i+1} = ClusterGen(F_i). If this is not empty, the algorithm proceeds to compute the candidate itemsets of levels $i+1, i+2, \ldots, 2i$ using the same ClusterGen procedure, except that now the *candidate set* of each level is used to generate the candidate set of the next level. That is, $C_{i+1} = ClusterGen(C_i)$.

We note here that our counting scheme is capable of counting candidates of length i to k for any k, $1 < k \leq i$. This is useful in the last pass, when there may not remain any candidates beyond level $i + k$ ($k < i$), or in case the number of candidates turns out to be very large.

The generated list of candidates is inserted into our DAG-based counting structure. After employing a variety of pruning techniques, the set of snakes to be read ($ReadList$) and written ($WriteList$) in this pass are identified. Then the snakes in $ReadList$ are read into memory and the counts of all the candidate itemsets generated in this pass ($C_{i+1}, \ldots, C_{2i}$) are concurrently computed using the *Snake-Merge* procedure, described in Section 7.3. Simultaneously, the snakes in $WriteList$ are written out to disk. When the database scan is over, with the identification of all frequent itemsets F_{i+1} through F_{2i}, the pass P is completed. Also, the snakes generated in the previous pass are *deleted* in order to minimize the disk overhead.

Termination: The above process is repeated until there are no more candidate itemsets. Finally, the global frequent itemset list, F, is returned alongwith the support of each of its elements. With this information, the desired association rules can be easily determined [2].

The details of the snake merging and candidate generation techniques are described in the following sections.

7.3 Snake Merging and Counting

The DAG counting scheme proceeds by intersections of i-snakes in a simultaneous merging technique. We exploit the fact that a candidate of length upto $2i$ can be represented as the union of some two i-length itemsets. Its support can therefore be calculated by intersecting the corresponding i-length snakes. Hence, in a single pass, our DAG-based merging algorithm computes support for candidates of length $i+1$ to $2i$.

Note that the set of candidates counted in a single pass of the DAG-merge scheme is an *optimistic* list, in the sense that the candidates upto level $2i$ are generated only on the basis of frequent-itemset information at level i. However, the associated adverse performance impact due to counting unnecessary candidates is mitigated by the following factors:

- If there is a "blowup" in the number of candidates, the counting is *truncated* to consider only candidates upto length $i + k$ ($k < i$) – the appropriate value of k is dependent on the amount of memory available.
- The snake merge scheme is efficient in that it does not involve the expensive subset-detection techniques used in algorithms such as Apriori. Hence the extra candidates are relatively cheap to count.
- The DAG structure uses very little memory.

The DAG merging technique offers scope for several optimizations – in the remainder of this section, we describe some of these optimizations that are currently implemented in VIPER.

7.3.1 DAG Merge Optimization

In each pass of the VIPER algorithm, a DAG of candidate itemsets is created – an example of this DAG is shown in the box labeled "Conceptual Picture" in Figure 7.2. The "leaves" of the DAG are the large itemsets at level i. Each intermediate node at height r is a candidate of length $i+r$, and is pointed to by some two of its subsets at height $r-1$. This is easy to arrange since if an itemset is a candidate, then all its subsets are also candidates, or large itemsets.

The intuition behind this DAG structure is as follows: We know that the union of any two $i+r-1$-subsets of the $i+r$-length candidate is the candidate itself. In this sense, the pair of child nodes can be said to *cover* the candidate itemset. Further we need to count an itemset only if its immediate subsets are also present in the transaction. Hence, for an $i+r$-length candidate, we choose a pair of $i+r-1$-subsets to cover it, instead of other smaller subsets.

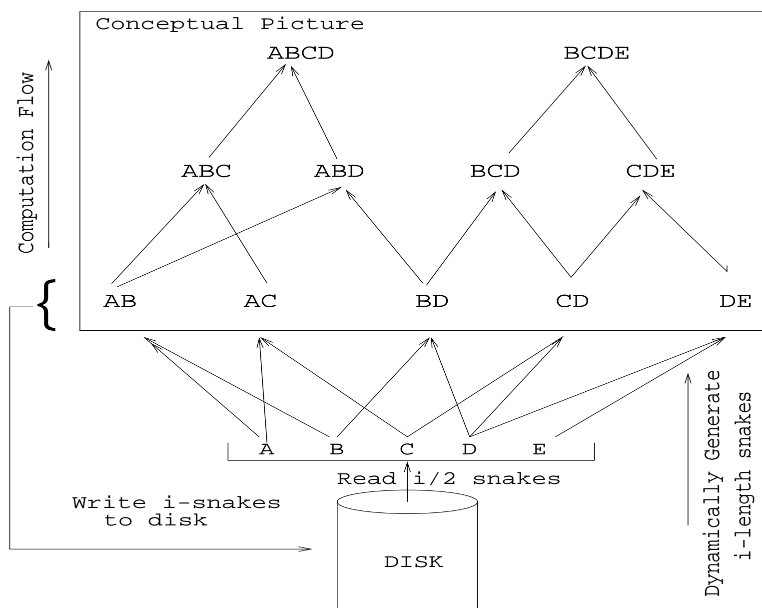

Figure 7.2: **The DAG of candidates**

These concepts are illustrated in Figure 7.2, which shows a sample portion of the DAG structure. The leaves of the DAG are the frequent 2-itemsets AB, AC, BD, CD and CE. The candidates ABC and BCD are pointed to by some two of their subsets at the previous level, namely (AB, AC) and (BD, CD), respectively. These candidates themselves point to the candidates at level 4 – $ABCD$ and $BCDE$.

Within the framework of the above structure, the counting scheme is simple: Read in the snakes corresponding to the leaf itemsets, one TID at a time. The reading of the leaves is co-ordinated so that the TIDs for all the leaf snakes are generated in sorted order. In the course of counting, the intermediate nodes are marked with TIDs, and the last TID with which they have been marked is retained.

The counting starts from the leaves which generate a stream of TIDs. Once a TID is read in for a leaf, all its parents with this TID are updated. If a parent has been marked with a smaller TID, then it is simply marked with this new TID instead. However, if it is already marked with the same TID, its counter is incremented, and its parents are in turn updated with this TID. Intuitively, this corresponds to generating the subset-snakes of a candidate *on-the-fly*, and intersecting them at the node. The upward-propagation of the updates at a node correspond to that node's participation in further intersections.

Figure 7.3 gives a pictorial example of the counting scheme.[4] Here, the snakes AB, AC and AD are being read in from disk. They are read in one TID at a time, and in sorted order. The first update is from the snake AD upwards – all its parents are marked with the TID 2. In the next step, AB is read, and the candidate ABC updated with the TID 3. The third step involves reading in AC's TID, and updating the candidates ABC and ACD with this TID. At this step, since ABC has already been marked with the TID 3, its counter is incremented, and the update passed up to the candidate $ABCD$. At the same time, since ACD is marked with a smaller TID (2), it is simply marked with 3.

The above mechanism for counting minimizes the number of updates required and performs them in an *on-demand* manner, thereby not incurring the expensive subset-detection techniques of the hash-tree approach.

7.3.2 Delayed Write

We implement a *delayed write* mechanism that substantially cuts down upon the number of snakes written – the snakes we write are only those snakes that are actually used for subsequent computations.

For example, while counting the $2i$-length candidates using the i-length snakes, we do not know which ones of the $2i$ candidates are large, and which snakes will be used to generate subsequent itemsets. Writing out all the candidate snakes to disk can be very expensive and wasteful as well. Hence we do not write *any* $2i$-length snake, but instead dynamically regenerate only the required snakes in the next DAG-merge step. For this purpose, we

[4]The DAG is again only partially shown.

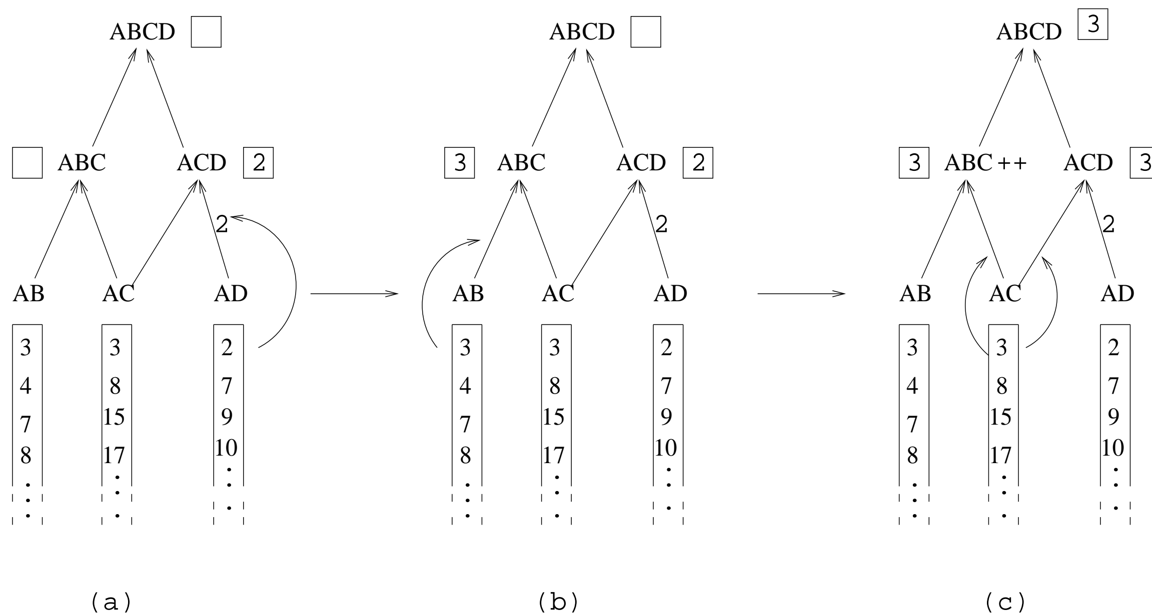

Figure 7.3: **The DAG counting scheme:On-demand intersection of snakes**

associate with each $2i$-candidate a pair of i-snakes that can be used to dynamically generate it in the next step. Correspondingly, in the current step, the leaf-snakes are regenerated using a pair of $i/2$-length snakes. These i-length snakes are written in this step for use in the subsequent pass.

This dynamic regeneration is easily incorporated into the DAG scheme by adding another level to the DAG corresponding to the $i/2$-length snakes. The modified DAG now looks like Figure 7.2, with the leaves being the $i/2$-length snakes that generate the i-length snakes. For example, though the conceptual picture shows the DAG leaves as the snakes AB, AC, BD, CD, CE, in reality each of these snakes are being generated dynamically from the snakes A, B, C, D and E. Also, at the end of this pass, these 2-length snakes are written away to disk. Now, the counting scheme remains essentially the same as before, except that it includes one more level of updates.

7.3.3 Write Cover and Write Pruning

As we have seen above, each top-level candidate (i.e., a candidate of length equal to the highest level of candidates being counted in the current step) is associated with a pair of leaf itemsets which can be used to regenerate it in the next step.

This selection of i-snake pairs is done in the following manner: After the merge step, we know which top-level candidates are large. So for each of these candidates, we do a tree traversal and collect all its leaves. Note that for this purpose we need to maintain *downward pointers* as well in the DAG. Further, we need pointers from each candidate to *all* its subsets, and not only two of them. Using these downward pointers (not shown in the diagram), we can collect all the i-length subsets of each top-level candidate. We then find a pair of leaves, the union of whose itemsets gives the candidate itemset. Their snakes can then be used in the next merge step to regenerate this top-level itemset on-the-fly.

This *write-cover* generation can in fact be done *before* the merge step itself. That is, we can find a pair of i-length itemsets for each top-level candidate even before counting it. By doing this we get a substantial benefit – we need to write to disk only those i-snakes that could possibly be used in generating a top-level candidate in the next step. Further, we have several choices for a covering pair for each top-level candidate. We can exploit this too, by choosing the write cover in an *overlapped* fashion – i.e., for each top-level candidate try as far as possible to use those i-snakes that are already being used to cover other itemsets. This will reduce the number of snakes written out even further.

As experimentally verified in Section 7.6, this technique can cut down substantially on the writes done at each step.

7.3.4 Focussed Writes

We have outlined above our techniques for choosing and minimizing the number of snakes to be written to disk. In this section we present an optimization that will make these snakes themselves much more sparse, and hence better compressed.

The key idea here is that the i-snakes being written to disk are *only* going to be used to re-generate the top-level candidates. Hence we need to add only those transaction-ids to the snake which contain the top-level itemset itself. Consider the following example: Suppose that snakes AB and CD are being written to disk in order to generate $ABCD$ later if it is required. Now if a transaction has the items A, B, D, E, we would add this transaction to the snake AB, but not to the snake CD. However, we can exploit the information that the snake AB is being used only to generate the snake $ABCD$, and hence this transaction is useless for that purpose. So we need not add this particular transaction to the AB snake.

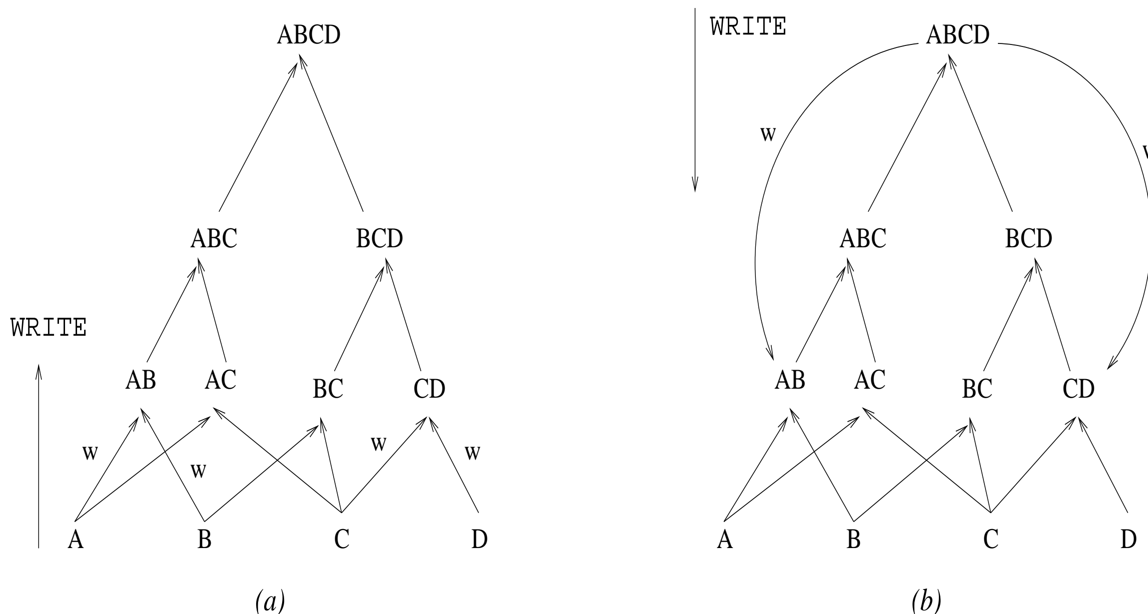

Figure 7.4: **Focussed writes to the i-length snakes**

The way we implement it is simple: The writes to the i-snakes, instead of being done in the *upward* stream of updates when the i-snake is detected in a transaction, is done *from the top*, when the top-level candidate has been detected in the transaction. This is represented pictorially in Figure 7.4.

As shown in the figure, the snakes AB and CD are normally appended to when they are generated by the intersections of A, B and C, D respectively. However, due to focussed writes, we actually append to the snakes when the top-level itemset $ABCD$ has been detected in a transaction through intersections of its subsets.

Note that this does not mean that the (trimmed) snake AB is equivalent to the snake ABC, because writes to AB may be done from several different top-level candidates that are covered by it. It is of course ensured that a particular TID is added only once to a snake.

7.4 Candidate Generation: ClusterGen

The predominant candidate generation algorithm in the mining literature is AprioriGen [2]. In AprioriGen, a *hash-tree* data structure is used for storing candidate itemsets and their running counts. A problem, however, is that this results in *scattering* "joinable" itemsets (i.e. itemsets with a common prefix upto their last element) across the hashtree. This means that identification of such itemsets becomes a computationally intensive task since all combinations have to be explicitly examined.

Another drawback of the AprioriGen approach is that it traverses the hashtree afresh even when multiple candidates either have common subsets or have subsets with a common prefix. So, for example, if {ABCDE}, {ABCDF} and {ABCDG} are potential candidates, then their subsets {BCDE}, {BCDF} and {BCDG} are searched for by traversing from the root in every instance although the {BCD} initial segment of the hash route is the same for all of them.

We present here a new candidate generation algorithm called **ClusterGen** that avoids the above kind of problems by employing the technique of *equivalence class* clustering [20]. Here, each itemset is represented by a lexically ordered list of items and frequent k-itemsets that share a common $k-1$-prefix are grouped together. Each such cluster is called an equivalence class.

```
SetOfItemsets ClusterGen( S_k ){
    Input: List of itemsets, each of length k, S_k.
    Output: List of candidate k+1-itemsets, C_{k+1}.

    for each itemset i in S_k do
        insert (i.prefix) into hashTable;
        insert (i.lastelement) into i.prefix -> extList;

    C_{k+1} = φ ;
    for each prefix P in the hashTable do {
        E = P -> extList ;
        for each element t in E do {
            newPrefix = P ∪ t ;
            remList = {i | i ∈ E and i > t} ;
            for each (k − 1) subset subPref of newPrefix do
                remList = remList ∩ (subPref -> extList) ;
            for each element q in remList do {
                candSet = newpref ∪ {q} ;
                C_{k+1} = C_{k+1} ∪ candSet ;
            }
        }
    }
    return C_{k+1} ;
}
```

Figure 7.5: **The ClusterGen Candidate Generation Algorithm**

The ClusterGen algorithm operates as follows: Given a set F_k from which to generate C_{k+1}, the items in each itemset of F_k are first ordered lexicographically. The itemsets are then partitioned into equivalence classes based on their $k-1$ length prefixes, and the class prefix is stored in a hash table. For each class, the last element of every itemset in the class is stored in a sorted list, called the extList.

For each prefix in the hash table, candidates are generated by combining the union of the prefix and any two items from the extList of the class. Once a candidate is formed, the presence of all its k-subsets is ascertained by searching across the relevant classes. This searching is very simple since the $k-1$ prefix of the subset that is being searched for indicates which class is to be searched.

We further optimize the above by ensuring that the two items for extension are such that the first is *lexicographically smaller* than the second. This has the desirable property that the prefix of the subsets of the candidates thus formed *will not* depend on the second extension item, which in turn means that all these subsets are shared and the same for each element in the extList. Hence, repeated searches for the same subsets can be avoided and they can be searched for *simultaneously*.

Example 7.4.1 Consider the itemsets $\{ABCD\}$, $\{ABCH\}$, $\{ABCM\}$ and $\{ABCR\}$, all of which belong to the same equivalence class g. The class prefix is $P_g =$ ABC and the associated extension list is $EL_g = [DHMR]$. To find the candidates extended by H, i.e., extended by the itemset $\{ABCH\}$, we extend using items that are larger than H, that is, $[MR]$. Searching for all subsets of these extensions means that we have to search for $\{ABCi\}$, $\{ACDi\}$, $\{BCDi\}$ where i is any item in $[MR]$. Since the prefix of these sets for various i is the same, they will belong to the same class if they exist and hence can be found together. Thus, if $\{ACDH\}$ exists it will be in the group with prefix $\{ACD\}$ and will be indicated by the presence of H in the corresponding item list. And similarly for $\{ACDR\}$. Hence we can select the group once and check concurrently for the existence of all the subsets instead of finding the class on a per-subset basis. □

Candidates are generated using the following procedure: An element, say e, is picked from the extList and the class prefix is extended with this element to get the $newPrefix$ itemset. The items in the extList that are greater than e are copied into another list called the *remnant* list, remList. The $k-1$ different classes are identified by enumerating the various $k-1$ subsets of $newPrefix$. For each of these classes, the associated subset exists if the last items are present in the extList. Hence an intersection of remList with extList gives the survivors after searching in this class. The survivors are reassigned to remList, and a new class identified and similarly searched. When all the classes have been considered, the items in remList are the ones that can extend the $newPrefix$ to give a candidate of size $k+1$.

The classes are processed in the increasing lexicographic order. Items for extending the class prefix to form the $newPrefix$ are also selected in a similar order. All this ensures that a class once processed will never have to be referred to again while processing the remaining classes.

The pseudocode of the ClusterGen algorithm is shown in Figure 7.5.

7.5 Previous Vertical Mining Algorithms

We now overview the vertical mining algorithms available in the literature. All these algorithms use either a tid-list format or a bit-vector format in their vertical layouts. In the following discussion and in the remainder of this chapter, we use the following notation:

I	Set of items in the database
D	Database of customer purchases
$\mid D \mid$	Cardinality of the database
$minSup$	User-specified minimum rule support
cnt_X	count of itemset X in D
sup_X	support of itemset X in D
F_k	set of frequent k-itemsets in D
F	set of frequent itemsets in D
C_k	Set of candidate k-itemsets in D

7.5.1 The Aggregate Algorithm

As mentioned earlier, the first work that we are aware of that considered the vertical mining approach is [13], where each item is associated with a tid-list. They presented an algorithm that we will refer to as the *Aggregate* algorithm wherein simple operations of set union and intersection are applied on the tid-lists of aggregations of items. Based on the support values of the outputs of these operations, inferences about the supports of various combinations of the items represented in the aggregate functions are made. For example, if the expression $(i_1 \cup i_2) \cap (i_3 \cup i_4)$ turns out to be infrequent, it can be inferred that none of the combinations $i_1 i_3$, $i_1 i_4$, $i_2 i_3$ or $i_2 i_4$ are frequent [13].

The union-intersection operation combination is recursively applied to construct a tree where the leaves are the individual items and the root is the union of all items. The tree is processed in a top-down manner, counting the supports of all itemsets except those that can be eliminated based on inferences like that mentioned above.

The advantage of this scheme is that it is easy to implement on a general-purpose DBMS since SQL supports the union and intersection operators. However, there are also several limitations: First, the inferencing is coarse-grained in that it may result in the counting of many itemsets that turn out to be infrequent. Second, the algorithm may not scale well to large databases or small minimum supports especially since the union operations may result in very large tid-lists. Third, they may end up making many more union and intersection operations[5]. Fourth, their performance study indicates that Aggregate's performance cannot compete with that of customized mining algorithms.

7.5.2 The DLG Algorithm

The *DLG* (Direct Large itemset Generation) algorithm was proposed in [16] for a bit-vector-based vertical representation. The algorithm operates in three phases: In the first phase, the database is scanned and a bit vector is created for each item; simultaneously, F_1 is computed. In the second phase, F_2 is computed and an "association graph" is constructed. The nodes of the graphs are the elements of F_1 and a directed edge from

[5]Each intersect or union operation causes a complete scan of the relevant TID-vectors on the disk.

item i_m to item i_n $(m < n)$ is introduced if the combination is in F_2. The data structure used to implement the association graph is a linked list.

The third and last phase iteratively generates all F_k $(k > 2)$ based on the association graph. This is done as follows: For each itemset in F_k, the last element of the k-itemset is used to extend the itemset into $k+1$-itemsets. Specifically, if $i_1 i_2 \ldots i_k$ is a large k-itemset and there is a directed edge from item i_k to item i_u, the k-itemset is extended to $i_1 i_2 \ldots i_k i_u$. The supports of these extensions is computed by intersecting (bit-wise ANDing) of the constitutent individual item bit-vectors. The algorithm terminates when F_k is empty.

The DLG algorithm has the following limitations: First, the graph construction and processing overhead may be rather high for databases with large F_2 – typically, F_2 has the largest number of elements among all the F_k(for reasonable *minSup*). Second, DLG assumes that bit vectors can completely fit into memory, and therefore scalability could be a problem for large databases. Third, the representation of the TID vector list for an item as a bit-vector can actually result in an explosion in the size of the database. For example, it the run of 0s is more than the word length, then we have actually inserted an extra word into the database which was not present in the horizontal representation. Many such long runs, indeed as are expected, would result in the size of the data set increasing. Fourth, the ANDing operation is done *afresh* for each itemset, and therefore there may be considerable repetition of previously performed work. For example, computing the support of $i_1 i_2 i_3 i_4$ requires a 4-way ANDing even if $i_1 i_2$ and $i_3 i_4$ have been previously computed. Overall, the entire mining process requires $\sum_{k=2}^{L} |C_k| * k$ intersection operations, where L is the length of the longest maximal frequent itemset.

[10] suggest an improvement for the intersection operations by decomposing the the operation to share these operations across itemsets having common prefixes. However, the number of intersections required is now $\sum_{i=2}^{k} d_j^k$, where d_j^k is the number of distinct j prefixes in C_k. This sum is bounded by the naive approach from above and potentially needs multiple scans for the same item bit-vector at each level.

7.5.3 The Eclat Algorithm

The *Eclat* (Equivalence CLAss Transformation) algorithm was proposed in [20] for vertical databases with a tid-list format. The algorithm is based on the concept of *equivalence class clustering.* The main idea here is that the itemsets in each F_{k-1} are partitioned into equivalence classes based on their common $k-2$ length prefixes, that is, the equivalence class $a \in F_{k-2}$ is given as
$S_a = [\mathbf{a}] = b[k-1] \in F_1 | a[1:k-2] = b[1:k-2]$
Candidate k-itemsets are generated from itemsets within a $k-2$ equivalence class by joining all $\binom{|S_i|}{2}$ pairs, with the class identifier as the prefix. Note that candidate itemsets produced by one equivalence class are independent of those produced by another, and hence the algorithm represents a "classify and conquer" approach.

While the above policy is used for all k $(k > 2)$, a special approach is taken for computing F_2. This is because, as mentioned earlier in Section 7.1, the direct method of computing the supports of all $\binom{|F_1|}{2}$ pairs is expected to be extremely expensive. Therefore, a *pre-processing* step is used to gather the counts of *all* 2-itemsets above a user-specified lower bound.

They have also considered an alternative wherein the horizontal format is used for the initial passes of the database and the vertical format is used thereafter. But this approach too has its own problems: First, the vertical format is not consistently used resulting in more complex code. Second, it is not clear as to when to move from the horizontal format to the vertical format. Third, generating the vertical format has to be done anew each time the database is mined, perhaps increasing the cost substantially.

However, this algorithm has the following limitations: First, Zaki et. al. use the time taken after computing F_2 for making the performance comparisons. The pre-processing step itself may be very expensive, especially if the number of items is large as is true for many real-life datasets. Further, it is claimed that this information is invariant during the life of the database and therefore the cost can be amortized over the number of times the data is mined. This assumes, however, a *static* database format whereas in practice historical databases are typically periodically augmented with the latest set of data leading to changes in database contents between successive minings of the database. Third, expecting the user to be able to choose a lower bound on all future mining support levels is perhaps unrealistic. Fourth, the ability of the algorithm to scale with database size has not been conclusively evaluated. Finally, the direct method of computing the supports of the candidates by performing $2 * |C_k|$ intersections at the kth level can be expensive.

7.5.4 The Clique Algorithm

The *Clique* algorithm[20] is a refinement of the Eclat algorithm wherein clique-based clustering is used instead of prefix-based clustering. The main idea here is to filter the equivalence class by incorporating the observation that for any frequent m-itemset, all its k-subsets ($k < m$) must be frequent. This fact corresponds, in graph-theoretic terms, to the following: If each item is a vertex in the hypergraph, and each k-subset an edge, then the frequent m-itemset must form a k-uniform hypergraph clique. By using this observation, we obtain *smaller* sub-lattices than those obtained in Eclat, resulting in less mining effort.

In addition to all the limitations mentioned above for the Eclat algorithm, the Clique algorithm has the following additional drawback: The enumeration of the maximum cliques can be computationally expensive, especially when the equivalence class graph is not sparse. In fact, for general graphs, the maximal clique decision problem in NP-Complete.

7.5.5 The MaxEclat Algorithm

In the Eclat algorithm described above, a bottom-up search strategy is used to enumerate all frequent itemsets. Variants of this algorithms were also proposed in [20] by substituting different search strategies. In particular, they proposed the **MaxEclat** algorithm, wherein a *hybrid* top-down cum bottom-up strategy is used.

The basic idea behind the hybrid approach is to quickly determine the "true" maximal itemsets, by starting with a single element from a cluster of frequent k-itemsets, and extending this by one more item until an infrequent itemset is generated. This comprises the top-down phase. In the bottom-up phaase, the remaining elements are combined with the elements in the first set to generate all the additional frequent itemsets.

To decide which elements of the cluster should be combined, the itemsets in the cluster are first sorted in decreasing order of their support. Then the element with the maximum support is extened with the next element in the sorted order and so on. The intuition here is that the larger the support of an itemset, the more likely that it will be part of a larger itemset.

The MaxEclat algorithm has similar limitations to those mentioned for its bottom-up version described above.

7.5.6 The MaxClique Algorithm

The MaxClique algorithm is a refinement of the basic Clique algorithm [20]. In the Clique algorithm, as described above, a *bottom-up search* strategy is used to enumerate all frequent itemsets. This search strategy is modified in the MaxClique algorithm to instead use a *hybrid* top-down cum bottom-up approach. The basic idea behind the hybrid approach is to quickly determine the "true" maximal itemsets, by starting with a single element from a cluster of frequent k-itemsets, and extending this by one more item until an infrequent itemset is generated. This comprises the top-down phase. In the bottom-up phase, the remaining elements are combined with the elements in the first set to generate all the additional frequent itemsets.

Limitations

The MaxClique algorithm uses lattice-theoretic concepts to gain processing efficiency, and [20] has reported an experimental study indicating considerable improvement over previous horizontal algorithms such as Apriori. However, the design of the MaxClique algorithm and its performance evaluation in [20] have a variety of limitations, some of which are described below:

- The enumeration of the maximum cliques can be computationally expensive, especially when the equivalence class graph is not sparse. In fact, for general graphs, the maximal clique decision problem is NP-Complete.
- Expecting the user to be able to choose a lower bound on all future mining support levels is perhaps unrealistic; this problem is further exacerbated in a shared environment, wherein a lower bound has to be chosen across all potential users of the system.
- It is claimed that the pre-processing information is invariant during the life of the database and therefore the cost can be amortized over the number of times the data is mined. This assumes, however, a *static* database format whereas in practice historical databases are typically periodically augmented with the latest set of data leading to changes in database contents between successive minings of the database.

- Only the time taken to complete the mining process *after* computing F_2 is used for making the performance comparisons. However, the pre-processing step itself may be very expensive, especially if the number of items is large as is true for many real-life datasets, and further, the time taken for the pre-processing may vary across algorithms.

- The TID-lists in a cluster are those corresponding to 2-length itemsets. However, they are not stored on disk, but are computed *on demand*, from the corresponding 1-item TID lists. This means that a single TID-list may be read several times. This problem has been mentioned in [17], where it is suggested that it may be possible to reorder the processing of cliques, or to process several cliques together. However there is no definite method described, and it is not clear how the batch-processing, or ordering of scans, can be reorganised to both reduce repeated scans as well as fit all TID lists into memory.

- For purposes of computing supports, entire TID-lists of items in a clique or cluster are read into memory. This may not be feasible in two cases – when the TID-lists are long, or when there are several TID-lists in a cluster. Our experiments in Section 7.6 show examples of this situation.

  This problem has also been anticipated in [17]. The proposed solution is to recursively decompose the partitions in terms of longer prefixes, until all TID-lists in a clique fit into memory. However, this would lead to much larger overlap of 1-items across cliques, and the disk reads would correspondingly multiply, leading to further performance degradation.

- The ability of the algorithm to scale with database size has not been conclusively evaluated in the light of the above drawbacks. In all the experiments described in [20] the *entire database could fit into main memory* – their experimental platform had 256 MB of memory and the largest database considered was 240 MB, while the typical case was of the order of 10 MB.

7.6 Performance Study

We conducted a detailed performance study to assess the performance characteristics of the VIPER algorithm. In particular, we compared its performance against that of MaxClique, which as mentioned before is the best vertical mining algorithm currently known. Our experiments covered a range of database and mining workloads. In particular, we conducted exactly the same experiments as those described in [20] – the only difference is that the database sizes we considered included those that were *significantly larger* than the available main memory. A range of rule support threshold values between 0.25% and 2% were considered in these experiments. The performance metric in all the experiments is the *total execution time* taken by the mining operation. (This total execution time includes the pre-processing time in the case of the MaxClique algorithm.)

Due to space limitations, we only show a few representative experiments here – the others are available in [14].

7.6.1 Database Generation

Parameter	Meaning	Default Values
N	Number of items	1000
$\mid T \mid$	Mean transaction length	10, 20
$\mid L \mid$	Number of potentially large itemsets	2000
$\mid I \mid$	Mean length of potentially large itemsets	4, 7
$\mid DB \mid$	Number of transactions in database DB	1M, 5M, 10M, 25M

Table 7.1: **Parameter Table**

The databases used in our experiments were synthetically generated using the technique described in [2] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator and their default values are described in Table 7.1.

7.6.2 Implementation Details

The VIPER algorithm was written in C++. For MaxClique, we used the original code, kindly supplied to us by Prof. Zaki. The experiments were conducted on a SGI Octane 225 MHz workstation running Irix 6.5. The

amount of main memory available was 128 MB, while the databases were resident on a local 4 GB SCSI disk. For the databases mentioned in Table 7.1 with parameters $T = 10$ amd $I = 4, 7$, the associated database sizes are approximately 100MB (2M tuples), 250MB (5M tuples), 500MB (10M tuples) and 1.25 GB (25M tuples). The other database used in the experiments presented below was T20I4D10M, of size approximately 900MB.

7.6.3 Baseline Experiment

In our first experiment, we evaluated the performance for the T10I4 database across a range of database sizes. The results of this experiment are shown in Figures 7.6a–d, which correspond to databases with 2M, 5M, 10M and 25M transactions, respectively. As is shown in these graphs, VIPER performs consistently better than MaxClique. Also, the difference in performance shows a marked increase with the size of the database. For example, in the database with 25M transactions (Figure 7.6d), we see a performance ratio of over 3 between MaxClique and VIPER at support = 0.25% whereas their performance at that support is comparable for the 2M database (Figure 7.6a).

Overall, the results of this experiment indicate that VIPER is better suited for handling large databases.

Performance Scalability

In Figure 7.6e, the results of Figures 7.6a–d are combined to show the scalability comparison between VIPER and MaxClique for the support value of 0.25%. In this figure, the database size is shown relative to the 2M database, while the running times have been normalized with respect to the running time of MaxClique for the 2M database.

The results show that VIPER has excellent scalability with database size – the ratios of time taken versus database sizes are nearly equal. This conforms to our expectation – since the computation cost in VIPER is on a per-transaction basis, it should scale linearly with increase in the number of transactions. In contrast, MaxClique shows comparatively much worse scalability.

Resource Usage

We now move on to present some results regarding the resource usage of the two algorithms.

Figure 7.6 (f) shows the disk usage on the T10I4D10M database over a range of supports. We see here that VIPER's disk traffic is consistently *less* than that of MaxClique, highlighting the effect of VIPER's optimizations to reduce disk read/writes – specifically compression, delayed writes, single-scan of snakes and write pruning. MaxClique, on the other hand, reads in a TID-list corresponding to a single item several times, depending upon the number of cliques in which it is present. As a result, the disk reads increase dramatically at low support where there is considerable overlap between clusters. In this regard, VIPER's approach of a single-scan of each snake as well as writing to disk after each pass appear to be the preferred choice.

Another feature of VIPER is that its main memory usage is independent of the database size. Memory is only required to store the DAG of candidates and the read and write snake buffers. Further, the data structures used also occupy very little memory. As an example, VIPER's peak memory usage for all the workloads considered in Figure 7.6 a–d is 4 MB. In contrast, MaxClique's memory usage depends on the lengths of the TID-lists and the number of TID-lists in each clique. Accordingly, MaxClique can use as much as 2.5 MB for the database with 1M transactions, and upto 23 MB for the database with D=10M.

7.6.4 Sensitivity Analysis

In this section we present two more experiments that show VIPER's consistent performance over different datasets. Figures 7.7.1 and 7.7.2 show two experiments where the parameters I and T have been varied respectively. Figure 7.7.1 shows performance comparison for a T10I7D10M database, while Figure 7.7.2 shows the performance on a T20I4D10M database. The results again show VIPER performing consistently better than MaxClique. Another point to be noted here is that the 0.25% support evaluation for the T20I4D10M database could not be conducted for MaxClique since it needed over 500MB of memory. This memory usage arises because the number of TID-lists in a clique is very large, and the combined memory requirement for the TID-lists of a clique exceeded the available memory.

7.6.5 Comparison with Horizontal Algorithms

In this section we show the comparison between VIPER and Apriori and also with MaxClique. Figure 7.8 shows the comparison between VIPER, Apriori amd Maxclique. Figure 7.8(a) shows the execution time taken by these algorithms at different supports for a T10I4D10M database while Figure 7.8(b) shows the scaleup of the algorithms at a support of 0.25 percent.

In Figure 7.8a, we first notice that Apriori, although doing similarly to VIPER at high supports, has a steep degradation in performance at lower supports. This is because it has to make several scans over the entire database at these lower supports. Another interesting observation here is that Apriori actually outperforms MaxClique over the entire higher support region. This might seem to be at odds with the results reported in [20], wherein MaxClique always beat Apriori by substantial margins. The difference here is that the pre-processing times are included in our execution time computations – these times were ignored (for all algorithms) in [20]. However, the pre-processing step takes different amounts of time for different algorithms – in Apriori, only the "join" of F_1 is counted in the second pass, whereas in MaxClique the "join "of $\mathcal{I}$ (the set of all items in the database) is counted in the second pass, and typically $\mathcal{I} \gg \mathcal{F}_\infty$ – this has a major impact at higher supports, where the preprocessing step takes up most of the overall execution time.

Moving on to Figure 7.8b, we observe that Apriori also has a linear scaleup like VIPER, although its absolute performance is worse than that of VIPER at lower supports.

7.6.6 Compression and Pruning

We now present supporting statistics showing the contributions of the several optimizations implemented in VIPER.

Firstly, our compression technique gives us substantial compression over the original database representation. As an example, the snake representation of the T10I4D10M database gave a compression ratio of 2.6 over the original database.

Secondly, the write pruning also results in considerable savings. Consider the following extract from VIPER's output.

```
Database:  t10i4d10m, supp:  0.25, Starting level = 2
Candidates at level3:  3458
Candidates at level4:  2402
# 2-snakes generated:  2504
# 2-snakes written:  1474
```

While counting levels 3 and 4, a total of 2504 2-length snakes are dynamically regenerated. However, only 1474 of them are written back to disk, as potential covers for the 2402 candidates at level 4.

7.7 Conclusions

In this chapter, we have addressed the problem of designing scalable high-performance vertical mining algorithms. We presented a new algorithm called VIPER that attempts to aggresively materialize the benefits offered by the vertical data layout. VIPER includes a novel DAG-based snake merging scheme which allows for the candidates at multiple levels to be efficiently counted in a single pass. Several optimizations have been incorporated in VIPER including snake compression, delayed writes, write prunes and cluster-based candidate generation. Our experimental results demonstrated that VIPER consistently performs much better than MaxClique, the best vertical mining algorithm current available – further, it has the added advantage of excellent scalability as well as low disk and memory usage.

In our future work, we propose to explore the issues of *parallel mining*, an area that is greatly facilitated by vertical partitioning of data. Other issues that need to be researched are the partitioning of itemsets to handle large number of items in the database, and development of a hybrid algorithm that can take advantage of the memory available to do in-memory computations where possible, while at the same time retaining robustness and minimizing computations.

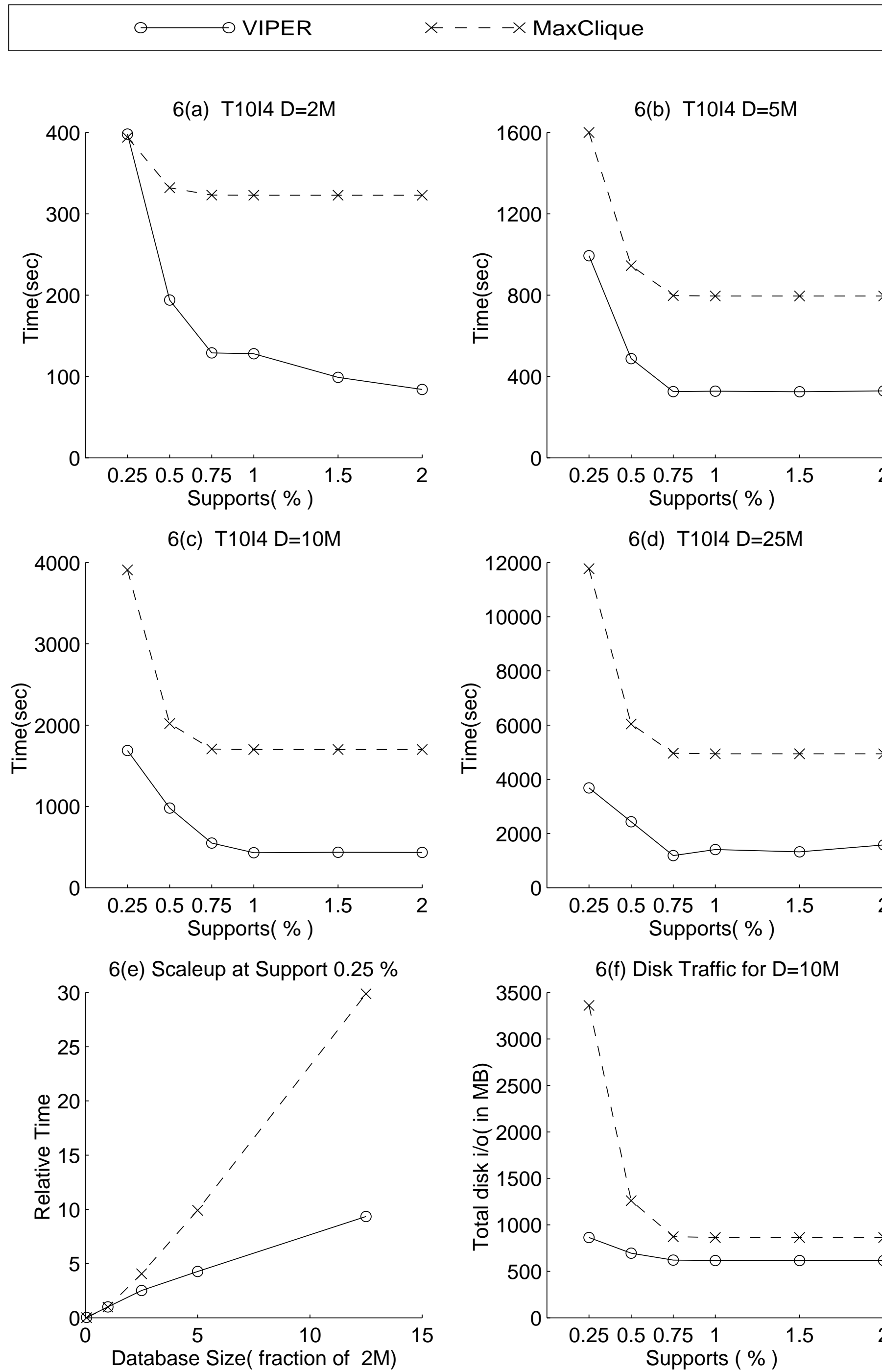

Figure 7.6: **Experimental Results for T = 10, I = 4**

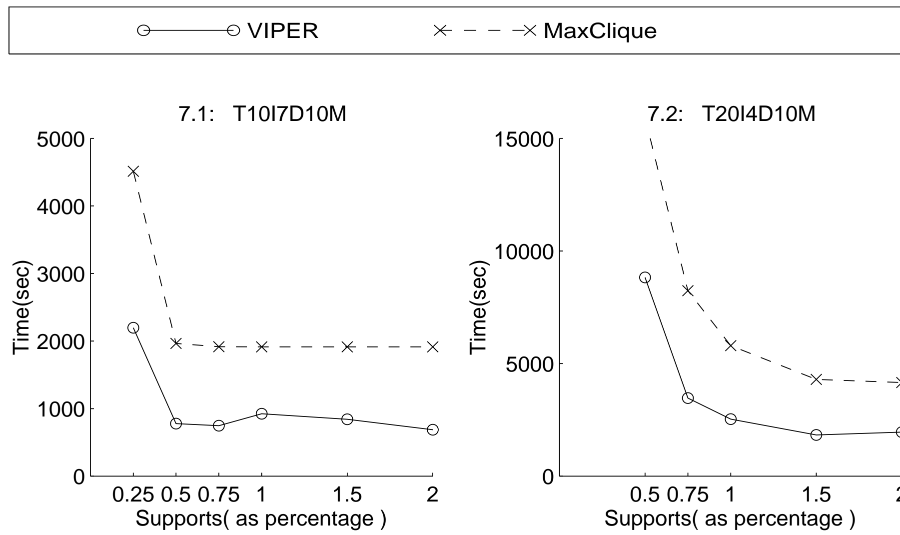

Figure 7.7: **Sensitivity Analysis**

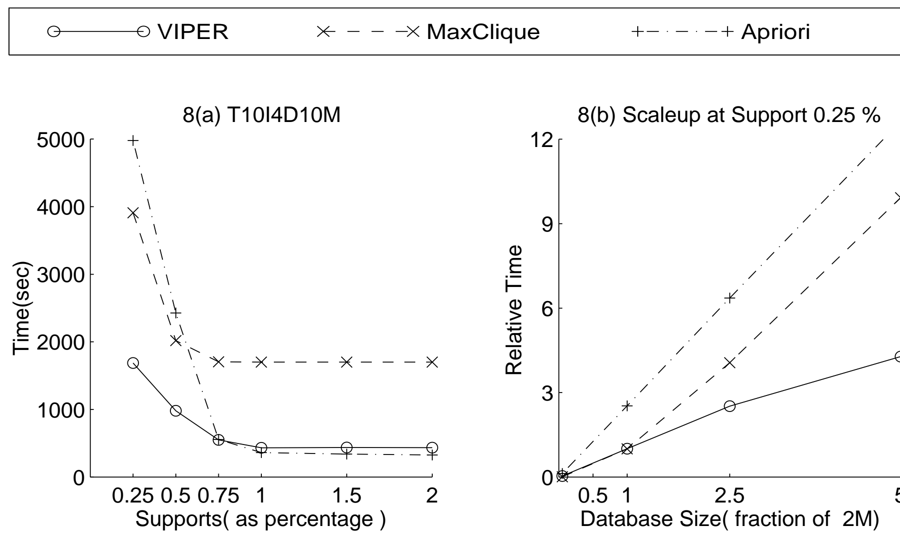

Figure 7.8: **Comparison with Horizontal algorithms**

Chapter 8

Implementation of MINTO

In the previous chapters, we discussed in detail the various mining algorithms that have been incorporated in the MINTO mining tool. We now discuss in detail the architecture and implementation of MINTO.

In MINTO, users can choose any of the different types of mining algorithms, based on their individual requirements. There are two user interfaces: (a) A graphical interface intended for managers and casual users that is extremely easy to use; and (b) A SQL like query interface for experienced software engineers. The SQL interface is much faster and easier to interface with programs written in other languages. Moreover, it allows MINTO to be run even on systems that do not support graphical interfaces.

MINTO is developed on a Unix platform using C++. The query interface is built using lex and yacc utilities. The GUI interface makes use of X and motif libraries. The whole effort involved about 15000 lines of C++ code. Three types of queries are supported in MINTO:

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

8.1 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. This organization is shown in Figure 8.1.

Data Structures All the major data structures used in the tool are present in this module. It contains all the data structures including the hash tree and the 2-D arrays for counting. 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.

8.1.1 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 8.2. 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 8.2, 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.

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.

The following procedure is used to find out 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 the Partition algorithm, we maintain a list of transaction ids with the itemsets with single elements.

8.1.2 Algorithm Module

MINTO has five boolean association rule algorithms, namely AIS, APRIORI, APRIORITID, DHP, and PARTITION, apart from Cumulate and QAR for generalized and quantitative association rules, respectively. It also includes SAMPLING, a mining algorithm based on sampling tehcniques, as also the CountDistribute parallel minng algorithm. Further, a variety of incremental mining algorithms including FUP, Borders, TBAR are implemented, apart from vertical mining algorithms such as Eclat, Clique, MaxEclat and MaxClique.

Apart from the above previously proposed algorithms, MINTO also incorporates our new TWOPASS, DELTA, VIPER and RULEGEN algorithms for first-time mining, incremental mining, vertical mining, and rule-generation, respectively.

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

8.1.3 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 8.3, 8.4 and 8.5.

8.1.4 Query Interface

We have also provided a SQL like query interface to MINTO. This grammar is implemented using lex and yacc.

Syntactic Queries

Syntactic queries are specified as follows:

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

For example,

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

Support Queries

Support queries are specified as follows:

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

For example,

select rule
from vegetable database
where support = 0.2 confidence = 0.6
using TWOPASS

Support and Syntactic Queries

For combinations of support and syntactic constraints, the query specification is as follows:

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

For example,

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

8.1.5 Interaction with the DBMS

MINTO exists as an application on top of a DBMS and therefore in order to interact with the DBMS back-end and access the data, an integration module is incorporated. Only this unit will be dependent on the underlying DBMS but not any other modules since the integration module presents a uniform interface to the other modules.

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 scanning through 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.

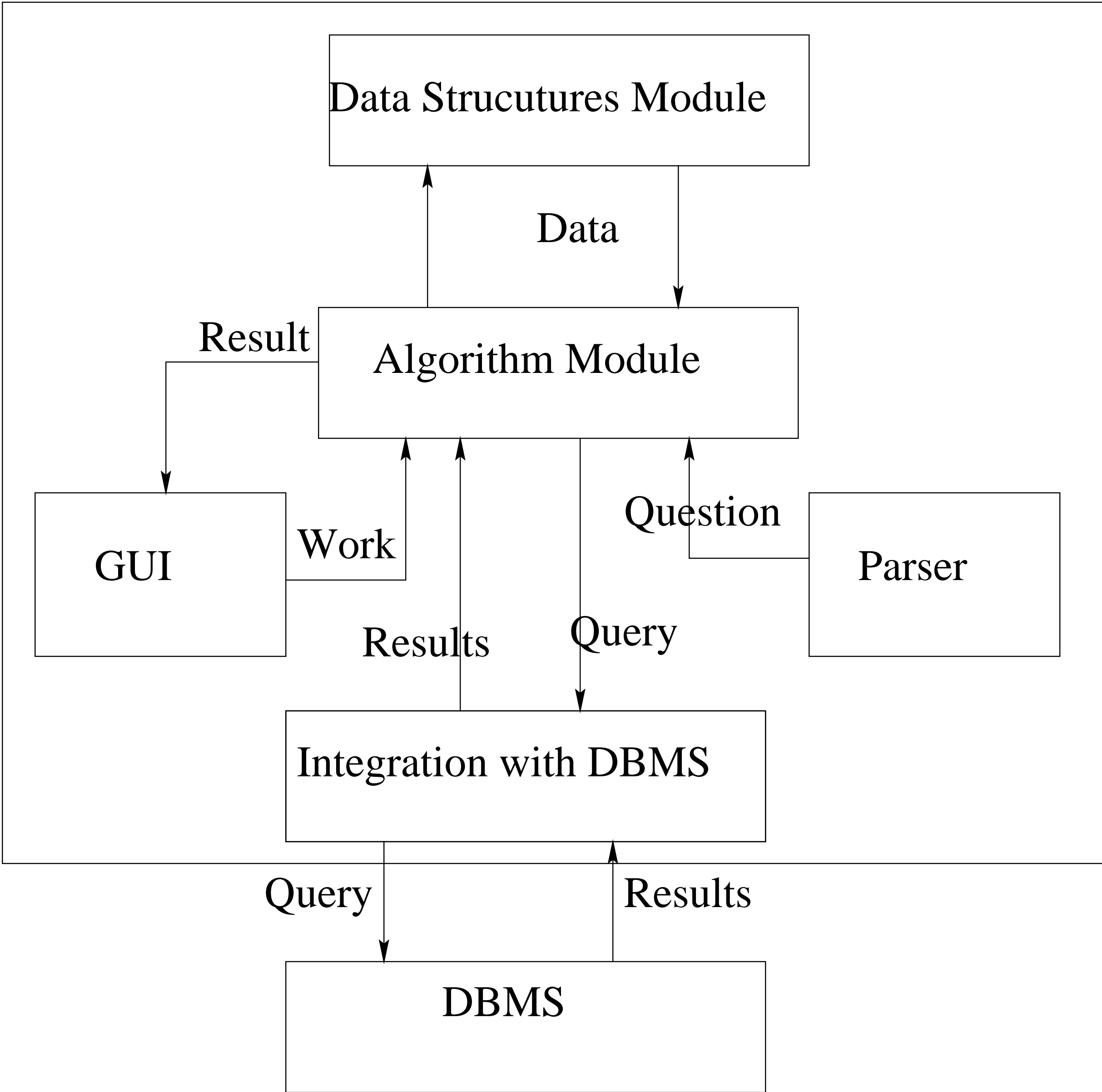

Figure 8.1: Organization of Modules in MINTO

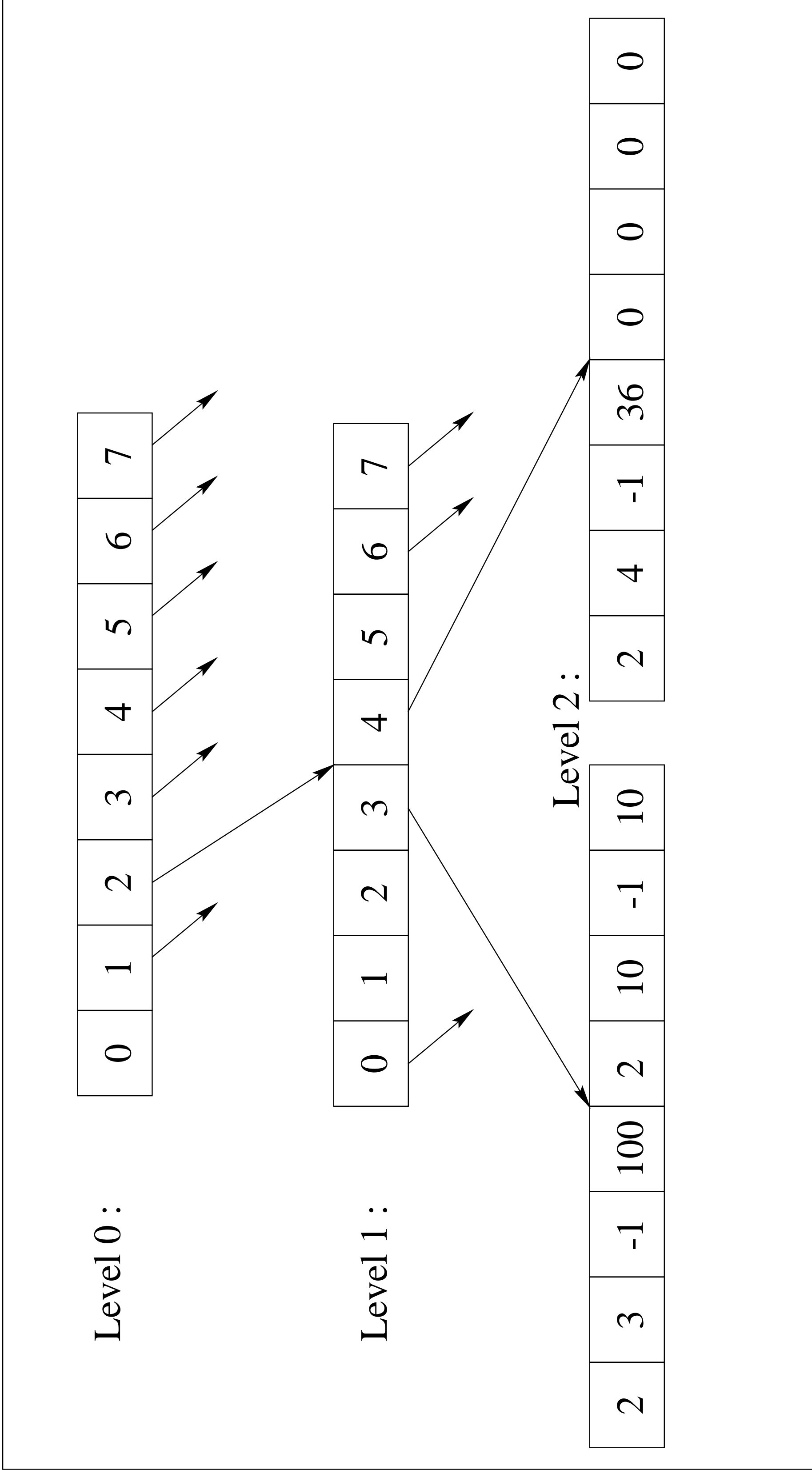

Figure 8.2: An example hash tree

Figure 8.3: Frontview of MINTO

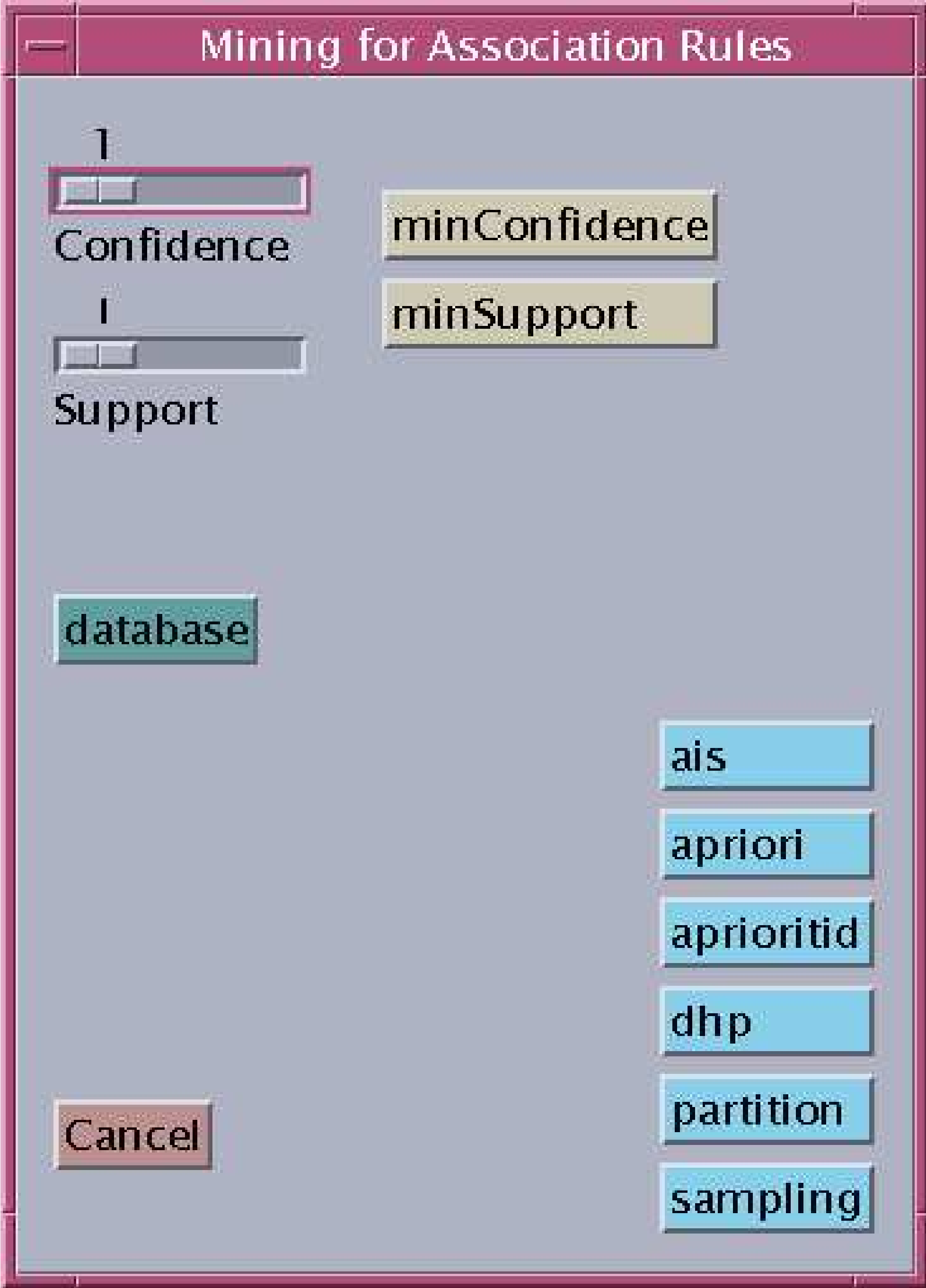

Figure 8.4: Form 1

Finding support and confidence of rule

To find: Support

Rule :

Potatoes => Onions

database

Cancel

Submit

Figure 8.5: Form 2

Chapter 9

Conclusions

In this project, we have attempted to design a software tool for efficiently supporting mining huge historical manufacturing databases. Apart from providing a variety of algorithms that have been proposed in the literature in an integrated and user-friendly manner, we have also designed and implemented a set of new algorithms: **TWOPASS**, **DELTA**, **VIPER** and **RULEGEN**, that are shown to perform significantly better than the state-of-the-art. We recommend the MINTO tool with the above algorithms to manufacturing personnel since they could considerably reduce the tremendous computational expense normally associated with mining operations.

9.1 Future Work

The MINTO system can be extended to incorporate extra functionalities that would increase its applicability to a wider variety of manufacturing systems. For example, it would be useful to include algorithmic support for other kinds of mining patterns such as classification, comparison, sequences, clustering, etc. Second, a middleware tool that converts the association rules discovered by the mining process into tangible business decisions would be extremely helpful for upper-level management. Third, extending the mining strategies to distributed databases would be essential since many manufacturing organizations store their data distributed across a variety of sites. Finally, it would be interesting to evaluate the complexity and the performance impact of integrating the mining tool, which currently operates as an application, into the kernel of a database engine.

Bibliography

[1] R. Agrawal, T. Imielinski and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *Proc. of 22nd SIGMOD Conf.*, May 1993.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proc. of 20th VLDB Conf.*, September 1994.

[3] D. Cheung, J. Han, V. Ng, A. Fu and Y. Fu, "A Fast Distributed Algorithm for Mining Association Rules", *Proc. of 4th PDIS Conf.*, December 1996.

[4] D. Cheung, J. Han, V. Ng and C. Wong, "Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique", *Proc. of 12th ICDE Conf.*, February 1996.

[5] D. Cheung, S. Lee and B. Kao, "A General Incremental Technique for Maintaining Discovered Association Rules" *Proc. of 5th DASFAA Conf.*, April 1997.

[6] R. Feldman, A. Amir, Y. Aumann, A. Zilberstein and H. Hirsh, "Incremental Algorithms for Association Generation", *Proc. of 1st PAKDD Conf.*, February 1997.

[7] R. Feldman, Y. Aumann, A. Amir and H. Mannila, "Efficient Algorithms for Discovering Frequent Sets in Incremental Databases", *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[8] E. Han, G. Karypis and V. Kumar, "Scalable Parallel Data Mining for Association Rules", *Proc. of 26th SIGMOD Conf.*, June 1997.

[9] S. Lee and D. Cheung, "Maintenance of Discovered Association Rules: When to Update?", *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[10] S. Thomas, S. Bodagala, K. Alsabti and S. Ranka, "An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases", *Proc. of 3rd KDD Conf.*, August 1997.

[11] H. Toivonen, "Sampling Large Databases for Association Rules", *Proc. of 22nd VLDB Conf.*, September 1996.

[12] S.W. Golomb. Run-length encoding. *IEEE Trans. on Information Theory*, 12(3), July 1966.

[13] M. Holsheimer, M. Kersten, H. Mannila and H. Toivonen. A perspective on databases and data mining. In *Intl. Conf. on Knowledge Discovery and Data Mining*, August 1995.

[14] P. Shenoy, J. Haritsa, S. Sudarshan, M. Bawa, G. Bhalotia and D. Shah. Vertical mining of large databases. Technical report, DSL, Indian Institute of Science, 1999.

[15] A. Savasere, E. Omiecinski and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of Intl. Conf. Very Large Databases (VLDB)*, 1995.

[16] S-J. Yen and A.L.P. Chen. An efficient approach to discovering knowledge from large databases. 1996.

[17] M. J. Zaki. *Scalable Data Mining for Rules.* PhD thesis, University of Rochester, July 1998.

[18] M. J. Zaki, M. Ogihara, S. Parthasarathy and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1996.

[19] M. J. Zaki, S. Parthasarathy and W. Li. A localized algorithm for parallel association mining. In *ACM Symposium Parallel Algorithms and Architectures*, June 1997.

[20] M. J. Zaki, S. Parthasarathy, M. Ogihara and W. Li. New algorithms for fast discovery of association rules. In *Intl. Conf. on Knowledge Discovery and Data Mining*, August 1997.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

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

[63] 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.

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

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

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

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

[68] 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.

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

[70] 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.

[71] 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.

[72] 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.

[73] Y. Aumann, R. Feldman, O. Lipsttat and H. Manilla, "Borders: An Efficient Algorithm for Association Generation in Dynamic Databases", *Journal of Intelligent Information Systems*, April 1999.

[74] R. Agrawal and R. Srikant, "Mining Sequential Patterns", *Proc. of 11th ICDE Conf.*, March 1995.

[75] L. Breiman, J. Friedman, R. Olshen and C. Stone, "Classification and regression trees", *Belmont: Wadsworth*, 1984.

[76] R. Agrawal and J. Stafer, "Parallel Mining of Association Rules: Design, Implementation and Experience", *Tech. Report RJ10004*, IBM Almaden Research Center, January 1996.

[77] D. Cheung, J. Han, V. Ng and C. Wong, "Maintenance of discovered association rules in large databases: An incremental updating technique", *Proc. of 12th ICDE Conf.*, February 1996.

[78] D. Cheung, S. Lee and B. Kao, "A general incremental technique for maintaining discovered association rules", *Proc. of 5th DASFAA Conf.*, April 1997.

[79] R. Feldman, Y. Aumann, A. Amir and H. Mannila, "Efficient algorithms for discovering frequent sets in incremental databases", *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[80] M. Garofalakis, S. Ramaswamy, R. Rastogi and K. Shim, "Of Crawlers, Portals, Mice and Men: Is there more to Mining the Web?", *Proc. of 28th SIGMOD Conf.*, May 1999.

[81] C. Hidber, "Online Association Rule Mining", *Proc. of 28th SIGMOD Conf.*, May 1999.

[82] J. Park, M. Chen and P. Yu, "An effective hash-based algorithm for mining association rules", *Proc. of 24th SIGMOD Conf.*, May 1995.

[83] V. Pudi and J. Haritsa, "Incremental Mining of Association Rules", *Tech. Report*, DSL, Indian Institute of Science, 1999.

[84] H. Toivonen, "Sampling large databases for association rules", *Proc. of 22nd VLDB Conf.*, September 1996.

[85] Y. Zhuge, H. Garcia-Molina, J. Hammer and J. Widom, "View Maintenance in a Warehousing Environment", *Proc. of 24th SIGMOD Conf.*, May 1995.