Efficient Discovery of Concise Association Rules from Large Databases

A Thesis
Submitted for the Degree of
Doctor of Philosophy
in the Faculty of Engineering

By
Vikram Pudi

Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

April 2003

Abstract

Association rules are interesting correlations among attributes in a database. These rules have many applications in areas ranging from e-commerce to sports to census analysis to medical diagnosis. The discovery of association rules is an extremely computationally expensive task and it is therefore imperative to have fast scalable algorithms for mining these rules. In this thesis, we present efficient techniques for discovering association rules from large databases and for removing redundancy from these rules so as to improve the quality of output. We also handle growing databases.

Specifically, we present three new algorithms: (1) ARMOR: This algorithm discovers association rules from databases and requires at most two database scans. We empirically show its performance to be within a factor of two of an unachievable lower bound. (2) g-ARMOR: This is an extension to ARMOR that is designed to remove redundancy from association rules during the mining process. This is especially important because the number of association rules generated in typical mining operations runs into the tens of thousands. g-ARMOR results in an orders of magnitude reduction in the number of rules thereby making the mining output comprehensible to end users. (3) DELTA: This algorithm incrementally mines evolving databases. It utilizes previous mining results to efficiently mine the current database after it has been updated with fresh data. It also handles situations where the mining specifications over the current database differ from those used over the original database, a common occurrence in practice.

Publications

- "Quantifying the Utility of the Past in Mining Large Databases"
  V. Pudi and J. Haritsa
  Information Systems, Elsevier Science Ltd., July 2000

- "How Good are Association-Rule Mining Algorithms?" (poster)
  V. Pudi and J. Haritsa
  Proc. of Intl. Conf. on Data Engineering (ICDE)
  San Jose, California, USA, February 2002

- "On the Efficiency of Association-rule Mining Algorithms"
  V. Pudi and J. Haritsa
  Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)
  Taipei, Taiwan, May 2002

- "Generalized Closed Itemsets: A Technique for Improving the Conciseness of Rule Covers" (poster)
  V. Pudi and J. Haritsa *Proc. of Intl. Conf. on Data Engineering (ICDE)*
  Bangalore, India, March 2003

- "Reducing Rule Covers with Deterministic Error Bounds"
  V. Pudi and J. Haritsa
  Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)
  Seoul, South Korea, May 2003

Acknowledgements

I would like to thank everyone who was instrumental in the creation of this work. In particular, I am deeply grateful to my research supervisor Prof. Jayant Haritsa, whose guidance is truly professional, whose enthusiasm is contagious and who is always ready to extend a helpful hand in times of need.

I am indebted to my friends who made my stay in the institute pleasant: Manjusha, Srikanta, Maya, Suresha, Kumaran, Nisha, Anurag, Prabodh and others. Surekha has been a great source of inspiration and support for me. Finally, I thank my parents and brother for their love and support that no amount of thanks can suffice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Consider a supermarket with a large collection of items. Typical business decisions that the management of the supermarket has to make include what to put on sale, how to design the store layout, what promotional strategies to consider, etc. Analysis of past sales data is a commonly used approach in order to improve the quality of such decisions. Until recently, however, only global data about the cumulative sales during some time period (a day, a week, a month, etc.) was available on the computer. Progress in bar-code technology has made it possible to store the so called *basket* data that stores items purchased on a per-transaction basis. Basket data type transactions do not necessarily consist of items bought together at the same point of time. It may consist of items bought by a customer over a period of time. Examples include monthly purchases by members of a book club or a music club.

Several organizations have collected massive amounts of such data. These data sets are usually stored on tertiary storage and are slowly migrating to database systems. Discovering associations between items, also known as *association rules*, enables such organizations to make informed business decisions. An example of such an association rule is: "30% of transactions that contain Surf washing powder and Rin detergent bar also contain Comfort fabric softener; 2% of all transactions contain all three of these

items together". The antecedent of this rule consists of Surf and Rin and the consequent consists of Comfort alone. Here 90% is called the *confidence* of the rule, and 2% the *support* of the rule. Both the antecedent and consequent may contain multiple items.

Given a market-basket database, it is desirable to find association rules that have high confidence and support (i.e. those satisfying a user-specified minimum confidence and minimum support). The confidence measure represents the strength of a rule or its likelihood of being true. The support of a rule represents its statistical significance (a rule with very low support is not statistically significant) and its applicability (a rule with high support is applicable in many transactions). Finding such association rules is valuable for many applications:

1. Cross marketing: Refers to suggesting customers to buy additional products based on those that they have already purchased.

2. Attached mailing: Refers to promotional offers that are attached to mails sent on a direct marketing campaign of some particular product.

3. Catalog design: Catalog pages that contain description of some product could, in addition, contain information regarding other products that are frequently purchased along with it.

4. Add-on sales: Refers to selling multiple products together at discounted rates.

5. Store layout: Items that are frequently purchased together could be placed near each other in the store so that customers would tend not to overlook them. Alternatively, they may be placed far away from each other, so that customers may pick up other items on the way.

Other Applications: Besides super-market basket analysis, association rule discovery has been applied in numerous other areas such as e-commerce, sports, analysis of census data and medical diagnosis. For example, in immunology it is often required to test a patient for sensitivity to various allergens. These tests are expensive and patients usually

have incomplete knowledge of possible allergens. Association rules such as "allergy to latex rubber usually co-occurs with allergies to banana and tomato" would be very valuable in deciding what allergens to test for.

The discovery of association rules is a computationally expensive task. Further, market basket databases are typically very large. It is therefore imperative to have fast scalable techniques for mining them. In this thesis, we present efficient techniques for discovering association rules from large databases and for removing redundancy from these rules so as to improve the quality of output. We also handle growing databases.

1.2 Data Mining

Association rule discovery is part of a larger field of study called *data mining* – a field that consists of techniques to automatically find interesting patterns and trends in large collections of data. In this Section, we provide a brief introduction to the broad area of data mining that has been extracted and summarized from [HK01].

Human capabilities of both generating and collecting data have been increasing rapidly in the last several decades. Contributing factors include the computerization of many business, scientific and government transactions, and advances in data collection tools ranging from scanned text and image platforms to satellite remote sensing systems. In addition, popular use of the World Wide Web as a global information system has flooded us with a tremendous amount of data and information. This explosive growth in stored data has generated an urgent need for new techniques and automated tools that can intelligently assist us in transforming the vast amounts of data into useful information and knowledge.

Knowledge Discovery in Databases (KDD) is the automated extraction of novel, understandable and potentially useful patterns implicitly stored in large databases, data warehouses and other massive information repositories. KDD is a multi-disciplinary field, drawing work from areas including database technology, artificial intelligence, machine learning, neural networks, statistics, pattern recognition, information retrieval, high-performance computing and data visualization.

Data mining is an essential step in the process of knowledge discovery in databases, in which intelligent methods are applied in order to extract patterns. Other steps in the knowledge discovery process include pre-mining tasks such as *data cleaning* (removing noise and inconsistent data) and *data integration* (bringing data from multiple sources to a single location and into a common format), as well as post-mining tasks such as *pattern evaluation* (identifying the truly interesting patterns representing knowledge) and *knowledge presentation* (presenting the discovered rules using visualization and knowledge representation techniques).

Many types of "interesting patterns" have been identified in the research literature and association rules constitute one such type. Data mining tasks to find these various patterns include:

1. Characterization: Data characterization is a summarization of the general characteristics or features of a user-specified target class of data. For example, the user may like to characterize software products whose sales increased by 10% in the last year. The output of data characterization can be presented in various forms such as pie charts, bar charts, multidimensional tables and data cubes.

2. Discrimination: Data discrimination is a comparison of the general features of a user-specified target class data objects with the general features of objects from one or a set of (user-specified) contrasting classes. For example, the user may like to compare the general features of software products whose sales increased by 10% in the last year with those whose sales decreased by at least 30% during the same period.

3. Association Analysis: Association analysis is the discovery of association rules showing attribute-value conditions that occur frequently together in a given set of data. Association analysis is widely used for *market basket* or transaction data analysis and forms the subject matter of this thesis.

4. Classification and Regression: Classification is the process of finding a set of models that describe and distinguish data classes or concepts, for the purpose of being able

to use the model to predict the class of objects whose class label is unknown. The derived model is based on the analysis of a set of *training data*. While classification predicts a categorical value, regression is applied if the field being predicted comes from a real-valued domain. Common applications of classification include credit card fraud detection, insurance risk analysis, bank loan approval, etc.

5. Cluster Analysis: Objects in a database are clustered or grouped based on the principle of *maximizing intraclass similarity and minimizing interclass similarity*. Unlike classification which has predefined labels, clustering must in essence automatically come up with the labels. Applications of clustering include demographic or market segmentation for identifying common traits of groups of people, discovering new types of stars in datasets of stellar objects, and so on.

6. Outlier Analysis: Outliers are data objects that do not comply with the general behaviour or model of the data. Most data mining methods discard outliers as noise or exceptions. However, in some applications such as fraud detection, the analysis and mining of outliers is crucial.

7. Evolution Analysis: Data evolution analysis describes and models regularities or trends for objects whose behaviour changes over time. Although this analysis may include any of the above functionalities on *time-related* data, distinct features of such an analysis include time-series data analysis, sequence or periodicity pattern matching, and similarity-based data analysis.

In general, data mining tasks can be classified into two categories: *descriptive* and *predictive*. Descriptive mining tasks characterize general properties of the data in the database. Examples include association rule discovery and clustering. On the other hand, predictive mining tasks perform inference on the current data in order to make predictions. Examples of predictive mining tasks include classification and regression.

We add a remark here that apart from the applications mentioned in the previous section, association rules have also been shown to be useful for classification [LHM98] and clustering [HKKM97] tasks. In [LHM98], association rules for each class in a classification

model are mined separately and then used to predict the class of objects whose class label is unknown. In [HKKM97], association rules are used to construct a hypergraph, and a hypergraph partitioning algorithm is used to find clusters of related items. This knowledge is then used to cluster the actual transactions in the database. These studies have shown that association rule mining enables efficient classification and clustering especially for databases that are large (in terms of either the number of transactions or the number of items).

1.3 Association Rule Mining

As has been explained earlier, association rule mining searches for interesting correlations among items in a given data set. It was originally proposed almost a decade ago, in [AIS93], and has since then attracted enormous attention in both academia and industry. In this section, we provide a formal description of the association rule mining problem along with an outline of the solution strategy and explain why the problem is technically challenging. We also briefly describe various extensions to the basic model that have been proposed in the research literature.

1.3.1 Problem Description

The inputs to this model are $\mathcal{I}$, a set of items sold by the store, and $\mathcal{D}$, a database of customer purchase transactions. In this context, an *association rule* is a (statistical) implication of the form $X \longrightarrow Y$, where $X, Y \subseteq \mathcal{I}$ and $X \cap Y = \phi$. Given an *itemset* X (i.e. a set of items), $X \subseteq \mathcal{I}$, its tidset is defined as $t(X) =$ set of tids of transactions that contain X. The support of X is defined as $support(X) = |t(X)|/|\mathcal{D}|$. The confidence of the rule $X \longrightarrow Y$ is given by $|t(X \cup Y)|/|t(X)|$, while its support is equal to $support(X \cup Y)$. The problem then, is to find all association rules whose confidence is not less than $minconf$ and whose support is not less than $minsup$ where $minconf$ and $minsup$ are user-specified parameters. Such rules are expected to be "interesting". Alternative measures of interestingness of rules are discussed in Chapter 3.

Solution Strategy: It has been observed in [AIS93] that association rule mining can be decomposed into two sub-tasks: (1) Find all *frequent* itemsets (i.e. itemsets whose support is not less than *minsup*). Algorithms that discover frequent itemsets usually follow the strategy of isolating itemsets that are potentially frequent (called *candidate* itemsets) and then compute the number of their occurrences (also called *counts*) over the database. The process of determining the counts of these itemsets is often referred to as *counting*. (2) For each discovered frequent itemset Z, generate rules of the form $X \longrightarrow (Z - X)$, $\forall X \subset Z$ and output those whose confidence is not less than $minconf$. The confidence of any of these rules can be calculated as $support(Z)/support(X)$. The support of Z would be available from step 1 of the solution strategy since it is frequent. Note that X, being a subset of Z, would also be frequent since it must be present in all transactions that contain Z. Therefore, its support would also be available from step 1 of the solution strategy. This result, originally from [AIS93], is highlighted in the lemma below.

Lemma 1 *All subsets of a frequent itemset are also frequent.*

Technical Challenges: In spite of the simplicity and elegance of the problem statement of association rule mining, it is difficult to solve satisfactorily. The first sub-task described above, which is to determine the frequent itemsets, is *extremely computationally intensive* and has been the focus of most of the research efforts on association rule mining. While the computational complexity of the second sub-task in terms of response-time performance is almost negligible in comparison, it has two major problems: (1) **Rule quantity:** too many rules are usually generated, and (2) **Rule quality:** not all of the rules are *interesting*. Both these problems are strongly related because approaches to identify interesting rules would automatically mitigate the rule quantity problem since only the interesting rules would be output.

1.3.2 Extensions

Rules generated by the basic association rule model discussed above are referred to as *boolean* association rules in the mining literature since the only relevant information in each database transaction is the presence or absence of an item. For brevity, we denote boolean association rule mining as *BAR-mining*. Many kinds of rules have been proposed in the research literature as extensions to BAR-mining. These include hierarchical, quantitative, categorical, cyclic, constrained and sequential rules. We briefly describe each of these extensions below:

1. Hierarchical Rules: It is possible to extract a semantically richer set of rules, called hierarchical rules [SA95], from a transaction database if an *is-a hierarchy* over the set of items in the database is provided. For example, given that sweaters and ski jackets are both instances of winter wear, the rules output could contain a "pseudo-item" called *winter wear* to denote "either sweater or ski jacket or both". An example of a hierarchical rule would be: *winter wear* $\longrightarrow$ *hiking boots*.

2. Quantitative and Categorical Rules: Relational tables in most business and scientific domains have richer attribute types than the boolean attributes considered in the basic problem for transactional databases. Attributes can be quantitative (e.g. age, income) or categorical (e.g. zip code, make of car). The problem of mining association rules over such attributes in relational databases has been addressed in [SA96]. An example of such a rule would be: (*Age*: 30...39) and (*Married*: Yes) $\longrightarrow$ (*NumCars*: 2).

3. Cyclic Rules: These rules, proposed in [ORS98], are association rules that display regular cyclic variation over time. For example, if we compute association rules over monthly sales data, we may observe seasonal variation where certain rules are true at approximately the same month each year. Discovering such rules and their periodicities may reveal interesting information that can be used for prediction and decision making.

4. Constrained Rules: In [NLHP98], the authors propose constrained rules as a means of specifying constraints (including domain, class and SQL-style aggregate constraints) to be satisfied by the antecedent and consequent of a mined association rule. For example, the user may want to find associations between itemsets whose types do not overlap, or associations from itemsets whose total price is under Rs.1,000 to itemsets whose average price is at least Rs.10,000.

5. Sequential Rules: While standard boolean association rules find associations between items within a single transaction, sequential rules, proposed in [AS95], discover associations between items purchased at different times. An example of such a rule is: customers typically rent "Star Wars", then "Empire Strikes Back" and then "Return of the Jedi".

BAR-mining is an important component in mining all of the above types of patterns. Previous works on generating hierarchical, quantitative and categorical rules (e.g. [SA95, SA96]) have shown that albeit requiring some preprocessing, these problems are finally reducible to BAR-mining. For cyclic and constrained rules, the authors in [ORS98, NLHP98], have integrated their techniques with existing BAR-mining algorithms. In [AS95], the strategy recommended for mining sequential rules includes a preprocessing stage that consists of standard BAR-mining. These examples, combined with the fact that BAR-mining can be successfully applied for classification and clustering tasks (refer Section 1.1) indicate that BAR-mining is an important "high-impact" problem. Therefore, in this thesis we mainly focus on BAR-mining.

1.4 Thesis Contributions

Various issues arise in BAR-mining including the efficiency of algorithms for the task, the conciseness of results that are output by these algorithms and the re-mining of a database after it has been updated with fresh data. Each of these issues are introduced below and are addressed in this thesis.

1.4.1 Issue 1: Efficiency of Algorithms

Market basket databases are typically very large and BAR-mining is computationally intensive due to the requirement of having to discover frequent itemsets in the data. It is therefore imperative to have fast scalable algorithms for this task. After the initial algorithms proposed in [AIS93, AS94], there have been a whole host of algorithms for addressing this problem (see Chapter 3). These algorithms have concentrated on improving both I/O costs by reducing the number of passes over the transaction database, and CPU costs by improving the efficiency of itemset counting techniques.

While the above efforts have certainly resulted in a variety of novel algorithms, each in turn claiming to outperform its predecessors on a representative set of databases, no logical end appears to be in sight. Therefore, in this thesis, we focus our attention on the question of how much space remains for performance improvement over current BAR-mining algorithms. The environment we consider, similar to the majority of the prior art in the field, is one where the data mining system has a single processor and the pattern lengths in the database are small enough that the frequent itemsets along with intermediate results produced by mining algorithms can fit in main memory. The case of longer patterns is discussed in Section 1.4.2.

Within the above framework, we make the following contributions (published in [PH02a, PH02b]):

First, we introduce the notion of an **"Oracle algorithm"** that knows *in advance* the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets to complete the mining process. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. Thus, this "Oracle approach" permits us to clearly demarcate the maximal space available for performance improvement over the currently available algorithms by comparing their performance against that of the Oracle. Further, it enables us to construct new mining algorithms from a completely different perspective, namely, as *minimally-altered derivatives* of the Oracle.

Second, we present a carefully engineered implementation of Oracle that makes the

best choices of data structures and database organizations (w.r.t. the enumeration of itemsets being counted). Our experimental results show that there is a considerable gap in the performance between the Oracle and existing mining algorithms.

Finally, we present a new mining algorithm, called **ARMOR** (Association Rule Mining based on ORacle), whose structure is derived by making minimal changes to the Oracle, and is guaranteed to complete in two passes over the database. Although ARMOR is derived from the Oracle, it may be seen to share the positive features of a variety of previous algorithms such as PARTITION [SON95], CARMA [Hid99], AS-CPA [LD98] and VIPER [SHS+00]. Our empirical study shows that ARMOR performs within a factor of two of the Oracle, over both real and synthetic databases for practical ranges of support specifications.

1.4.2 Issue 2: Conciseness of Results

The number of association rules generated in typical mining operations could run into the thousands, tens of thousands or even more. This makes it impractical for manual examination of the mining output [LHM99]. While this is true for sparse datasets where frequent itemsets are "short", it is often impractical to even generate all frequent itemsets and their associated supports for dense datasets. For instance, if the length of frequent itemsets grow beyond a mere thirty, the total number of frequent itemsets exceeds one billion! This result is due to the fact that all subsets of a frequent itemset must also be frequent (see Lemma 1).

In this thesis, we present techniques to reduce the output size of BAR-mining algorithms by identifying and pruning "redundant" rules. For this, we propose the *generalized closed itemset framework* (also referred to as g-closed itemset framework), published in [PH03a, PH03b]. In our scheme, we do not output *exact* supports of frequent itemsets – however, the supports of frequent itemsets can be estimated within a *deterministic*, user-specified "tolerance" factor. We empirically show that after removing redundant rules, our scheme results in exponentially fewer rules for most datasets and support specifications than the total number of frequent itemsets, even by allowing for a very small

tolerance. Our experiments were run on a variety of databases, both real and synthetic as well as sparse and dense, to confirm that the scheme works across a broad spectrum of database schemas and contents.

A side-effect of allowing for a tolerance in itemset supports is that the supports of some "borderline" infrequent itemsets may be over-estimated causing them to be incorrectly identified as frequent. We feel that this is acceptable in most mining scenarios for tolerance factors that are much less than the minimum support threshold. As such, by allowing for the tolerance factor, the user has authorized the supports of these borderline itemsets to be estimated above the minimum support. Finally, we ensure that *no false negatives* are ever produced – all frequent itemsets are correctly identified as frequent.

Our scheme can be used in one of two ways: (1) as a post-processing step of the mining process, or (2) as an integrated solution. We show that our scheme can be integrated into both levelwise algorithms as well as the more recent two-pass mining algorithms. We chose the classical Apriori algorithm [AS94] as a representative of the levelwise algorithms and the ARMOR algorithm [PH02b] (proposed in this thesis), as a representative of the class of two-pass mining algorithms. Integration into Apriori yields a new algorithm, *g*-**Apriori** and into ARMOR, yields *g*-**ARMOR**. Our experimental results show that these integrations often result in a significant reduction in response-time, especially for dense datasets.

We note that integration of our scheme into two-pass mining algorithms is a novel and important contribution because two-pass algorithms have several advantages over Apriori-like levelwise algorithms. These include: (1) significantly less I/O cost, (2) significantly better overall performance as shown in [P+01, PH02b], and (3) the ability to provide approximate supports of frequent itemsets at the end of the first pass itself, as in [Hid99, PH02b]. This ability is an essential requirement for mining *data streams* [MM02] as it is infeasible to perform more than one pass over the complete stream.

1.4.3 Issue 3: Re-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 and $DB \cup db$. Mining db is necessary to evaluate the effects of business strategies; whereas mining $DB \cup db$ is necessary to maintain the updated set of mining rules. This issue of "incremental" mining is also addressed in this thesis. Practical applications where incremental mining techniques are especially useful include data warehouses and web mining since these systems are constantly updated with fresh data – on the web, for instance, about one million pages are added daily [GRRS99].

In this thesis, we present and evaluate an incremental mining algorithm called **DELTA** (Differential Evaluation of Large iTemset Algorithm), published in [PH00]. DELTA represents a practical algorithm that can be effectively utilized for real-world databases. DELTA mines the frequent itemsets in both db as well as in $DB \cup db$ and guarantees that the entire mining process is completed in at most *three passes* over the increment and *one pass* over the previous database. We expect that such bounds will be useful to businesses for the proper scheduling of their mining operations.

DELTA can handle *multi-support* environments, where the minimum support specified by the user for the current database is *not* the same as for the previous database. It requires only *one* additional pass over the current database to achieve this functionality.

By integrating optimizations previously proposed for first-time hierarchical mining

algorithms, the DELTA design has been extended to efficiently handle incremental mining of *hierarchical* association rules. This illustrates the point noted in Section 1.3.2 that extensions to the basic association rule model including hierarchical, categorical and quantitative rules are finally reducible to BAR-mining.

The performance of DELTA is evaluated on a variety of dynamic databases and compared with that of Apriori and the previously proposed incremental mining algorithms for boolean association rules. For hierarchical association rules, we compare DELTA against the Cumulate first-time mining algorithm presented in [SA95]. All experiments are made on databases that are significantly larger than the entire main memory of the machine on which the experiments were conducted. The effects of database skew are also modeled. The results of our experiments show that DELTA can provide significant improvements in execution times over the previous algorithms in all these environments. Further, DELTA's performance is comparatively robust with respect to database skew.

We also include in our evaluation suite the performance of an *an Oracle* that has complete apriori knowledge of the identities of all the frequent itemsets both in the current database as well as in the increment and only requires to find their respective counts. Our experiments show that DELTA's efficiency is *close to that obtained by the oracle* for many of the workloads 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.

A final remark: Our work on incremental mining presented in this thesis was actually done prior to our work on the other two issues discussed above. However, for pedagogical reasons, we present it in the end.

1.4.4 Overall Architecture

In summary, the overall architecture for BAR-mining that we advocate in this thesis is shown in Figure 1.1. The user inputs the database and the following mining parameters – minimum support, minimum confidence and the tolerance factor for support approximation. The BAR-mining system performs the required processing by accessing the database and first produces concise frequent itemsets and other intermediate results. These results

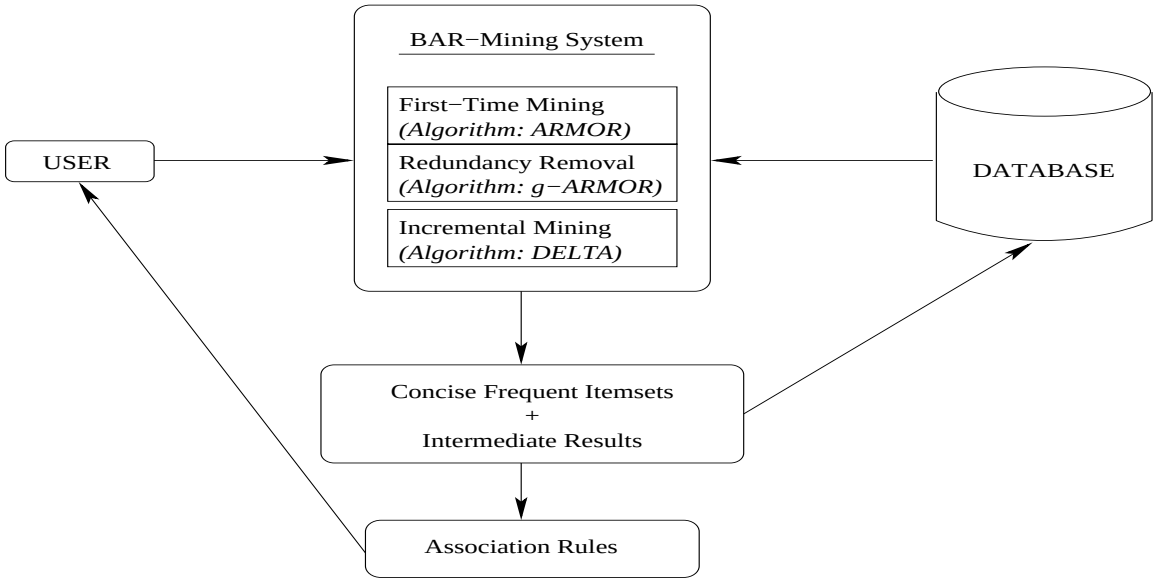

Figure 1.1: Architecture for BAR-mining

are then used to form the association rules that are presented to the user. In this thesis, we present the ARMOR algorithm for first-time mining, followed by the g-ARMOR algorithm which is an enhancement of ARMOR to remove redundancy from the output. Finally, we present the DELTA algorithm to handle re-mining in an incremental fashion.

1.5 Organization

The remainder of this dissertation is organized in the following fashion: In Chapter 2, we describe the overall methodology and scope of our work. Next, in Chapter 3, we review the published research related to our work. In Chapter 4, we present the Oracle approach using which we evaluate the performance of current BAR-mining algorithms. In this chapter we also present and evaluate the ARMOR algorithm for BAR-mining. Next, in Chapter 5 we present the g-closed itemset framework along with the g-Apriori and g-ARMOR algorithms for mining frequent g-closed itemsets. The DELTA algorithm for incremental mining is presented and evaluated in Chapter 6. Finally, Chapter 7 summarizes the main contributions of our study and outlines future avenues to explore.

Chapter 2

Methodology and Scope

The problem of BAR-mining has been described in the previous chapter. In this chapter, we describe the overall methodology and scope of this thesis in terms of the database, system and pattern characteristics considered in our study. Our choices are such that they match those selected in the majority of the previous studies. For ease of reference, we also describe at the end of this chapter, the notation used throughout this thesis.

2.1 Database and System Characteristics

Conceptually, a market-basket database is a two-dimensional matrix where the rows represent individual customer purchase transactions and the columns represent the items on sale. This matrix can be implemented in the following four different ways [SHS+00], which are pictorially shown in Figure 2.1:

Item-vector (IV): The database is organized as a set of rows with each row storing a transaction identifier (TID) and a bit-vector of 1's and 0's to represent for each of the items on sale, its presence or absence, respectively, in the transaction.

Item-list (IL): This is similar to IV, except that each row stores an ordered list of item-identifiers (IID), representing only the items *actually* purchased in the transaction.

Tid-vector (TV): The database is organized as a set of columns with each column

storing an IID and a bit-vector of 1's and 0's to represent the presence or absence, respectively, of the item in the set of customer transactions.

Tid-list (TL): This is similar to TV, except that each column stores an ordered list of only the TIDs of the transactions in which the item was purchased.

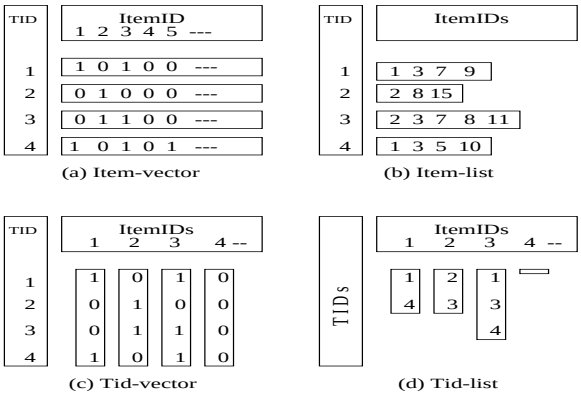

Figure 2.1: Comparison of Data Layouts

While a mining algorithm is free to dynamically change the database layout during the mining process, we assume that the *initial* database is always provided in the horizontal item-list (IL) format.

System Characteristics While there has been significant work in designing algorithms for the *parallel mining* of association rules [AS96, HKK97, ZPOL97b, PZOL01], in this study we focus on single processor environments. We also assume that the available main memory in the system is typically much smaller than the database size.

2.2 Pattern Characteristics

In this section, we describe the patterns that are output by the mining algorithms developed in this thesis.

2.2.1 Boolean Association Rules

In most of this thesis, we restrict our attention to the problem of generating *boolean* association rules where the only relevant information in each database transaction is the presence or absence of an item. As mentioned in Chapter 1, BAR-mining is an important component in mining other patterns such as hierarchical rules, quantitative rules, etc. In order to illustrate this point, we include the incremental mining of hierarchical rules in Chapter 6.

2.2.2 Negative Border

In designing algorithms in this thesis, we often utilize the concept of the *negative border* [Toi96] of a set of itemsets. Intuitively, the negative border consists of *minimal* infrequent itemsets. More formally, the negative border N of a set of itemsets F is defined as follows: An itemset X belongs to N iff $X \notin F$ but all subsets of X are in F. Algorithms that mine the collection of frequent itemsets also typically generate the itemsets in its negative border and their associated supports. The negative border information is important in BAR-mining due to the following reasons:

- It has been shown in [MGKS97, MT97] that in certain restricted models of computation all the itemsets in the negative border *have* to be examined. In particular, it was shown that:

  Theorem 1 *Any algorithm that computes the set of frequent itemsets and accesses the data using only queries of the following form: "Is itemset X frequent?" must use at least $|N|$ such queries.*

- The negative border information has been found to be especially useful in the design of *incremental* mining algorithms [PH00, T$^+$97, F$^+$97].

Due to these reasons, we include the negative border as required output in all the algorithms that we design in this thesis.

2.2.3 g-Closed Itemsets

In Chapter 5, we introduce the concept of g-closed itemsets for removing redundancy from mining results. The set of frequent g-closed itemsets is such that it is typically much smaller than the set of all frequent itemsets. However, the identities and supports of all frequent itemsets can be estimated from those of the frequent g-closed itemsets. The discrepancy in estimation of supports is guaranteed to be within a user-specified *tolerance* factor ϵ.

2.2.4 Pattern Length

In this thesis, while designing algorithms to discover *all* frequent itemsets, the environment we consider is of *sparse* databases where the pattern lengths in the database are small enough that the frequent itemsets along with intermediate results can fit in main memory. It is infeasible to consider longer patterns when mining all frequent itemsets because the number of frequent itemsets grows exponentially with increasing pattern lengths.

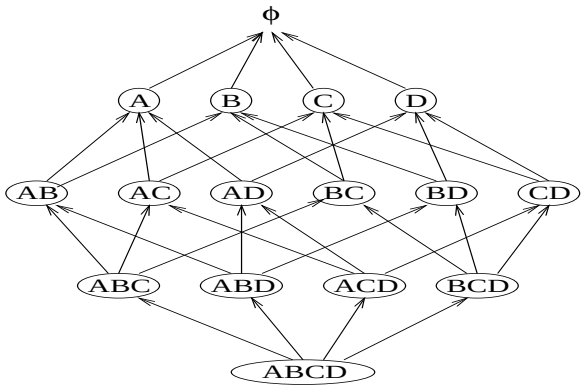

Figure 2.2: Complete Itemset Lattice for Items {A,B,C,D}

Further, we consider only *bottom-up* approaches to enumerate the solution space consisting of the lattice of all possible itemsets (see Figure 2.2 for an example of such a lattice). When mining *all* frequent itemsets, there would be no particular advantage in counting the support of an itemset X before counting the supports of its subsets. This is because even if X is frequent, its subsets have to be counted anyway. An exception to this rule would occur when there are itemsets that have supersets with *exactly* equal supports. The

number of such itemsets is likely to be small in sparse databases. However, we address this issue in Chapter 5, where we introduce the g-closed itemset framework that is designed to handle both sparse and dense databases.

2.3 Mining Algorithms Input/Output

In this section, we define the input and output of the algorithms that we develop in this thesis. However, we do not impose the restrictions implied in these definitions to algorithms that have been developed elsewhere. For example, although we require standard first-time mining algorithms developed in this thesis to include the negative border of frequent itemsets as part of the output, we recognize algorithms that have been developed elsewhere that do not meet this requirement.

2.3.1 First-Time Mining

All standard first-time *online* mining algorithms in our study take as input the database $\mathcal{D}$ in item-list (IL) format and the minimum support threshold *minsup* and produce as output the set of frequent itemsets F and its negative border N along with their corresponding supports.

The Oracle algorithm for first-time mining, on the other hand, takes as input the database $\mathcal{D}$ in item-list (IL) format, the set of frequent itemsets F and its negative border N, and produces as output the supports of itemsets in $F \cup N$.

2.3.2 Redundancy Removal

The algorithms that we propose in this thesis to remove redundancy from the mining results take as input the database $\mathcal{D}$ in item-list (IL) format, the minimum support threshold *minsup* and the tolerance factor ϵ, and produce as output the set of *frequent g-closed* itemsets and its negative border along with their corresponding supports.

2.3.3 Incremental Mining

Incremental mining algorithms take as input the original database DB, the increment db (may consist of both insertions and deletions to DB), the original minimum support threshold $minsup_{DB}$, the new minimum support threshold $minsup_{DB \cup db}$, the set of previous frequent itemsets F_{DB}, its negative border N_{DB}, and their associated supports. The output is the updated versions of the frequent itemsets and their negative border, namely, $F_{DB \cup db}$ and $N_{DB \cup db}$ along with their supports. In addition, the mining results for solely the increment, namely, $F_{db} \cup N_{db}$, are also output.

2.4 Implementation Complexity and Platforms

All mining algorithms that have been designed/evaluated in our work are implemented in standard C++. The code is highly portable and currently supports Linux, Solaris and Irix. The entire source code written in our implementation spans 129 files and includes 42,048 lines of code. In addition to this, several programs were written using the Bash and Perl scripting languages to automate many of the tasks in evaluating the mining algorithms. These tasks include formatting of the input to the implemented algorithms, timing their response times and using their output to generate graphs to compare their performance.

2.5 Notation

For ease of exposition and reference, we will use the notation shown in Table 2.1 in the remainder of this thesis.

Frequent Itemset Mining Algorithms Input/Output	
$\mathcal{I}$	Set of items in the database
$\mathcal{D}$	Database of customer purchase transactions
$minsup$	User-specified minimum rule support
$minconf$	User-specified minimum rule support
F	Set of frequent itemsets in $\mathcal{D}$
N	Negative border of F
$support(X)$	Support of itemset X
$t(X)$	Tidset of itemset X
$i(T)$	Set of items that are common to transactions in T
For Oracle, ARMOR and g-ARMOR Algorithms	
$P_1, P_2, ..., P_n$	Set of n disjoint partitions of $\mathcal{D}$
d	No of transactions in partitions scanned so far during algorithm execution *excluding* the current partition
d^+	No of transactions in partitions scanned so far during algorithm execution *including* the current partition
$\mathcal{G}$	DAG structure to store candidates during algorithm execution
For g-Apriori and g-ARMOR Algorithms	
$c(X)$	Closed itemset corresponding to itemset X
$g(X)$	g-Closed itemset corresponding to itemset X
ϵ	Tolerance factor
C_k	Set of candidate k-itemsets
G_k	Set of frequent k-generators
G	Set of all frequent generators produced so far
For the DELTA Algorithm	
DB, db, $DB \cup db$	Previous, increment, and current database
$minsup_{DB}$	Previous Minimum Support Threshold
$minsup_{DB \cup db}$	New Minimum Support Threshold
$minsup$	Minimum Support Threshold when $minsup_{DB} = minsup_{DB \cup db}$
F_{DB}, F_{db}, $F_{DB \cup db}$	Set of frequent itemsets in DB, db and $DB \cup db$
N_{DB}, N_{db}, $N_{DB \cup db}$	Negative borders of F_{DB}, F_{db} and $F_{DB \cup db}$
F_{known}	Set of known-frequent itemsets during algorithm execution: $F_{DB \cup db} \cap (F_{DB} \cup N_{DB})$
N_{known}	Negative border of F_{known}
$Infrequent$	Set of known-infrequent itemsets during algorithm execution
$Infrequent_{db}$	Set of known-infrequent (within db) itemsets during algorithm execution

Table 2.1: Notation

Chapter 3

Related Work

In this chapter, we review the published research related to our work in each of the three issues described in the Introduction – namely, the efficiency of BAR-mining algorithms, the conciseness of mining results and incremental mining.

3.1 Efficiency of Algorithms

There have been over thirty BAR-mining algorithms in the research literature. In this section, we briefly review a representative set of the major algorithms proposed. As mentioned in Chapter 2, we consider only *bottom-up* algorithms that were designed to mine *sparse* databases.

1. **AIS**: The very first algorithm was AIS [AIS93]. It was proposed in [AIS93] in which the problem of BAR-mining was introduced. This is a "multi-pass" algorithm in which candidate itemsets are generated while scanning the database by extending known-frequent itemsets with items from each transaction. An estimate of the supports of these candidates is used to guide whether these candidates need to be extended further to produce more candidates. It was later discovered in [AS94] that AIS generates too many candidates and is thereby inefficient.

2. **Apriori**: The AIS algorithm was followed by the Apriori algorithm [AS94] that was shown to perform better than AIS by an order of magnitude. The most important

aspect of Apriori is to completely incorporate the *subset frequency based pruning* optimization – that is, it does not process any itemset whose subset is known to be infrequent. It utilizes a data structure called *hashtree* to store the counters of candidate itemsets. The main drawback in this algorithm is that it performs n passes over the database, where n is the length of the longest frequent itemset. In the k^{th} pass, the counts of candidate itemsets of length k (called *k-itemsets*) are obtained. An other drawback is that Apriori follows a tuple-by-tuple approach – that is, it updates counters of candidate itemsets after reading in *each* transaction from the database. It hence suffers from the drawback that much redundant work (traversal of the data structure holding the counters of itemsets) is performed after each and every transaction.

3. **Partition**: The *partitioning strategy* was introduced in [SON95], wherein the database is logically divided into a number of disjoint partitions. The Partition algorithm requires at most two passes and is based on the observation that an itemset can be globally frequent over the entire database iff it is locally frequent in at least one partition. The counting strategy in this algorithm computes for each candidate itemset, a list of tids of transactions that contain the itemset. These lists (also referred to as *tid-lists*) are computed separately for each partition and are used for efficient counting.

4. **Sampling**: This algorithm, proposed in [Toi96] first mines a random sample of the database to obtain itemsets that are frequent within the sample. These itemsets could be considered as a representative of the actual frequent itemsets in applications where approximate mining results are sufficient. In order to obtain accurate mining results, this algorithm requires one or two scans over the entire database. The Sampling algorithm too follows a tuple-by-tuple approach and hence, like Apriori, suffers from the above mentioned drawback.

5. **AS-CPA**: This is a variation of Partition proposed in [LD98] that makes use of the cumulative count of each candidate to achieve an illusion of a "large partition". At

any instant, it stores only the candidates that are frequent over their respective large partitions. However, there are no details of data-structures or of tid-list computation in [LD98].

6. **DIC**: In DIC [BMUT97], candidates are generated and removed after every M transactions where M is a parameter to the algorithm. Although it is a multi-pass algorithm, it was shown to complete within two passes typically. It however, suffers from the drawbacks of tuple-by-tuple approaches.

7. **CARMA**: This is a 2-pass algorithm proposed in [Hid99] that has the feature of dynamically generating and removing candidates after each tuple of the database is processed. Though a novel approach, the CARMA algorithm suffers from the drawbacks of tuple-by-tuple approaches. It was shown in [Hid99] that while CARMA did not perform consistently better than Apriori, its memory utilization was less by an order of magnitude.

8. **FP-growth**: After a preprocessing scan over the database, this algorithm proposed in [HPY00] constructs a condensed representation of the database called an FP-tree and then performs mining over the FP-tree.

9. **MaxClique**: While the above algorithms were primarily horizontal (tuple) based approaches, the MaxClique [ZPOL97a] algorithm is designed to efficiently mine databases that are available in a vertical layout.

10. **VIPER**: Unlike earlier vertical mining algorithm which were subject to various restrictions on the underlying database size, shape, contents or the mining process, the VIPER [SHS+00] algorithm does not have any such restrictions. It includes many optimizations to enable efficient processing and was shown to outperform earlier vertical mining algorithms. It also scales well with the database size.

All the above-mentioned studies (except VIPER, as discussed below) have focussed on evaluating the performance of mining algorithms with respect to their predecessors. In particular, most of them compare against the classical Apriori online mining algorithm.

With regard to evaluating the performance of mining algorithms with respect to idealized, offline algorithms, a preliminary step was taken in our work on incremental mining. As mentioned in Chapter 1, this work was done prior to our work on the other two issues addressed in this thesis, but is presented in the end due to pedagogical reasons. In this work, we compared the DELTA algorithm against the oracle version of *Apriori* suitably modified for incremental mining. We refer to this algorithm as Apriori-Oracle. It differs very much from the Oracle algorithm used in this thesis to evaluate the performance of *first-time* mining algorithms in the following significant aspects: (1) The Apriori-Oracle primarily used the hashtree data structure [AS94] whereas Oracle primarily uses the DAG structure (as defined in Chapter 4). (2) The Apriori-Oracle does counting with a tuple-by-tuple approach, while Oracle follows a partitioning approach. (3) Finally, no proofs of optimality are associated with the Apriori-Oracle. Another version of the Apriori-Oracle was later used in [SHS+00] for comparison with VIPER.

3.2 Conciseness of Results

The algorithms discussed in the previous section were designed to address the first sub-task of BAR-mining, which is to generate frequent itemsets. In this section, we review various approaches to solving the problems associated with the second sub-task, which is to generate rules from the discovered frequent itemsets. As discussed earlier, these problems are: (1) **Rule quantity:** too many rules are usually generated, and (2) **Rule quality:** not all of the rules are *interesting*. Both these problems are strongly related because approaches to identify interesting rules would automatically mitigate the rule quantity problem since only the interesting rules would be output. Here, we discuss related work that primarily addresses the problem of rule quantity. Techniques that primarily address the problem of rule quality are discussed in Section 3.4.1.

3.2.1 Post-Mining Rule Pruning Schemes

A number of techniques to discover "redundancy" in association rules have been proposed that are *post-mining rule analysis schemes*. That is, they are to be applied *after* frequent itemsets have been mined using a standard BAR-mining algorithm (like those discussed in the previous section).

The concept of *association rule covers* was proposed in [TKR+95]. In this context, a cover is a subset of the original set of rules such that for each tuple in the database there is an applicable rule in the cover. Rules that are not in the cover are considered redundant and are pruned. In this framework, a rule $X \longrightarrow Y$ "covers" all rules that contain a superset of X in the antecedent. These latter rules are therefore pruned. A drawback of this approach is that if these pruned rules have significantly different confidence compared to the rule that covers them, then clearly, information is lost.

In [AY98], a rule is considered redundant w.r.t another rule, if it is possible to derive *just the identity* of the redundant rule from the latter. Note that we do not need to be able to derive the support and confidence of the redundant rule. Clearly, information can be lost in this approach because the support and confidence of the pruned redundant rule could be significantly different from what is expected by analyzing the non-redundant rules. Another rule pruning technique was presented in [BAG99] using the concept of *improvement*, which is the difference between the confidence of a rule and the confidence of any proper sub-rule[1] with the same consequent. Those rules that do not meet a user-specified minimum improvement threshold are pruned.

The work in [DL98] introduces the notion of the *neighbourhood* of a rule. It then defines the interestingness of a rule based on certain parameters of its neighbourhood such as the average confidence in the neighbourhood, the density of rules, etc. In [LHM99], the authors used the standard χ^2 test to prune insignificant rules. A general pruning technique was presented in [SLR99] consisting of several *pruning rules* to identify and remove redundant itemsets. These rules are applicable for many different types of patterns such as associations and implications and for various statistical measures such as confidence,

[1] A rule Q is a sub-rule of another rule P iff P contains all the items present in Q.

support, interest, etc.

A limitation of the above-mentioned studies is that their techniques to discover redundancy are to be applied *after* frequent itemsets have been mined using a standard BAR-mining algorithm. These approaches are therefore inefficient and sometimes even infeasible because the number of frequent itemsets could be very large, especially for dense databases.

3.2.2 Pruning During Mining

Techniques to prune rules during the frequent itemset discovery phase itself have been proposed and are discussed below.

In [KMR+94], the authors propose an approach to allow the user to specify what rules are required using *templates*. The system then retrieves those rules that match these templates. A related scheme in [SVA97] enables users to specify *constraints* to be satisfied by the mined rules. A more comprehensive scheme was proposed in [NLHP98] to enable users to specify a larger variety of constraints (including domain, class and SQL-style aggregate constraints) to be satisfied by the antecedent and consequent of a mined association rule.

Another approach that has been considered in the mining literature for reducing the size of mining output is to mine only the *maximal frequent itemsets* [Bay98, LK98, AAP98, GZ01]. A frequent itemset is called maximal if it is not a subset of any other frequent itemset. The motivation for this approach is that all subsets of a maximal frequent itemset are frequent and hence might be considered redundant. A drawback of these approaches is that maximal itemsets cannot be used directly for rule generation, since support of subsets is required for confidence computation. While an extra database scan could be made to gather these supports, we revert to the problem of many redundant rules.

Closed Itemset Based Techniques

Alternative techniques for pruning uninteresting rules based on the *closed itemset* framework [ZH02, PBTL99] have been previously presented in [Zak00, TPBL00, CS02, BBR00,

BB00]. These techniques have a tighter requirement for redundancy: A rule is redundant only if its identity and support can be *derived* from another "non-redundant" rule. Therefore, in this framework, no information is lost by pruning because both the identities and supports of all frequent itemsets can be regenerated *completely* from the frequent closed itemsets, which is a subset of the frequent itemsets. However, as we will show in this thesis, the usefulness of the basic closed itemset framework depends on the presence of frequent itemsets that have supersets with *exactly the same support*. This means that even minor changes in the database can result in a significant increase in the number of frequent closed itemsets.

In this thesis, we too follow the tighter approach for a rule to be considered redundant. However, as mentioned in Chapter 1, we relax the requirement of deriving exact supports – instead, it is sufficient if the supports can be estimated within a deterministic user-specified *tolerance* factor. This strategy of relaxing the requirement of deriving exact supports has also been considered in [BBR00, BB00]. In [BBR00], the authors develop the notion of *freesets* along with an algorithm called *MINEX* to mine them. The bound on approximation error in the freesets approach increases linearly with itemset length in contrast to the constant bound featured in our approach. In [BB00], the authors do not provide any bounds on approximation error. Further, the focus in [BBR00, BB00] is *only* on highly correlated, i.e. "dense" data sets, whereas we show that our techniques can be profitably applied even on sparse data sets. Another difference is that our technique to mine g-closed itemsets bypasses the additional processing that is required in the MINEX algorithm to test for "freeness". Finally, there was no attempt in [BBR00, BB00] to incorporate their scheme into two-pass mining algorithms, which as mentioned in Chapter 1, is essential for mining *data streams*.

One of the algorithms proposed in this thesis for mining frequent g-closed itemsets, g-Apriori, is based on the classical Apriori algorithm. We note that there was another algorithm called A-Close for mining frequent closed itemsets that was also based on Apriori. It first mines what are known as the "generators" of frequent closed itemsets and then makes an additional database scan to determine the closed itemsets from their respective

generators. Our algorithm significantly differs from A-Close (even for the zero tolerance case) in that it does not require the additional database scan to mine closed itemsets.

We note that the approaches based on the closed itemset concept, including the techniques in this thesis, are complementary to the other approaches for rule pruning and can be combined with them to perhaps achieve even better results.

3.3 Incremental Algorithms

In this section, we provide an overview of the algorithms that have been developed over the last few years for incremental BAR-mining.

3.3.1 The FUP Algorithm

The **FUP** (Fast UPdate) algorithm [CHNW96, CLK97, CVB96] 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. Further, it does not generate the mining results for solely the increment.

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 12 (Section 6.2). 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, 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 [F$^+$97] and [T$^+$97] with the goal of achieving more efficiency in the

incremental mining process. In this approach, itemsets that were originally in the negative border of the frequent itemsets and later become frequent after the database has been updated are referred to as *promoted borders*. Algorithms that follow the negative border approach typically compute what is known as the *negative border closure*. This consists of all possible extensions of the promoted borders except those that have subsets known to be infrequent. In the sequel, we will use Borders to refer to the algorithm in [F+97], and TBAR to refer to the algorithm in [T+97].

3.3.2 The Borders Algorithm

The original **Borders** algorithm computes the entire negative border closure at one shot and then makes a scan of the entire database to compute the counts of itemsets in the closure. This could potentially result in a "candidate explosion" problem that is later described in Section 6.2.3 of Chapter 6.

A new version of the Borders algorithm was proposed in [AFLM99]. This version goes to the other extreme of the closure computation, and makes one scan of the entire database for each "layer" of the negative border closure. As mentioned in Section 6.2.3 of Chapter 6, this strategy could result in a significant increase in the number of database passes, and may therefore be problematic for large databases.

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.

Finally, like FUP, Borders also does not generate the mining results for solely the increment.

3.3.3 The TBAR Algorithm

The **TBAR** algorithm initially *completely mines* the increment *db* by applying the Apriori algorithm. We expect this strategy to be inefficient for large increments since the previous mining results are not used at all in this mining process.

Next, it adopts an approach similar to Borders in that it computes the entire negative border closure at one shot. However, since the results of mining the increment are available at this time, this information could be used to prune more candidates from the closure – after computing each level of the closure, itemsets that are infrequent in the increment are excluded from further candidate generation. Therefore, unlike Borders, the candidate explosion problem is unlikely to occur. However, even with this pruning, there are likely to be too many unnecessary candidates in TBAR, especially for skewed increments since it relies solely on the increment for its pruning.

3.3.4 Other Algorithms

It was briefly mentioned in [Hid99] that CARMA, a first-time mining algorithm could be also applied for incremental mining. Although the algorithm is a novel and efficient approach for 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 $sup_{min} * |DB|$.

An incremental mining algorithm, called **MLUp**, for updating "multi-level" association rules over a taxonomy hierarchy was presented in [CVB96]. While MLUp's goal is superficially similar to the incremental hierarchical mining discussed in this thesis, it has the following major differences: Firstly, a *different* minimum support threshold is used for each level of the hierarchy. Secondly, MLUp restricts its attention to deriving *intra-level* rules, that is, rules within each level. In contrast, our focus in this thesis is on the formulation given in [SA95] where there is only one minimum support threshold and *inter-level* rules form part of the output.

3.4 Other Issues

In this section, we discuss those issues, which although not directly related to out work, are nevertheless relevant to BAR-mining.

3.4.1 Interestingness Measures

In the original formulation of the BAR-mining problem, confidence and support are two of the interestingness measures proposed. The confidence of a rule represents its likelihood of being true whereas the support of a rule represents its statistical significance. Support also measures the applicability of a rule since a rule with high support would be applicable in a large number of transactions. It was subsequently shown in [BMUT97] that the confidence measure is often misleading in practical situations. For example, if a rule states that "90% of researchers drink coffee" and $minconf = 85\%$, it might seem to imply a strong positive correlation between being a researcher and drinking coffee. However, if further analysis shows that 95% of *all* people drink coffee, it would indicate that there is actually a *negative* correlation between being a researcher and drinking coffee.

The above point motivated additional measures for identifying interesting rules including *conviction* [BMUT97] and *interest* [BMS97]. If $P(X)$ represents the probability of occurance of itemset X in the database, the conviction of a rule $X \longrightarrow Y$ is given by $P(X)P(\neg Y)/P(X, \neg Y)$ whereas its interest is given by $P(X,Y)/P(X)P(Y)$. Rules with high conviction are referred to as *implication rules*. Although these interestingness measures improved the quality of mining output, it was observed in [BMUT97] that the number of rules generated were still too many.

A novel scheme was proposed in [CSD98] that departs considerably from the BAR-mining norm in that it does not rely on the minimum support threshold. In this scheme, an itemset is considered uninteresting if the correlation between the items contained in it can be estimated given correlations of its subsets, and correlations at earlier points in *time*. For example, even a very frequent itemset would be considered uninteresting if its support does not vary appreciably over time. As might be expected, the algorithms to solve this problem are quite complex – quadratic time w.r.t the number of database

transactions. However, the authors in [CSD98] provide various heuristics to obtain an almost linear-time complexity. Further, they make the observation that techniques based on the minimum support threshold can be integrated with their scheme.

3.4.2 Backend

Our implementations of mining algorithms utilize a file system backend similar to most available research prototypes for BAR-mining. More specifically, our transaction data is stored in binary files that contain sequences of transactions. The format of each transaction is a quadruple: (1) An integer representing the transaction id (tid), (2) An integer representing the customer id, (3) the number of items purchased by the customer, and (4) integers representing the the actual items purchased.

While the above backend suffices for our purposes of algorithmic performance evaluation, there has been some work on integrating BAR-mining with *relational database backends* [STA98, RCIC99, NT99]. The most comprehensive of these works, [STA98], evaluates various alternative ways of integrating the classical Apriori algorithm with RDBMS backends. These alternatives include: loose-coupling through a SQL cursor interface; encapsulation of a mining algorithm in a stored procedure; caching the data to a file system on-the-fly and mining; tight-coupling using primarily user-defined functions and SQL implementations for processing in the DBMS. Their evaluation shows that the Cache-Mine option is superior to other alternatives from a response-time performance perspective.

A related issue of interest is the mining of associations *across* many databases. In real life, large collections of data may be organized in the form of a set of relations which is partitioned into several databases. These databases may hold interesting *inter-database* associations such as "89% of employees having a salary in the range (Rs. 15,000 – Rs. 25,000) own cars of type Maruti-800". Sophisticated algorithms for finding such inter-database associations are presented in [SAM99].

3.4.3 Privacy

The knowledge models produced through data mining techniques are only as good as the accuracy of their input data. One source of data inaccuracy is when users deliberately provide wrong information. This is especially common with regard to customers who are asked to provide personal information on Web forms to e-commerce service providers. The compulsion for doing so may be the (perhaps well-founded) worry that the requested information may be misused by the service provider to harass the customer. As a case in point, consider a pharmaceutical company that asks clients to disclose the diseases they have suffered from in order to investigate the correlations in their occurrences – for example, "Adult females with malarial infections are also prone to contract tuberculosis". While the company may be acquiring the data solely for genuine data mining purposes that would eventually reflect itself in better service to the client, at the same time the client might worry that if her medical records are either inadvertently or deliberately disclosed, it may adversely affect her employment opportunities.

Recently, there has been much interest in the data mining community on investigating whether customers can be encouraged to provide correct information by ensuring that the mining process cannot, with any reasonable degree of certainty, violate their privacy. At the same time, the mining process should be as accurate as possible in terms of its results. The difficulty lies in the fact that these two metrics: *privacy* and *accuracy*, are typically contradictory in nature, with the consequence that improving one usually incurs a cost in the other.

The first work on privacy-preserving mining appeared in [AS00], which investigated this issue in the context of *classification rule* mining. An approach of value distortion, wherein a random value is added to each original value, was taken in this work. They presented two algorithms, *ByClass* and *Local*, which knowing the distribution of the random values, attempt to reconstruct the original distribution within some acceptable error bound. The privacy attained is quantified by the "fuzziness" provided by the system, that is, for a given level of confidence, the size of the interval that is expected to hold the original true value. A followup of this work in [AA01] showed that the privacy estimates

of [AS00] were overstated since they did not account for the additional knowledge that the miner obtains from the reconstructed aggregate distribution. An alternative privacy formulation that takes such "side-information" into account was presented.

With regard to privacy in association rule mining, there have been a number of papers that have appeared over the last year [SVC01, ABE+99, DVEB01, SVE02, VC02, KC02, EGSA02]. The focus in [SVC01, ABE+99, DVEB01, SVE02] is to prevent sensitive rules from being inferred by the miner – they achieve this by either altering some of the entries in the true database or by replacing some of the entries with NULL values.

In [LP02, VC02, KC02], the problem considered is that of obtaining data mining results across a *distributed* set of sites with each site only willing to share data mining results, but not the source data. While [LP02] addresses this problem in the context of decision tree classisfiers, [VC02, KC02] address it in the context of association rules. [LP02] model it as a problem of secure multi-party computation and proposes a solution that demands very few rounds of communication and is efficient with regards to network bandwidth consumption. In [VC02], they consider data that is vertically partitioned, that is, different columns of the database reside on different sites, while [KC02], consider the complementary situation where the data is horizontally partitioned across the sites.

Chapter 4

Efficiency of Mining Algorithms

4.1 Introduction

The problem of efficiently mining frequent itemsets from large historical "market-basket" databases was introduced almost a decade ago, in [AIS93]. Since then, a whole host of algorithms for addressing this problem have been proposed [AIS93, AS94, SON95, PCY95b, HKK97, Hid99, HPY00, SHS+00, AAP01]. The latest include FP-growth [HPY00], which utilizes a prefix-tree structure for compactly representing and processing pattern information, and VIPER [SHS+00], which organizes and processes the database on a vertical (column) basis as opposed to the more traditional horizontal (row) basis.

While the above efforts have certainly resulted in a variety of novel algorithms, each in turn claiming to outperform its predecessors on a representative set of databases, no logical end appears to be in sight. Therefore, in this chapter, we focus our attention on the question of how much space remains for performance improvement over current frequent itemset mining algorithms. As discussed in Chapter 2, the environment we consider, similar to the majority of the prior art in the field, is one where the data mining system has a single processor and the pattern lengths in the database are small enough that the frequent itemsets along with intermediate results produced by mining algorithms can fit in main memory. That is, we restrict our attention to the class of *sequential bottom-up* mining algorithms to mine *sparse* databases.

Within the above framework, we make the following contributions:

First, we introduce the notion of an **"Oracle algorithm"** that knows *in advance* the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets to complete the mining process. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. Thus, this "Oracle approach" permits us to clearly demarcate the maximal space available for performance improvement over the currently available algorithms. Further, it enables us to construct new mining algorithms from a completely different perspective, namely, as *minimally-altered derivatives* of the Oracle.

Second, we present a carefully engineered implementation of Oracle that makes the best choices of data structures and database organizations (w.r.t. the enumeration of itemsets being counted). Our experimental results show that there is a considerable gap in the performance between the Oracle and existing mining algorithms.

Third, we present a new mining algorithm, called **ARMOR** (Association Rule Mining based on ORacle), whose structure is derived by making minimal changes to the Oracle, and is guaranteed to complete in two passes over the database. Although ARMOR is derived from the Oracle, it may be seen to share the positive features of a variety of previous algorithms such as PARTITION [SON95], CARMA [Hid99], AS-CPA [LD98] and VIPER [SHS+00]. Our empirical study shows that ARMOR performs within a factor of two of the Oracle, over a variety of databases and practical ranges of support specifications.

Finally, 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. However, previous performance studies have been largely conducted on databases that completely *fit in main memory*. For example, the standard experiment is one that has only 100K tuples with an average tuple width of 50 bytes – this fits easily in current memories that are typically in the hundreds of megabytes. Therefore, the ability of these algorithms to scale with database size, an important requirement for mining applications, has not been con-

clusively shown. In [SHS+00], the authors had demonstrated that this was an important issue and that algorithms that worked very well for memory-resident databases did not necessarily perform as well in disk-resident databases. Consistent with that observation, here too we conduct experiments with such large disk-resident databases.

A fallout of this approach is that we find algorithms such as FP-growth, despite having an attractive design, to perform rather poorly in practice.

For ease of exposition, we will use the notation shown in Table 2.1 of Chapter 2 in the remainder of this chapter. The relevant part of this table has been reproduced in Table 4.1 for convenience.

$\mathcal{D}$	Database of customer purchase transactions
$minsup$	User-specified minimum rule support
F	Set of frequent itemsets in $\mathcal{D}$
N	Set of itemsets in the negative border of F
$P_1, P_2, ..., P_n$	Set of n disjoint partitions of $\mathcal{D}$
d	No of transactions in partitions scanned so far during algorithm execution *excluding* the current partition
d^+	No of transactions in partitions scanned so far during algorithm execution *including* the current partition
$\mathcal{G}$	DAG structure to store candidates during algorithm execution

Table 4.1: Notation (from Table 2.1)

4.1.1 Organization

The remainder of this chapter is organized as follows: The design of the Oracle algorithm is described in Section 4.2 and is used to evaluate the performance of current algorithms in Section 4.3. Our new ARMOR algorithm is presented in Section 4.4. The details of candidate generation in ARMOR are discussed in Section 4.5, while its main memory requirements are discussed in Section 4.6. The performance of ARMOR is evaluated in Section 4.7. Finally, in Section 4.8, we summarize the conclusions of our study.

4.2 The Oracle Algorithm

In this section we present the Oracle algorithm which, as mentioned in the Introduction, "magically" knows in advance the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. Oracle takes as input the database, $\mathcal{D}$, the set of frequent itemsets, F, and its corresponding negative border, N, and outputs the supports of these itemsets by making *one scan* over the database. While the initial database layout is in the item-list (IL) format, the Oracle algorithm uses different formats during the course of its execution for efficient processing. We first describe the mechanics of the Oracle algorithm below and then move on to discuss the rationale behind its design choices in Section 4.2.2.

4.2.1 The Mechanics of Oracle

For ease of exposition, we first present the manner in which Oracle computes the supports of 1-itemsets and 2-itemsets and then move on to longer itemsets. Note, however, that the algorithm actually performs all these computations *concurrently* in one scan over the database.

Counting Singletons and Pairs

Data-Structure Description The counters of singletons (1-itemsets) are maintained in a 1-dimensional lookup array, $\mathcal{A}_1$, and that of pairs (2-itemsets), in a lower triangular 2-dimensional lookup array, $\mathcal{A}_2$ (Similar arrays are also used in Apriori [AS94, SA95] for its first two passes.) The k^{th} entry in the array $\mathcal{A}_1$ contains two fields: (1) *count*, the counter for the itemset X corresponding to the k^{th} item, and (2) *index*, the number of frequent itemsets prior to X in $\mathcal{A}_1$, if X is frequent; **null**, otherwise.

Algorithm Description The ArrayCount function shown in Figure 4.1 takes as inputs, a transaction T along with $\mathcal{A}_1$ and $\mathcal{A}_2$, and updates the counters of these arrays over T. In the ArrayCount function, the individual items in the transaction T are enumerated (lines

```
ArrayCount (T, A_1, A_2)
Input: Transaction T, Array for 1-itemsets A_1, Array for 2-itemsets A_2
Output: Arrays A_1 and A_2 with their counts updated over T
1.      Itemset T^f = null; // to store frequent items from T in IL format
2.      for each item i in transaction T
3.          A_1[i.id].count++;
4.          if A_1[i.id].index ≠ null
5.              append i to T^f
6.      for j = 1 to |T^f| // enumerate 2-itemsets
7.          for k = j + 1 to |T^f|
8.              index_1 = A_1[T^f[j].id].index // row index
9.              index_2 = A_1[T^f[k].id].index // column index
10.             A_2[index_1, index_2]++;
```

Figure 4.1: Counting Singletons and Pairs in Oracle

2–5) and for each item, its corresponding count in $\mathcal{A}_1$ is incremented (line 3). During this process, the frequent items in T are stored in a separate itemset T^f in Item-list (IL) format (line 5). We then enumerate all pairs of items contained in T^f (lines 6–10) and increment the counters of the corresponding 2-itemsets in $\mathcal{A}_2$ (lines 8–10).

Counting k-itemsets, $k > 2$

Data-Structure Description Itemsets in $F \cup N$ of length greater than 2 and their related information (counters, etc.) are stored in a DAG structure $\mathcal{G}$, which is pictorially shown in Figure 4.2 for a database with items {A, B, C, D}. Although singletons and pairs are stored in lookup arrays, as mentioned before, for expository ease, we assume that they too are stored in $\mathcal{G}$ in the remainder of this discussion.

Each itemset is stored in a separate node of $\mathcal{G}$ and is linked to the first two (in a lexicographic ordering) of its subsets. We use the terms "mother" and "father" of an itemset to refer to the (lexicographically) smaller subset and the (lexicographically) larger subset, respectively. E.g., {A, B} and {A, C} are the mother and father respectively of {A, B, C}. For each itemset X in $\mathcal{G}$, we also store with it links to those supersets of X for which X is a mother. We call this list of links as *childset*.

Since each itemset is stored in a separate node in the DAG, we use the terms "itemset"

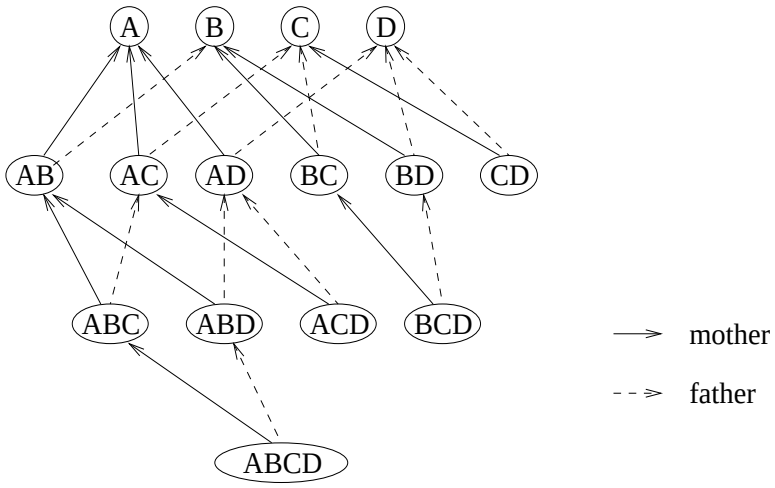

Figure 4.2: DAG Structure Containing Power Set of {A,B,C,D}

and "node" interchangeably in the remainder of this discussion. Also, we use $\mathcal{G}$ to denote the set of itemsets that are stored in the DAG structure $\mathcal{G}$.

Algorithm Description We use a *partitioning scheme* [SON95] wherein the database is logically divided into n disjoint horizontal partitions $P_1, P_2, ..., P_n$. In this scheme, itemsets being counted are enumerated only at the *end of each partition* and not after every tuple. Each partition is as large as can fit in available main memory. For ease of exposition, we assume that the partitions are equi-sized. However, we hasten to add that the technique is easily extendible to arbitrary partition sizes.

The pseudo-code of Oracle is shown in Figure 4.3 and operates as follows: The ReadNextPartition function (line 3) reads tuples from the next partition and *simultaneously* creates tid-lists (within that partition) of singleton itemsets in $\mathcal{G}$. Note that this conversion of the database from the item-list (IL) format to the tid-list (TL) format is an *on-the-fly* operation and does not change the complexity of Oracle by more than a (small) constant factor. Next, the Update function (line 5) is applied on each singleton in $\mathcal{G}$. This function takes a node M in $\mathcal{G}$ as input and updates the counts of all descendants of M to reflect their counts over the current partition. The count of any itemset within a partition is equal to the length of its corresponding tidlist (within that partition). The tidlist of an itemset can be obtained as the intersection of the tidlists of its mother and father and

```
Oracle (D, G)
Input: Database D, Itemsets to be Counted G = F ∪ N
Output: Itemsets in G with Supports
1.      n = Number of Partitions
2.      for i = 1 to n
3.          ReadNextPartition(P_i, G);
4.          for each singleton X in G
5.              Update(X);
```

Figure 4.3: The Oracle Algorithm

```
Update (M)
Input: DAG Node M
Output: M and its Descendents with Counts Updated
1.      B = convert M.tidlist to Tid-vector format
            // B is statically allocated
2.      for each node X in M.childset
3.          X.tidlist = Intersect(B, X.father.tidlist);
4.          X.count += |X.tidlist|
5.      for each node X in M.childset
6.          Update(X);
```

Figure 4.4: Updating Counts

```
Intersect (B, T)
Input: Tid-vector B, Tid-list T
Output: B ∩ T
1.      Tid-list result = φ
2.      for each tid in T
3.          offset = tid+1− (tid of first transaction in current partition)
4.          if B[offset] = 1 then
5.              result = result ∪ tid
6.      return result
```

Figure 4.5: Intersection

this process is started off using the tidlists of frequent 1-itemsets. The exact details of tidlist computation are discussed later.

We now describe the manner in which the itemsets in $\mathcal{G}$ are enumerated after reading in a new partition. The set of links, $\bigcup_{M \in \mathcal{G}} M.childset$, induce a spanning tree of $\mathcal{G}$ (e.g. consider only the solid edges in Figure 4.2). We perform a *depth first search* on this spanning tree to enumerate all its itemsets. When a node in the tree is visited, we compute the tidlists of all its children. This ensures that when an itemset is visited, the tidlists of its mother and father have already been computed.

The above processing is captured in the function Update whose pseudo-code is shown in Figure 4.4. Here, the tidlist of a given node M is first converted to the tid-vector (TV) format (line 1) discussed in Section 2.1. Then, tidlists of all children of M are computed (lines 2–4) after which the same children are visited in a depth first search (lines 5–6).

The mechanics of tidlist computation, as promised earlier, are given in Figure 4.5. The Intersect function shown here takes as input a tid-vector B and a tid-list T. Each *tid* in T is added to the result if $B[offset]$ is 1 (lines 2–5) where *offset* is defined in line 3 and represents the position of the transaction T relative to the current partition.

4.2.2 Rationale for the Oracle Design

Having described the mechanics of the Oracle design, we now move on to providing the rationale for its construction. We show that it is optimal in two respects: (1) It enumerates only those itemsets in $\mathcal{G}$ that need to be enumerated, and (2) The enumeration is performed in the most efficient way possible. The following theorem shows that there is *no wasted enumeration* of itemsets in Oracle in typical mining scenarios.

Theorem 2 *If the size of each partition is large enough that every itemset in $F \cup N$ of length greater than 2 is present at least once in it, then the only itemsets being enumerated in the Oracle algorithm are those whose counts need to be incremented in that partition.*

Proof: The first observation is that *all* 1-itemsets must be in either F or N. Hence every occurrence of a 1-itemset in the entire database needs to be accounted for in the

final output. Oracle does no more than this as it enumerates each singleton in every transaction only once (lines 2–5 in Figure 4.1).

The 2-itemsets that are enumerated (lines 6–10 in Figure 4.1) are all guaranteed to be either in F or in N since only combinations of frequent 1-itemsets are considered. Hence there is no wasted work in enumerating them.

If each partition is large enough that every itemset in $F \cup N$ of length greater than 2 is present at least once in it, then it is *necessary* to increment the counts of all these itemsets over that partition. This is precisely what is done in Oracle. Also, note that by the definition of depth first search, each node in the DAG is visited *only once*. Hence, it follows that there is no wasted enumeration of itemsets in Oracle. □

The assumption in Theorem 2 that every itemset in $F \cup N$ of length greater than 2 is present *at least once* in each partition would typically hold on large partitions. Even if this does not strictly hold, the Oracle algorithm degrades gracefully in that: If there are m itemsets that are not present in some partition, then the amount of wasted enumeration is only m.

We now move on to the second part of our proof, namely, to show that the data-structures used in the Oracle algorithm are the most efficient for the range of operations required in Oracle.

Theorem 3 *The cost of enumerating each itemset in Oracle is* $\Theta(1)$.

Proof: Since the counts of singletons and pairs are stored in direct lookup arrays, the cost of finding the counters of an arbitrary singleton or pair is $\Theta(1)$.

For an itemset X such that $|X| \geq 2$, the cost of enumerating its children is $\Theta(|X.childset|)$ since links to all nodes in $X.childset$ are available in the node containing X. Amortizing this cost over all the children results in $\Theta(1)$ cost per child. Also, X has direct pointers to its mother and father. Hence the cost of finding them in order to compute the tid-list of X is $\Theta(1)$.

Since the only operations done in Oracle in each visit to a node during the depth first search are to compute the tidlists of each of its children, the amortized cost incurred for enumerating each node is $\Theta(1)$. □

We assume that the underlying computing model is a unit cost RAM [CR73]. In this model, operations such as accessing an arbitrary element in an array and following a pointer have unit cost and cannot therefore be improved upon. Since the costs involved in the above proof are of array lookups and following pointers, the constant factor involved in the $\Theta(1)$ expression is *tight*.

While Oracle is optimal in most respects as described above, we note that there may remain some scope for improvement in the details of *tidlist computation*. That is, the Intersect function (Figure 4.5) which computes the intersection of a tid-vector B and a tid-list T requires $\Theta(|T|)$ operations. B itself was originally constructed from a tid-list, although this cost is amortized over many calls to the Intersect function. We plan to investigate in our future work whether the intersection of two sets can, in general, be computed more efficiently – for example, using **diffsets**, a novel and interesting approach suggested in [ZG01]. The diffset of an itemset X is the set-difference of the tid-list of X from that of its mother. Diffsets can be easily incorporated in Oracle – only the Update function in Figure 4.4 of Section 4.2 is to be changed to compute diffsets instead of tidlists by following the techniques suggested in [ZG01]. We found that incorporating diffsets in Oracle did not yield a significant performance gain. This was because of two reasons – (1) Our experiments were run on sparse data on which the benefit of diffsets is moderate [ZG01]. (2) The cost of finding the difference of two sets A and B is $\mathcal{O}(2 \times (|A| + |B|))$ while the intersection of a tid-list T with a tid-vector in Oracle is $\Theta(|T|)$ operations.

Advantages of Partitioning Schemes

Oracle, as discussed above, uses a partitioning scheme. An alternative commonly used in current association rule mining algorithms, especially in hashtree [AS94] based schemes, is to use a tuple-by-tuple approach. A problem with the tuple-by-tuple approach, however, is that there is considerable wasted enumeration of itemsets. The core operation in these algorithms is to determine all candidates that are subsets of the current transaction. Given that a frequent itemset X is present in the current transaction, we need to determine

all candidates that are immediate supersets of X and are also present in the current transaction. In order to achieve this, it is often necessary to enumerate and check for the presence of many more candidates than those that are actually present in the current transaction.

4.3 Performance Study

In the previous section, we have described the Oracle algorithm. In order to assess the performance of current mining algorithms with respect to the Oracle algorithm, we have chosen VIPER [SHS+00] and FP-growth [HPY00], among the latest in the suite of online mining algorithms. For completeness and as a reference point, we have also included the classical Apriori in our evaluation suite.

Our experiments cover a range of database and mining workloads, and include the typical and extreme cases considered in previous studies – the only difference is that we also consider database sizes that are *significantly larger* than the available main memory. The performance metric in all the experiments is the *total execution time* taken by the mining operation.

The databases used in our experiments were synthetically generated using the technique described in [AS94] 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 4.2. In particular, we consider databases with parameters T10.I4, T20.I12 and T40.I8 with 10 million tuples in each of them.

Parameter	Meaning	Default Values
N	Number of items	1000
T	Mean transaction length	10, 20, 40
L	Number of potentially frequent itemsets	2000
I	Mean length of potentially frequent itemsets	4, 8, 12
D	Number of transactions in the database	10M

Table 4.2: Parameter Table

We set the rule support threshold values to as low as was feasible with the available

main memory. At these low support values the number of frequent itemsets exceeded twenty five thousand! Beyond this, we felt that the number of rules generated would be enormous and the purpose of mining – to find interesting patterns – would not be served. In particular, we set the rule support threshold values for the T10.I4, T20.I12 and T40.I8 databases to the ranges (0.1%–2%), (0.4%–2%) and (1.15%–5%), respectively.

Our experiments were conducted on a 700-MHz Pentium III workstation running Red Hat Linux 6.2, configured with a 512 MB main memory and a local 18 GB SCSI 10000 rpm disk. For the T10.I4, T20.I12 and T40.I8 databases, the associated database sizes were approximately 500MB, 900MB and 1.7 GB, respectively. All the algorithms in our evaluation suite are written in C++. We implemented a basic version of the FP-growth algorithm wherein we assume that the entire FP-tree data structure fits in main memory. Finally, the partition size in Oracle was fixed to be 20K tuples.

4.3.1 Experimental Results for Current Mining Algorithms

We now report on our experimental results. We conducted two experiments to evaluate the performance of current mining algorithms with respect to the Oracle. Our first experiment was run on large (10M tuples) databases, while our second experiment was run on small (100K tuples) databases.

Experiment 1: Performance of Current Algorithms

In our first experiment, we evaluated the performance of Apriori, VIPER and Oracle algorithms for the T10.I4, T20.I12 and T40.I8 databases each containing 10M transactions and these results are shown in Figures 4.6a–c. The x-axis in these graphs represent the support threshold values while the y-axis represents the response times of the algorithms being evaluated.

In these graphs, we see that the response times of all algorithms increase exponentially as the support threshold is reduced. This is only to be expected since the number of itemsets in the output, $F \cup N$, increases exponentially with decrease in the support threshold.

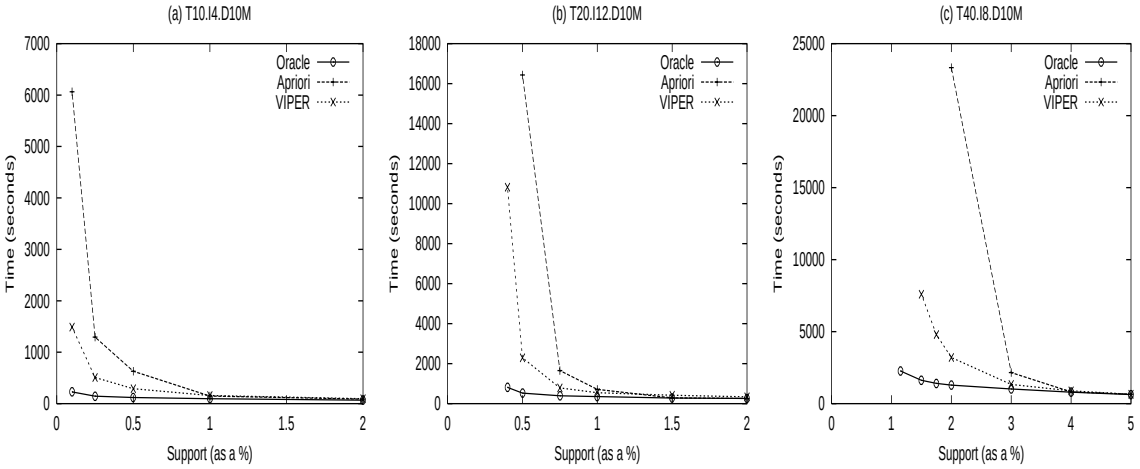

Figure 4.6: Performance of Current Algorithms (Large Databases)

We also see that there is a considerable gap in the performance of both Apriori and VIPER with respect to Oracle. For example, in Figure 4.6a, at a support threshold of 0.1%, the response time of VIPER is more than 6 times that of Oracle whereas the response time of Apriori is more than 26 times!

In this experiment, we could not evaluate the performance of FP-growth because it did not complete in any of our runs on large databases due to its heavy and database size dependent utilization of main memory. The reason for this is that FP-growth stores the database itself in a condensed representation in a data structure called FP-tree. In [HPY00], the authors briefly discuss the issue of constructing disk-resident FP-trees. We however, did not take this into account in our implementation. We return to this issue later in Section 4.3.1.

Experiment 2: Small Databases

Since, as mentioned above, it was not possible for us to evaluate the performance of FP-growth on large databases due to its heavy utilization of main memory, we evaluated the performance of FP-growth and other current algorithms on small databases consisting of 100K transactions. The results of this experiment are shown in Figures 4.7a–c, which correspond to the T10.I4, T20.I12 and T40.I8 databases, respectively.

In these graphs, we first see there continues to be a considerable gap in the performance

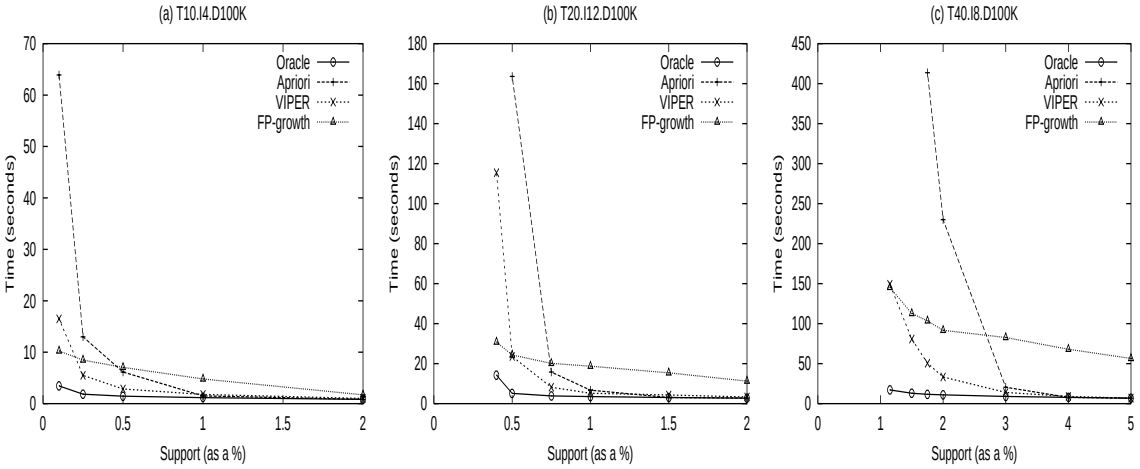

Figure 4.7: Performance of Current Algorithms (Small Databases)

of current mining algorithms with respect to Oracle. For example, for the T40.I8 database, the response time of FP-growth is more than 8 times that of Oracle for the entire support threshold range.

Second, although FP-growth does well at low supports, its performance is worse than Apriori for high supports. These results are inconsistent with those shown in [HPY00] where it was shown that FP-growth consistently performs better than Apriori for the entire support range. While this could possibly be due to differences between our respective implementations of FP-growth and/or Apriori, we feel that there are logical reasons for this behaviour as explained below.

At high supports Apriori typically performs only two passes over the data since with these supports there are usually no frequent itemsets of length greater than two. In these cases, the first pass of Apriori is *identical* to the preprocessing pass in FP-growth in which all frequent singletons are obtained. The second pass of Apriori is quite efficient since the counts of candidate 2-itemsets are maintained in a 2-dimensional lookup array. FP-growth, on the other hand, constructs an FP-tree during the second pass. The FP-tree is updated on a tuple-by-tuple basis. Each node in the FP-tree contains an *item-name* field. A critical operation during FP-tree construction is to find the child of a node given a key *item-name*. If these keys are stored in lookup-arrays, the memory requirements of FP-tree would be still worse. The alternative is to use an indexing data structure such as

a red-black tree or a skip-list that requires $\mathcal{O}(\log n)$ time to perform the find operation, but this would make the FP-tree construction slow. Even assuming that the cost of FP-tree construction is equal to the second pass of Apriori, FP-growth still needs to mine the FP-tree. Hence FP-growth finally loses out at high supports.

4.4 The ARMOR Algorithm

```
ARMOR (D, I, minsup)
Input: Database D, Set of Items I, Minimum Support minsup
Output: F ∪ N with Supports
1.      n = Number of Partitions

        //—– First Pass —–
2.      G = I      // candidate set (in a DAG)
3.      for i = 1 to n
4.            ReadNextPartition(P_i, G);
5.            for each singleton X in G
6.                  X.count += |X.tidlist|
7.                  Update1(X, minsup);

        //—– Second Pass —–
8.      RemoveSmall(G, minsup);
9.      OutputFinished(G, minsup);
10.     for i = 1 to n
11.           if (all candidates in G have been output)
12.                 exit
13.           ReadNextPartition(P_i, G);
14.           for each singleton X in G
15.                 Update2(X, minsup);
```

Figure 4.8: The ARMOR Algorithm

In the previous section, our experimental results have shown that there is a considerable gap in the performance between the Oracle and existing mining algorithms. We now move on to describe our new mining algorithm, ARMOR (Association Rule Mining based on ORacle). In this section, we overview the main features and the flow of execution of ARMOR – the details of candidate generation are deferred to the following section.

The guiding principle in our design of the ARMOR algorithm is that we consciously

make an attempt to determine the *minimal amount of change* to Oracle required to result in an online algorithm. This is in marked contrast to the earlier approaches which designed new algorithms by trying to address the limitations of *previous* online algorithms. That is, we approach the association rule mining problem from a completely different perspective.

In ARMOR, as in Oracle, the database is conceptually partitioned into n disjoint blocks $P_1, P_2, ..., P_n$. At most *two* passes are made over the database. In the first pass we form a set of candidate itemsets, $\mathcal{G}$, that is guaranteed to be a superset of the set of frequent itemsets. During the first pass, the counts of candidates in $\mathcal{G}$ are determined over each partition in exactly the same way as in Oracle by maintaining the candidates in a DAG structure. The 1-itemsets and 2-itemsets are stored in lookup arrays as in Oracle. But unlike in Oracle, candidates are inserted and removed from $\mathcal{G}$ at the end of each partition. Generation and removal of candidates is done *simultaneously* while computing counts. The details of candidate generation and removal during the first pass are described in Section 4.5. For ease of exposition we assume in the remainder of this section that all candidates (including 1-itemsets and 2-itemsets) are stored in the DAG.

Along with each candidate X, we also store the following three integers as in the CARMA algorithm [Hid99]: (1) $X.count$: the number of occurrences of X *since* X was last inserted in $\mathcal{G}$. (2) $X.firstPartition$: the index of the partition *at* which X was inserted in $\mathcal{G}$. (3) $X.maxMissed$: upper bound on the number of occurrences of X *before* X was inserted in $\mathcal{G}$. $X.firstPartition$ and $X.maxMissed$ are computed when X is inserted into $\mathcal{G}$ in a manner identical to CARMA.

While the CARMA algorithm works on a tuple-by-tuple basis, we have adapted the semantics of these fields to suit the partitioning approach. If the database scanned so far is d (refer Table 2.1), then the support of any candidate X in $\mathcal{G}$ will lie in the range $[X.count/|d|, (X.maxMissed + X.count)/|d|]$ [Hid99]. These bounds are denoted by $\mathsf{minSupport}(X)$ and $\mathsf{maxSupport}(X)$, respectively. We define an itemset X to be d-frequent if $\mathsf{minSupport}(X) \geq minsup$. Unlike in the CARMA algorithm where only d-frequent itemsets are stored at any stage, the DAG structure in ARMOR contains other candidates, including the *negative border* of the d-frequent itemsets, to ensure efficient

candidate generation. The details are given in Section 4.5.

At the end of the first pass, the candidate set $\mathcal{G}$ is pruned to include only d-frequent itemsets and their negative border. The counts of itemsets in $\mathcal{G}$ over the entire database are determined during the second pass. The counting process is again identical to that of Oracle. No new candidates are generated during the second pass. However, candidates may be removed. The details of candidate removal in the second pass is deferred to Section 4.5.1.

The pseudo-code of ARMOR is shown in Figure 4.8 and is explained below.

4.4.1 First Pass

At the beginning of the first pass, the set of candidate itemsets $\mathcal{G}$ is initialized to the set of singleton itemsets (line 2). The ReadNextPartition function (line 4) reads tuples from the next partition and simultaneously creates tid-lists of singleton itemsets in $\mathcal{G}$.

After reading in the entire partition, the Update1 function (details in Section 4.5) is applied on each singleton in $\mathcal{G}$ (lines 5–7). It increments the counts of existing candidates by their corresponding counts in the current partition. It is also responsible for generation and removal of candidates.

At the end of the first pass, $\mathcal{G}$ contains a superset of the set of frequent itemsets. For a candidate in $\mathcal{G}$ that has been inserted at partition P_j, its count over the partitions $P_j, ..., P_n$ will be available.

4.4.2 Second Pass

At the beginning of the second pass, candidates in $\mathcal{G}$ that are neither d-frequent nor part of the current negative border are removed from $\mathcal{G}$ (line 8). For candidates that have been inserted in $\mathcal{G}$ at the first partition, their counts over the entire database will be available. These itemsets with their counts are output (line 9). The OutputFinished function also performs the following task: If it outputs an itemset X and X has no supersets left in $\mathcal{G}$, X is removed from $\mathcal{G}$.

During the second pass, the ReadNextPartition function (line 13) reads tuples from the

next partition and creates tid-lists of singleton itemsets in $\mathcal{G}$. After reading in the entire partition, the Update2 function (details in Section 4.5.1) is applied on each singleton in $\mathcal{G}$ (lines 14–15). Finally, before reading in the next partition we check to see if there are any more candidates. If not, the mining process terminates.

4.5 Candidate Generation in ARMOR

ARMOR utilizes a technique from *incremental* mining algorithms [PH00, T$^+$97, F$^+$97] in order to generate candidates efficiently. These algorithms are designed to efficiently derive the current mining output by utilizing previous mining results when a database has been updated with an increment. ARMOR treats the database scanned so far, d, as the "original database" and the current partition being processed as the "increment". Let d^+ denote the portion of the database scanned so far *including* the current partition being processed (see Table 2.1 in Chapter 2). Let F^d and F^{d^+} be the sets of frequent itemsets over d and d^+, respectively, and N^d and N^{d^+} be their corresponding negative borders. In this context, it is shown in [T$^+$97] that:

Theorem 4 *If X is an itemset that is not in F^d but is in F^{d^+}, then there must be some subset x of X which was in N^d and is now in F^{d^+}.*

The itemsets that move from N^d to F^{d^+} are called *promoted borders*. The above Theorem then means that the only candidates that need to be generated are those that are supersets of the promoted borders. We use the term *expanding* a promoted border P to denote the process of generating the required supersets of P.

We present now a technique for efficiently expanding a promoted border. Our technique is captured in the Expand function presented in Figure 4.9, the inputs to which are P, the promoted border to be expanded and $\mathcal{G}$, the current set of candidates. The Expand function is similar to the AprioriGen function [AS94] since the siblings of P are exactly those itemsets in $\mathcal{G}$ that differ from P in the last item. However, the Expand function and its usage differs from AprioriGen in that: (1) It is applied dynamically whenever a candidate that was in the negative border becomes d-frequent; (2) It is applied to individual

itemsets, whereas the AprioriGen function is applied to sets of itemsets; (3) It performs a parent based pruning optimization unlike AprioriGen which enumerates all immediate subsets of a candidate in order to prune it.

```
Expand (P, G)
Input: Promoted Border P, DAG G
    for each sibling X of P in G
        if (X is d-frequent) then
            S = P ∪ X      // new candidate
            Insert S into G as a child of P
```

Figure 4.9: Expanding a Promoted Border

At first glance, it may appear surprising that we do not consider the same pruning strategy as of AprioriGen in Expand. The reason we do not do so is because it results in significant overheads due to the dynamic and incremental manner in which candidate generation occurs in Expand. We illustrate this with the following example: Consider the situation in which the itemsets $\{U, V\}$ and $\{U, W\}$ are d-frequent but $\{V, W\}$ is not. Then $\{U, V, W\}$ will not be in $\mathcal{G}$ if Apriori-type pruning were incorporated in Expand. If $\{V, W\}$ also becomes d-frequent, then $\{U, V, W\}$ will need to be added to $\mathcal{G}$. But $\{V, W\}$ cannot be combined with another itemset that differs only in the *last* item to produce $\{U, V, W\}$. This means that if we incorporate Apriori-type pruning, the Expand function needs to combine $\{V, W\}$ with itemsets that differ from it in *any one* item.

From the above discussion, it is clear that incorporating Apriori-type pruning in the Expand function, results in significant cost for two reasons: (1) It requires a separate traversal of the DAG structure to find all itemsets that differ from a given itemset in *any one* item. (2) All immediate subsets of a given itemset need to be searched for in the DAG.

Without Apriori-type pruning, in the above example, $\{U, V, W\}$ would have already been in $\mathcal{G}$ regardless of whether $\{V, W\}$ is d-frequent or not since it would not have been pruned due to the absence of $\{V, W\}$. Therefore, when $\{V, W\}$ becomes d-frequent, it is not necessary to regenerate $\{U, V, W\}$.

Due to the above reasons we do not incorporate Apriori-type pruning in ARMOR. Instead, a candidate is automatically pruned if one of its parents is not d-frequent since it would not even be generated in the first place. Our experiments (Section 4.7) showed that the number of additional candidates generated in ARMOR compared to Apriori's $|F \cup N|$ candidates was marginal – the worst case being about *ten percent* more.

The Expand function is incorporated into ARMOR by calling it from the Update1 function that is invoked for each partition scanned during the first pass. The Update1 function is presented in Figure 4.10 and is explained below.

```
Update1 (M, minsup)
Input: DAG Node M, Minimum Support minsup
Output: M and its Descendents Updated
1.      B = convert M.tidlist to Tid-vector format
            // B is statically allocated
2.      for each node X in M.childset
3.          X.tidlist = Intersect(B, X.father.tidlist);
4.          X.count += |X.tidlist|
5.      for each node X in M.childset
6.          if maxSupport(X) ≤ minsup then
7.              if |X.childset| > 0      // already expanded
8.                  remove all supersets of X reachable from X in G
9.          else
10.             if |X.childset| = 0      // not yet expanded
11.                 Expand(X);
12.     for each node X in M.childset
13.         Update1(X);
```

Figure 4.10: Updating Counts

The manner in which the counts of candidates are computed in Update1 is exactly the same as that in Update (described in Section 4.2). The extra processing in Update1 is only to generate and remove candidates dynamically. This is done in one enumeration of all children of a given node M (lines 5–11). For each child X that is enumerated, if it has supersets but is not d-frequent, then we remove all supersets of X that are reachable from X in the DAG (lines 6–8). Note that X itself is not removed since it could be part of the current negative border. On the other hand, if X is d-frequent and has not yet

been expanded, then it is now expanded by calling the Expand function (lines 10–11).

4.5.1 Candidate Removal During Second Pass

A candidate X is removed during the second pass whenever the following two conditions are satisfied: (1) The count of X over the entire database is available, which becomes true when $X.firstPartition$ is the next partition to be processed; and (2) X has no supersets in $\mathcal{G}$.

We now describe the Update2 function (called from ARMOR in Figure 4.8), which is responsible for removing candidates as described above. The Update2 function increments the counts of existing candidates by their corresponding counts in the current partition in a manner identical to that of the Update function of Oracle (described in Section 4.2). It differs from Update only in that it also outputs candidates whose counts over the entire database are known. If it outputs an itemset X and X has no supersets left in $\mathcal{G}$, X is removed from $\mathcal{G}$.

4.6 Memory Utilization in ARMOR

In the design and implementation of ARMOR, we have opted for speed in most decisions that involve a space-speed tradeoff. Therefore, the main memory utilization in ARMOR is certainly more as compared to algorithms such as Apriori. However, in the following discussion, we show that the memory usage of ARMOR is well within the reaches of current machine configurations. This is also experimentally confirmed in the next section.

The main memory consumption of ARMOR comes from the following sources: (1) The 1-d and 2-d arrays for storing counters of singletons and pairs, respectively; (2) The DAG structure for storing counters of longer itemsets, including tidlists of those itemsets, and (3) The current partition.

The total number of entries in the 1-d and 2-d arrays and in the DAG structure corresponds to the number of candidates in ARMOR, which as we have discussed in Section 4.5, is only marginally more than $|F \cup N|$. For the moment, if we disregard

the space occupied by tidlists of itemsets, then the amortized amount of space taken by each candidate is a small constant (about 10 integers for the dag and 1 integer for the arrays). E.g., if there are 1 million candidates in the dag and 10 million in the array, the space required is about 80MB. Since the environment we consider is one where the pattern lengths are small, the number of candidates will typically be comparable to or well within the available main memory. [XD99] discusses alternative approaches when this assumption does not hold.

Regarding the space occupied by tidlists of itemsets, note that ARMOR only needs to store tidlists of d-frequent itemsets. The number of d-frequent itemsets is of the same order as the number of frequent itemsets, $|F|$. The total space occupied by tidlists while processing partition P_i is then bounded by $|F| \times |P_i|$ integers. E.g., if $|F| = 5K$ and $|P_i| = 20K$, then the space occupied by tidlists is bounded by about 400MB. We assume $|F|$ to be in the range of a few thousands at most because otherwise the total number of rules generated would be enormous and the purpose of mining would not be served. Note that the above bound is very pessimistic. Typically, the lengths of tidlists are much smaller than the partition size, especially as the itemset length increases.

Main memory consumed by the current partition is small compared to the above two factors. E.g., If each transaction occupies 1KB, a partition of size 20K would require only 20MB of memory. Even in these extreme examples, the total memory consumption of ARMOR is 500MB, which is acceptable on current machines.

Therefore, *in general we do not expect memory to be an issue* for mining market-basket databases using ARMOR. Further, even if it does happen to be an issue, it is easy to modify ARMOR to free space allocated to tidlists at the expense of time: $M.tidlist$ can be freed after line 3 in the Update function shown in Figure 4.4.

A final observation to be made from the above discussion is that the main memory consumption of ARMOR is proportional to the size of the *output* and does not "explode" as the input problem size increases.

4.7 Experimental Results for ARMOR

We evaluated the performance of ARMOR with respect to Oracle on a variety of databases and support characteristics. We now report on our experimental results for the same performance model described in Section 4.3. Since Apriori, FP-growth and VIPER have already been compared against Oracle in Section 4.3.1, we do not repeat those observations here, but focus on the performance of ARMOR w.r.t. that of Oracle. This lends to the visual clarity of the graphs. We hasten to add that ARMOR does outperform the other algorithms.

4.7.1 Experiment 3: Performance of ARMOR

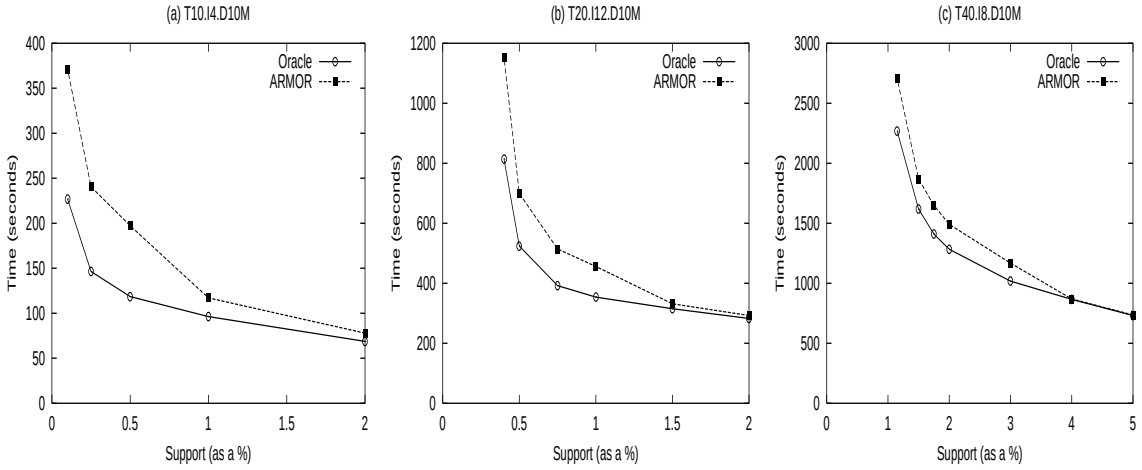

Figure 4.11: Performance of ARMOR (Synthetic Datasets)

In this experiment, we evaluated the response time performance of the ARMOR and Oracle algorithms for the T10.I4, T20.I12 and T40.I8 databases each containing 10M transactions and these results are shown in Figures 4.11a–c.

In these graphs, we first see that ARMOR's performance is close to that of Oracle for high supports. This is because of the following reasons: The density of the frequent itemset distribution is sparse at high supports resulting in only a few frequent itemsets with supports "close" to *minsup*. Hence, frequent itemsets are likely to be locally frequent within most partitions. Even if they are not locally frequent in a few partitions, it is very

likely that they are still d-frequent over these partitions. Hence, their counters are updated even over these partitions. Therefore, the complete counts of most candidates would be available at the end of the first pass resulting in a "light and short" second pass. Hence, it is expected that the performance of ARMOR will be close to that of Oracle for high supports.

Since the frequent itemset distribution becomes dense at low supports, the above argument does not hold in this support region. Hence we see that ARMOR's performance relative to Oracle decreases at low supports. But, what is far more important is that ARMOR consistently performs within a *factor of two* of Oracle. This is highlighted in Table 4.3 where we show the ratios of the performance of ARMOR to that of Oracle for the lowest support values considered for each of the databases.

Database	$minsup(\%)$	ARMOR (seconds)	Oracle (seconds)	ARMOR/Oracle
T10.I4.D10M	0.1	371.44	226.99	1.63
T20.I12.D10M	0.4	1153.42	814.01	1.41
T40.I8.D10M	1.15	2703.64	2267.26	1.19

Table 4.3: Worst-case Efficiency of ARMOR w.r.t Oracle

4.7.2 Experiment 4: Memory Utilization in ARMOR

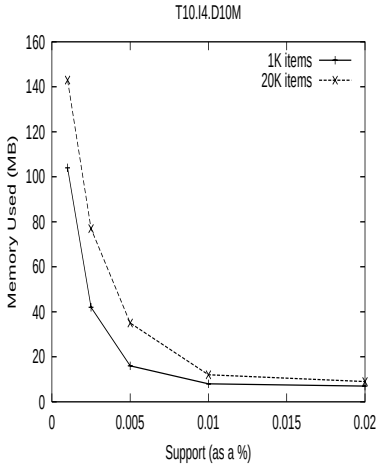

Figure 4.12: Memory Utilization in ARMOR

The previous experiments were conducted with the total number of items, N, being set to 1K. In this experiment we set the value of N to 20K items for the T10.I4 database – this environment represents an extremely stressful situation for ARMOR with regard to memory utilization due to the very large number of items. Figure 4.12 shows the memory utilization of ARMOR as a function of support for the $N = 1$K and $N = 20$K cases. We see that the main memory utilization of ARMOR scales well with the number of items. For example, at the 0.1% support threshold, the memory consumption of ARMOR for $N = 1$K items was 104MB while for $N = 20$K items, it was 143MB – an increase in less than 38% for a 20 times increase in the number of items! The reason for this is that the main memory utilization of ARMOR does not depend directly on the number of items, but only on the size of the output, $F \cup N$, as discussed in Section 4.6.

4.7.3 Experiment 5: Real Datasets

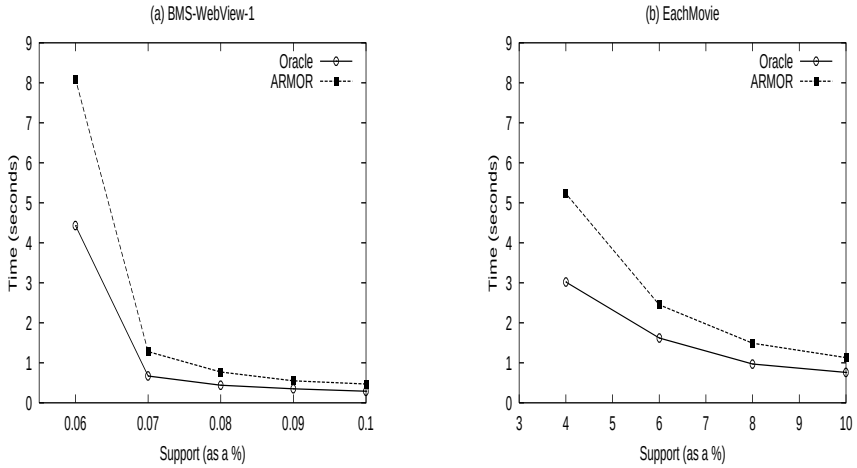

Figure 4.13: Performance of Armor (Real Datasets)

Despite repeated efforts, we were unable to obtain large real datasets that conform to the sparse nature of market basket data since such data is not publicly available due to proprietary reasons. The datasets in the UC Irvine public domain repository [BM98] which are commonly used in data mining studies were not suitable for our purpose since they are dense and have long patterns. We could however obtain two datasets – BMS-WebView-1, a clickstream data from Blue Martini Software [ZKM01] and EachMovie, a

movie database from Compaq Equipment Corporation [com97], which we transformed to the format of boolean market basket data. The resulting databases had 59,602 and 61,202 transactions respectively with 870 and 1648 distinct items.

We set the rule support threshold values for the BMS-WebView-1 and EachMovie databases to the ranges (0.06%–0.1%) and (3%–10%), respectively. The results of these experiments are shown in Figures 4.13a–b. We see in these graphs that the performance of ARMOR continues to be within twice that of Oracle. The ratio of ARMOR's performance to that of Oracle at the lowest support value of 0.06% for the BMS-WebView-1 database was 1.83 whereas at the lowest support value of 3% for the EachMovie database it was 1.73.

4.7.4 Discussion of Experimental Results

We now explain the reasons as to why ARMOR should typically perform within a factor of two of Oracle. First, we notice that the only difference between the single pass of Oracle and the first pass of ARMOR is that ARMOR continuously generates and removes candidates. Since the generation and removal of candidates in ARMOR is dynamic and efficient, this does not result in a significant additional cost for ARMOR.

Since candidates in ARMOR that are neither d-frequent nor part of the current negative border are continuously removed, any itemset that is locally frequent within a partition, but not globally frequent in the entire database is likely to be removed from G during the course of the first pass (unless it belongs to the current negative border). Hence the resulting candidate set in ARMOR is a good approximation of the required mining output. In fact, in our experiments, we found that in the worst case, the number of candidates counted in ARMOR was only about *ten percent* more than the required mining output.

The above two reasons indicate that the cost of the first pass of ARMOR is only slightly more than that of (the single pass in) Oracle.

Next, we notice that the only difference between the second pass of ARMOR and (the single pass in) Oracle is that in ARMOR, candidates are continuously removed. Hence

the number of itemsets being counted in ARMOR during the second pass quickly reduces to much less than that of Oracle. Moreover, ARMOR does not necessarily perform a complete scan over the database during the second pass since the second pass ends when there are no more candidates. Due to these reasons, we would expect that the cost of the second pass of ARMOR is usually less than that of (the single pass in) Oracle.

Since the cost of the first pass of ARMOR is usually only slightly more than that of (the single pass in) Oracle and that of the second pass is usually less than that of (the single pass in) Oracle, it follows that ARMOR will typically perform within a factor of two of Oracle.

In summary, due to the above reasons, it appears unlikely for it to be possible to design algorithms that substantially reduce either the number of database passes or the number of candidates counted. These represent the primary bottlenecks in association rule mining. Further, since ARMOR utilizes the same itemset counting technique of Oracle, further overall improvement without domain knowledge seems extremely difficult. Finally, even though we have not proved optimality of Oracle with respect to tidlist intersection, we note that any smart intersection techniques that may be implemented in Oracle can also be used in ARMOR.

4.8 Conclusions

A variety of novel algorithms have been proposed in the recent past for the efficient mining of association rules, each in turn claiming to outperform its predecessors on a set of standard databases. In this chapter, our approach was to quantify the algorithmic performance of association rule mining algorithms with regard to an idealized, but practically infeasible, "Oracle". The Oracle algorithm utilizes a partitioning strategy to determine the supports of itemsets in the required output. It uses direct lookup arrays for counting singletons and pairs and a DAG data-structure for counting longer itemsets. We have shown that these choices are optimal in that only required itemsets are enumerated and that the cost of enumerating each itemset is $\Theta(1)$. Our experimental results showed that there was a substantial gap between the performance of current mining algorithms and

that of the Oracle.

We also presented a new online mining algorithm called ARMOR (Association Rule Mining based on ORacle), that was constructed with minimal changes to Oracle to result in an online algorithm. ARMOR utilizes a new method of candidate generation that is dynamic and incremental and is guaranteed to complete in two passes over the database. Our experimental results demonstrate that ARMOR performs within a *factor of two* of Oracle.

Chapter 5

Conciseness of Mining Results

5.1 Introduction

The gigantic number of association rules generated in typical mining operations makes it impractical for manual examination of the mining output [LHM99]. While this is true for sparse datasets, it is often impractical to even generate all frequent itemsets and their associated supports for dense datasets. For instance, if the length of frequent itemsets grows beyond a mere thirty, the total number of frequent itemsets exceeds one billion!

Recent approaches (such as those described in Chapter 3) to handle the information overload produced as mining output follow the strategy of pruning "uninteresting" rules. These studies are based on the observation that, in practice, many rules have the same predictive power as other rules with fewer items, making them redundant.

Among the earlier approaches, the *closed itemset* framework [Zak00, TPBL00] is attractive in that both the identities and supports of all frequent itemsets can be derived *completely* from the frequent closed itemsets, which is a subset of the frequent itemsets. However, this framework, as shown in this chapter, suffers from the drawback that its usefulness critically depends on the presence of frequent itemsets that have supersets with *exactly the same support*. This means that even minor changes in the database can result in a significant increase in the number of frequent closed itemsets. For example, adding a select 438 transactions to the 8,124 transaction mushroom dataset (from the UC Irvine

Repository) caused the number of closed frequent itemsets at a support threshold of 20% to increase from 1,390 to 15,541 – a factor of 11 times!

In this chapter we show that the number of redundant rules is *far more* than what was previously estimated. We propose the *generalized closed itemset framework* (also referred to as g-closed itemset framework) that builds upon the closed itemset framework and overcomes its above mentioned limitation. In our scheme, the supports of frequent itemsets can be estimated within a *deterministic*, user-specified "tolerance" factor. Our experimental results show that even by allowing for a very small tolerance, we produce exponentially fewer rules for most datasets and support specifications than the closed itemsets, which are themselves much fewer than the total number of frequent itemsets.

Our scheme is also more robust to the database contents. For the same mushroom example mentioned above, the number of frequent g-closed itemsets only increased from 1,386 to 2243, for a tolerance of 0.05%. We feel that this tolerance factor is negligible since it is 400 times smaller than the minimum support threshold of 20%.

In our scheme, it is possible to correctly estimate the *identities* of all frequent itemsets. No false negatives are ever produced, although some "borderline" infrequent itemsets may be incorrectly identified as frequent. Borderline itemsets refers to those infrequent itemsets that would become frequent if the support threshold were reduced by an amount equal to the tolerance factor. We feel that this is acceptable in most mining scenarios for tolerance factors that are much less than the minimum support threshold.

We provide theoretical arguments to show why the g-closed itemset scheme works and substantiate these observations with experimental evidence. Our experiments were run on a variety of databases, both real and synthetic as well as sparse and dense, to confirm that the scheme works across a broad spectrum of database schemas and contents. On sparse datasets, hardly any pruning occurs when the closed itemset scheme is used. On the other hand, the pruning achieved by our scheme is quite significant even for these datasets. On dense datasets, the pruning achieved by our scheme is much more dramatic than that achieved by the closed itemset approach.

Our scheme can be used in one of two ways: (1) as a post-processing step of the

mining process (like in [LHM99, DL98, SLR99, CS02, TK00]), or (2) as an integrated solution (like in [ZH02, PBTL99]). We show that our scheme can be integrated into both levelwise algorithms as well as the more recent two-pass mining algorithms. We chose the classical Apriori algorithm [AS94] as a representative of the levelwise algorithms and the ARMOR algorithm (presented in Chapter 4), as a representative of the class of two-pass mining algorithms. Integration into Apriori yields a new algorithm, ***g*-Apriori** and into ARMOR, yields ***g*-ARMOR**. Our experimental results show that these integrations often result in a significant reduction in response-time, especially for dense datasets.

We note that integration of our scheme into two-pass mining algorithms is a novel and important contribution because two-pass algorithms have several advantages over Apriori-like levelwise algorithms. These include: (1) significantly less I/O cost, (2) significantly better overall performance as shown in [P+01, PH02b], and (3) the ability to provide approximate supports of frequent itemsets at the end of the first pass itself, as in [Hid99, PH02b]. This ability is an essential requirement for mining *data streams* [MM02] as it is infeasible to perform more than one pass over the complete stream.

For ease of exposition, we will use the notation shown in Table 2.1 of Chapter 2 in the remainder of this chapter. The relevant part of this table has been reproduced in Table 5.1 for convenience.

$\mathcal{I}$	Set of items in the database
$\mathcal{D}$	Database of customer purchase transactions
$minsup$	User-specified minimum rule support
$minconf$	User-specified minimum rule support
$support(X)$	Support of itemset X
$t(X)$	Tidset of itemset X
$i(T)$	Set of items that are common to transactions in T
$c(X)$	Closed itemset corresponding to itemset X
$g(X)$	g-Closed itemset corresponding to itemset X
ϵ	Tolerance factor
C_k	Set of candidate k-itemsets
G_k	Set of frequent k-generators
G	Set of all frequent generators produced so far
$\mathcal{G}$	DAG structure to store candidates during g-ARMOR execution

Table 5.1: Notation (from Table 2.1)

5.1.1 Organization

The remainder of this chapter is organized as follows: In Section 5.2 we review the concept of closed itemsets and identify its limitations. In Section 5.3 we present the g-closed itemset framework that overcomes these limitations. Then, in Section 5.4, we describe the process of rule generation given the frequent g-closed itemsets. We incorporate the g-closed itemset framework into the Apriori and ARMOR algorithms resulting in g-Apriori and g-ARMOR in Sections 5.5 and 5.6, respectively. The performance of g-Apriori, g-ARMOR and of the g-closed itemset framework is evaluated in Section 5.7. Finally, in Section 5.8, we summarize the conclusions of our study.

5.2 Closed Itemsets

In this section we briefly review the concept of closed itemsets [ZH02, PBTL99], identify its limitations, and set the stage for extending it to remove these limitations.

5.2.1 Background

The tidset of an itemset, X, is defined as: $t(X)$ = set of tuple identifiers of transactions that contain X. Similarly, the itemset of a tidset, T, is defined as: $i(T)$ = set of items that are common to all transactions in T. Then $c(X) = i(t(X))$ is a closure operator and thereby satisfies the following properties: (1) Extension: $X \subseteq c(X)$; (2) Monotonicity: if $X \subseteq Y$, then $c(X) \subseteq c(Y)$; (3) Idempotency: $c(c(X)) = c(X)$.

Definition 1 *An itemset, X, is* closed *iff $c(X) = X$.*

For ease of exposition, we will refer to itemsets that are not closed as *open*. For every open itemset, Y, there exists a superset, $c(Y)$, of Y that is closed. This follows from the idempotency property above. $c(Y)$ is called as the *closure* of Y and its support is equal to the support of Y. The support of an itemset Y is given by $support(Y) = |t(Y)|/|\mathcal{D}|$ where $|\mathcal{D}|$ is the number of database transactions.

5.2.2 Exact Equality of Supports

The closed itemset framework is useful when the number of closed frequent itemsets is significantly less than the total number of frequent itemsets since the remaining open frequent itemsets can be pruned. It is therefore desirable, in view of this framework, to have a large number of open frequent itemsets. However, we opine that this restriction is too stringent since, as proved below, an itemset can be open only if it has a superset with *exactly* equal support.

Theorem 5 *An itemset, X, is open iff it has a proper superset with equal support.*

Proof: By definition, $c(X) = i(t(X))$ consists of those items that are present in all transactions in which X is present. Hence, if X is open (i.e. $c(X) \neq X$), it means that there is an item $j \notin X$ that is present in all transactions in which X is present. This implies that X has a proper superset, $X \cup \{j\}$, with exactly the same support.

Correspondingly, if X has a proper superset, Y, with exactly the same support, then the items in $Y - X$ are present in all transactions in which X is present. These items would be present in the closure of X, but not in X. Hence, X is open. □

The above result causes the closed itemset approach to be highly sensitive to the data being mined. For example – as mentioned in Section 5.1, the addition of a small number of transactions to the mushroom dataset caused the number of frequent closed itemsets to increase by an order of magnitude.

5.2.3 Propagation of Openness

Despite the above drawback, if at all an itemset X, happens to be open then the following theorem shows that many supersets of X will also be open. That is, the *openness* of an itemset propagates up the itemset lattice. This property of openness propagation will carry over to the g-closed itemset framework as will be shown in Section 5.3.

Theorem 6 *If X and Y are itemsets such that $Y \supseteq X$ and $support(Y) = support(X)$, then for every itemset $Z : Z \supseteq X$, $support(Z) = support(Y \cup Z)$.*

Proof: Since $Y \supseteq X$ and $support(Y) = support(X)$, it is clear that Y is present in every transaction that contains X. Further, since $Z \supseteq X$, X is present in every transaction that contains Z. Hence, it follows that Y is present in every transaction that contains Z. Therefore, $support(Z) = support(Y \cup Z)$. □

Combining this result with Theorem 5, it follows that if X is an open itemset having a superset Y with equal support, then every itemset $Z : Z \supseteq X \bigwedge Z \not\supseteq Y$ is also open.

5.2.4 Equal Support Pruning

Theorem 6 suggests a general technique to incorporate in mining algorithms: If an itemset X, has an immediate superset Y[1] with equal support, then prune Y and avoid generating any candidates that are supersets of Y. The support of any of these pruned itemsets, say W, will be identical to one of its subsets, $(W - Y) \cup X$. We refer to this technique as *equal support pruning*.

In fact, the pruning technique adopted in the A-Close algorithm [PBTL99] for generating frequent closed itemsets is equivalent to the scheme outlined above. In this algorithm, equal support pruning is combined with the classical Apriori-type subset-support based pruning. The resulting unpruned itemsets are referred to as *generators* since they are later used to generate the frequent closed itemsets. For any closed itemset Y, the shortest itemset X for which $c(X) = Y$ is referred to as the generator of Y.

5.2.5 Generating Closed Itemsets

We now present a simple technique to generate closed itemsets from their generators. This technique does not involve an additional database scan as is required in the A-Close algorithm. Also, the technique will directly carry over to the g-closed itemset case. For any closed itemset Y with generator X, the following theorem enables us to determine $Y - X$ using information that could be gathered while performing equal support pruning. We refer to $Y - X$ as $X.pruned$.

[1] An immediate superset of X is a superset of X with cardinality $|X| + 1$. Likewise, an immediate subset of X is one with cardinality $|X| - 1$.

Theorem 7 *Let Y be a closed itemset and $X \subseteq Y$ be the generator of Y. Then for each item A in $Y - X$, $X \cup \{A\}$ would be pruned by the equal support pruning technique. No other immediate supersets of X would be pruned by the same technique.*

Proof: Since X is the generator for Y, they have the same support. It follows that for every item A in $Y - X$, $X \cup \{A\}$ would have the same support as X. Hence it would be pruned by the equal support pruning technique.

To prove the second part, assume that there is some immediate superset Z, of X, $Z \not\subseteq Y$ that is pruned using the equal support pruning technique. Then Z has the same support as X and must therefore be present in every transaction that contains X. By definition of closure, $Z \subseteq Y$. Hence proved by contradiction. □

Therefore, in order to generate a closed itemset Y from its generator X, it is sufficient to compute $X.pruned$ while performing equal support pruning. Note that if some subset W of X had a proper subset V with equal support, then W would be pruned using the equal support pruning technique. It would then be necessary to include all the items in $V.pruned$ in $X.pruned$. That is, the *pruned* value of any itemset needs to be propagated to all its supersets.

5.3 Generalized Closed Itemsets

The closed itemset framework is attractive in that both the identities and supports of all frequent itemsets can be derived *completely* from a smaller subset. But, in order to provide this feature, the framework requires exact equality between the supports of some critical itemsets, as discussed earlier. In this section, we introduce the *generalized closed itemset framework* (also referred to as g-closed itemset framework) – a generalization of the closed itemset framework that overcomes its limitation of exact support equality. Relationship of our scheme to existing work was discussed in Section 3.2 of Chapter 3.

In order to achieve the above generalization, we need to marginally sacrifice the ability to completely derive both identities and supports of frequent itemsets. We however ensure the following: (1) The supports of all frequent itemsets can be derived within a

deterministic, user-specified "tolerance" factor. (2) The identities of all frequent itemsets can be derived from the smaller set. However, some borderline infrequent itemsets may be declared as frequent. (3) If the tolerance factor is fixed to be zero, the smaller set becomes equal to the closed frequent itemsets.

5.3.1 Generalized Openness Propagation

The key concept in the g-closed itemset framework lies in that the openness propagation property (Section 5.2.3) holds even if the supports of itemsets are only *approximately* equal to those of their supersets. By approximate equality, we mean the following: The supports of itemsets X and Y are approximately equal (denoted as $support(X) \approx support(Y)$) iff they differ by at most ϵ, where ϵ is a user-specified "tolerance" factor. For brevity, we use the term ϵ-equality to denote approximate equality.

Definition 2 *The supports of itemsets X and Y are said to be approximately equal or ϵ-equal (denoted as $support(X) \approx support(Y)$) iff $|support(X) - support(Y)| \leq \epsilon$.*

We refer to the allowable error in itemset counts as *tolerance count*. The term "tolerance" is reserved for the allowable error in itemset supports and is equal to the tolerance count normalized by the database size. The generalized openness propagation property is stated and proved as a corollary to the following theorem:

Theorem 8 *If Y and Z are supersets of itemset X, then $support(Z) - support(Y \cup Z) \leq support(X) - support(Y)$.*

Proof: Since $Y \supseteq X$, it is clear that Y is present in every transaction that contains X except in at most $|t(X)| - |t(Y)|$ transactions. Further, since $Z \supseteq X$, X is present in every transaction that contains Z. Hence, it follows that Y is present in every transaction that contains Z except in at most $|t(X)| - |t(Y)|$ transactions. Therefore, $support(Z) - support(Y \cup Z) \leq support(X) - support(Y)$. □

Corollary 1 *If X and Y are itemsets such that $Y \supseteq X$ and $support(X) \approx support(Y)$, then for every itemset $Z : Z \supseteq X$, $support(Z) \approx support(Y \cup Z)$.*

Like Theorem 6, this result also suggests a general technique to incorporate into mining algorithms, which we refer to as *ϵ-equal support pruning*: If an itemset X has an immediate superset Y, with ϵ-equal support, then prune Y and avoid generating any candidates that are supersets of Y. The support of any of these pruned itemsets, say W, will be ϵ-equal to one of its subsets, $(W - Y) \cup X$.

By incorporating the above technique into mining algorithms, it is possible for us to produce a relaxed version of the generators discussed in Section 5.2.3. We could then apply the technique proposed in Section 5.2.5 to this relaxed version of the generators to produce a relaxed version of the closed itemsets.

5.3.2 Approximation Error Accumulation

The question remains as to whether the relaxed version of the closed itemsets generated above (in Section 5.3.1) serves our purpose – i.e., using these itemsets and their supports, is it possible to determine the supports of all frequent itemsets at least approximately? The answer, in general, is *no* due to the following reasons:

The generalized openness propagation property considers for any itemset X, only *one* superset Y with ϵ-equal support. If X has more than one superset (say $Y_1, Y_2, \ldots, Y_n$) with ϵ-equal support then a naive interpretation of the generalized openness propagation property would seem to indicate the following: Every itemset $Z : Z \supset X \bigwedge Z \not\supseteq Y_k, k = 1 \ldots n$, also has a proper superset $Y_1 \cup Y_2 \cup \cdots \cup Y_n \cup Z$ with ϵ-equal support. This would be true in the exact closed itemset case because then the support of $Y_1 \cup Y_2 \cup \cdots \cup Y_n \cup Z$ would be exactly equal to that of Z. In the general case, this is not necessarily true. However, we show in the following theorem that the difference between the supports of $Y_1 \cup Y_2 \cup \cdots \cup Y_n \cup Z$ and Z cannot be more than the sum of the differences between the supports of each $Y_k, k = 1 \ldots n$ and X.

Theorem 9 *If $Y_1, Y_2, \ldots, Y_n, Z$ are supersets of itemset X, then $support(Z) - support(\bigcup_{k=1}^{n} Y_k \cup Z) \leq \sum_{k=1}^{n}(support(X) - support(Y_k))$.*

Proof: By induction on n. When $n = 1$ this reduces to Theorem 8. Let the theorem be true for some n.

Since $Y_{n+1} \supseteq X$, it is clear that Y_{n+1} is present in every transaction that contains X except in at most $|t(X)| - |t(Y_{n+1})|$ transactions. Further, since $Z \supseteq X$, X is present in every transaction that contains Z. Hence, it follows that Y_{n+1} is present in every transaction that contains Z except in at most $|t(X)| - |t(Y_{n+1})|$ transactions.

Now, since the theorem is true for n according to the inductive step, $\bigcup_{k=1}^{n} Y_k$ is present in every transaction that contains Z except in at most $\sum_{k=1}^{n}(|t(X)| - |t(Y_k)|)$ transactions.

Combining the results in the above two paragraphs, it is clear that $\bigcup_{k=1}^{n+1} Y_k$ is present in every transaction that contains Z except in at most $\sum_{k=1}^{n+1}(|t(X)| - |t(Y_k)|)$ transactions. Hence, $support(Z) - support(\bigcup_{k=1}^{n+1} Y_k \cup Z) \leq \sum_{k=1}^{n+1}(support(X) - support(Y_k))$. □

In our approach we solve the problem of approximation error accumulation by ensuring that an itemset is pruned using the ϵ-equal support pruning technique only if the *maximum* possible cumulative error in approximation does not exceed the user-specified tolerance ϵ. Whenever an itemset X, having more than one immediate superset $Y_1, Y_2, \ldots, Y_n$, with ϵ-equal support is encountered, we prune each superset Y_k only as long as the sum of the differences between the supports of each pruned superset and X is within tolerance.

While performing the above procedure, at any stage, the sum of the differences between the support counts of each pruned superset and X is denoted by $X.debt$. Recall from Section 5.2.5 that these pruned supersets are included in $X.pruned$. Since $X.pruned$ needs to be propagated to all unpruned supersets of X, it becomes necessary to propagate $X.debt$ as well.

5.3.3 Problem Formulation

We now move on to providing a formal description of the g-closed itemset mining problem. This problem takes as input $\mathcal{I}$, a set of items sold by the store, $\mathcal{D}$, a database of customer purchase transactions, $minsup$, the minimum support threshold and ϵ, the tolerance factor. It produces as output the set of all frequent g-closed itemsets.

g-closed itemsets are those that result by incorporating the ϵ-equal support pruning technique into mining algorithms, while ensuring that the approximation error does not accumulate beyond the user-specified tolerance. Note that the set of g-closed itemsets

is not necessarily *unique*, but depends on the order in which pruning is performed. In practice, however, the order of the number of frequent g-closed itemsets remains same irrespective of the manner in which pruning is performed, and is usually much less than the number of frequent itemsets. Clearly, the set of g-closed itemsets satisfies the following properties:

1. If the tolerance factor is fixed to be zero, then it reduces to the set of "exact" closed itemsets. This is because – (1) approximation errors don't accumulate by definition, and (2) the generalized openness propagation property reduces to the normal openness propagation property.

2. The supports of frequent itemsets can be derived within a deterministic user-specified tolerance factor. This is because – (1) approximation error accumulation is checked by avoiding pruning of certain critical itemsets, and (2) generalized openness propagation property ensures approximation error for supersets of generators to be within the tolerance factor.

3. The identities of all frequent itemsets can be derived from the frequent g-closed itemsets. This is possible if while performing equal support pruning, we ensure that the support of an itemset is consistently approximated by the support of its subset. Hence we would always over-estimate the supports of itemsets. Although this would result in some borderline infrequent itemsets (whose support exceeds $minsup - \epsilon$) being declared as frequent, it would ensure that there are no false negatives.

Similar to the exact closed itemset case, we have for every itemset X, a g-closed itemset $Y : Y \supseteq X$ whose support is ϵ-equal to that of X. We refer to the g-closed itemset corresponding to an itemset X as its *g-closure* and denote it as $g(X)$.

Note that in the above problem formulation, we have applied the g-closed itemset framework only w.r.t. the frequent itemsets and *not* to their negative border. The reason for this is that any itemset in the negative border cannot have a subset or superset that is also in the negative border. Since the g-closed itemset framework depends on the presence

of itemsets that have supersets with ϵ-equal support, it cannot be applied on the negative border.

5.4 Rule Generation

In this section, we move on to describe the process of association rule generation given the frequent g-closed itemsets and their associated supports. Recall that an association rule is of the form $X_1 \longrightarrow X_2$, where $X_1, X_2 \subseteq \mathcal{I}$. Its support equals $|t(X_1 \cup X_2)|$, and its confidence equals $|t(X_1 \cup X_2)|/|t(X_1)|$. We are interested in finding all rules whose support and confidence are at least $minsup$ and $minconf$, respectively.

Since the support of an itemset X is ϵ-equal to the support of its g-closure, the rule $X_1 \longrightarrow X_2$ can be approximated by $g(X_1) \longrightarrow g(X_2)$. Let sup and $conf$ be the support and confidence of the original rule, respectively. From the generalized rule, the support of the original rule can be estimated to be within $(sup - \epsilon, sup)$, by the definition of approximate equality of supports and the nature of the g-closed itemset framework.

The following theorem shows that the confidence of a rule (whose actual confidence is $conf$) can be estimated to be within $(conf \times \lambda, conf/\lambda)$ where λ is given by $(1-\epsilon/minsup)$. It is clear that the approximation error becomes acceptable if the tolerance ϵ is much less than the minimum support. The tolerance ϵ could be chosen by the user after allowing for the approximation error in rule confidences.

Theorem 10 *Given the g-closed itemsets and their associated supports, the confidence $conf$, of a rule $X_1 \longrightarrow X_2$ can be estimated to be within $(conf \times \lambda, conf/\lambda)$ where λ is given by $(1 - \epsilon/minsup)$.*

Proof: We know $conf = support(X_1 \cup X_2)/support(X_1)$. Estimated confidence is $support(g(X_1 \cup X_2))/support(g(X_1))$. Since the support of an itemset X can be estimated within $(support(X) - \epsilon, support(X))$, the extremities in the estimated confidence are $((support(X_1 \cup X_2) - \epsilon)/support(X_1)$ and $support(X_1 \cup X_2)/(support(X_1) - \epsilon))$. It can be shown by algebraic manipulation that since $X_1 \cup X_2$ and X_1 have supports at least equal to $minsup$, the above range is subsumed by $(conf \times \lambda, conf/\lambda)$. □

From the above discussion, it is clear that it is sufficient to consider rules *only* among the frequent g-closed itemsets. Further, it has been shown in [Zak00, TPBL00] that it is sufficient to consider rules among adjacent frequent itemsets in the itemset lattice since other rules can be inferred by transitivity. This result carries over even to the frequent g-closed itemsets. Techniques to prune rules that have the same predictive power as other rules with fewer items have also been presented in [Zak00, TPBL00]. These techniques are complementary to the techniques proposed here and could be incorporated in them to further reduce the size of rule covers.

5.5 Incorporation in Levelwise Algorithms

In the previous section we presented the g-closed itemset framework and the theory supporting it. In this section we show that the framework can be integrated into levelwise algorithms. We chose the classical Apriori algorithm as a representative of the levelwise mining algorithms. The same techniques can be used to integrate the framework into other levelwise algorithms such as VIPER and FP-growth, yielding corresponding improvements in their output sizes and response times. Integration of our scheme into Apriori yields g-Apriori, an algorithm for mining frequent g-closed itemsets. After describing the design of the new algorithm in Section 5.5.1, we explain the details of its operation in Section 5.5.2. Finally, in Section 5.5.3, we show that g-Apriori indeed generates the frequent g-closed itemsets.

5.5.1 The Design of g-Apriori

The g-Apriori algorithm is obtained by combining the ϵ-equal support pruning technique described in Section 5.2.4 with the subset-based pruning of Apriori. This makes it similar to the A-Close algorithm [PBTL99] for mining frequent closed itemsets. However, g-Apriori significantly differs from A-Close (even for the zero tolerance case) in that it does not require an additional database scan to mine closed itemsets. This is achieved by utilizing the technique described in Section 5.2.5.

As discussed in Section 5.3.2, we incorporate techniques to check the accumulation of approximation error in itemset supports. This is achieved in g-Apriori by maintaining an extra field, *debt*, with each generator. This field stores the approximation error that accumulates for each generator. Whenever the value in this field might exceed tolerance, g-Apriori avoids pruning of the corresponding generator, thereby ensuring that the approximation error never exceeds tolerance.

5.5.2 The Mechanics of g-Apriori

The pseudo-code of the g-Apriori algorithm is shown in Figure 5.1 and works as follows: The code between lines 1–9 of the algorithm, excluding lines 6 and 7, consists of the classical Apriori algorithm. The SupportCount function (line 4) takes a set of itemsets as input and determines their counts over the database by making one scan over it.

Every itemset X in C_k (the set of candidate k-itemsets), G_k (frequent k-generators) and G (the frequent generators produced so far) has an associated counter, $X.count$, to store its support count during algorithm execution. Every itemset X in G has two fields in addition to its counter: (1) $X.pruned$ (described in Section 5.2.5). (2) $X.debt$: an integer value that is used to check the accumulation of approximation error in itemset supports.

The Prune function is applied on G_k (line 6) before the $(k+1)$-candidates C_{k+1} are generated from it using AprioriGen (line 8). Its responsibility is to perform ϵ-equal support pruning while ensuring that approximation error in the supports of itemsets is not accumulated. The pseudo-code for this function is shown in Figure 5.2 and it performs the following task: it removes any itemset X from G_k if X has a subset Y with ϵ-equal count, provided $Y.debt$ remains within tolerance.

The code in lines 1–9, excluding line 7, is analogous to the A-Close algorithm [PBTL99] for generating frequent closed itemsets. At the beginning of line 10, G would contain the equivalent of the "generators" of the A-Close algorithm.

The PropagatePruned function is applied on G_k (line 7) and it ensures that the *pruned* value of each itemset X in G_k is appended with the *pruned* values of each immediate subset of X. The pseudo-code for this function is shown in Figure 5.3. The necessity for

performing this function was explained in Section 5.2.5, where we showed that the *pruned* value of an itemset should be propagated to all its supersets.

Finally, in lines 10–11, the g-closed itemsets are output.

5.5.3 Proof of Correctness

At zero tolerance, g-Apriori reduces to the A-Close algorithm – however, the final extra database scan in A-Close is avoided using the technique described in Section 5.2.5. As proved earlier, the new technique ensures that all closed frequent itemsets are enumerated. Hence g-Apriori is correct at zero tolerance threshold.

As discussed in Section 5.3.3, g-closed itemsets are those that result by incorporating the ϵ-equal support pruning technique into mining algorithms, while ensuring that the approximation error does not accumulate beyond the user-specified tolerance. Since g-Apriori incorporates ϵ-equal support pruning into Apriori, it suffices to prove that the approximation error in itemset supports does not exceed the tolerance factor ϵ.

As discussed in Section 5.3.2, g-Apriori keeps track of the approximation error that accumulates for each generator, X, in a field named $X.debt$. g-Apriori then avoids pruning of any generator when *debt* might exceed tolerance. This ensures that the approximation error is always within the tolerance factor. Hence proved.

5.6 Incorporation in Two Pass Algorithms

In this section we show that the g-closed framework can be incorporated into two-pass mining algorithms. As mentioned in the Introduction, this is a novel and important contribution because two pass algorithms are typically much faster than level-wise algorithms and also because they can be tweaked to work on data streams [MM02]. We selected ARMOR (described in Chapter 4) as a representative of the class of two-pass mining algorithms. Integration of the g-closed framework into ARMOR yields g-ARMOR, a two-pass algorithm for mining frequent g-closed itemsets. In order to describe our strategy, we first review the overall structure of ARMOR, focussing on those features that are

necessary for the subsequent description.

5.6.1 The ARMOR Algorithm

In the ARMOR algorithm, the database is conceptually partitioned into disjoint blocks. Data is read from disk and processed partition by partition. At most two passes are made over the database.

In the first pass, the algorithm starts with the set of all 1-itemsets as candidates (i.e. potentially frequent itemsets). After processing each partition, the set of candidates (denoted as $\mathcal{G}$) is updated – new candidates may be inserted and existing ones removed. The algorithm ensures that at any stage, if d is the database scanned so far, then the frequent itemsets within d (also called d-frequent itemsets) are available. The algorithm also maintains the *partial counts* of these itemsets – the partial count of an itemset is its count since it has been inserted into $\mathcal{G}$.

In the second pass, complete counts of the candidates obtained at the end of the first pass are determined. During this pass, there are no new insertions into $\mathcal{G}$. However, candidates that can no longer become frequent are removed at each stage.

5.6.2 Details of Incorporation

The rule that we follow in incorporating the g-closed framework into ARMOR is simple: While processing a partition during the first pass, if we find the partial count of an itemset X to be ϵ-equal to that of its superset Y, then prune every proper superset of Y from $\mathcal{G}$ while ensuring that the approximation error does not accumulate beyond the tolerance limit. That is, whenever an itemset X, that has more than one immediate superset $Y_1, Y_2, \ldots, Y_n$, with ϵ-equal partial count is encountered, we prune each superset Y_k only as long as the sum of the differences between the partial counts of each pruned superset and X is within tolerance.

We now show that incorporating this rule in ARMOR yields valid results w.r.t the g-closed framework. This is first shown for a restricted case after which the restrictions are removed one by one. Consider the following three cases with increasing degree of

complexity:

Case 1: During the first pass, itemset X and its superset Y are inserted into $\mathcal{G}$ after processing the *first* partition itself and are never removed later. They *consistently* have ϵ-equal partial counts (w.r.t tolerance ϵ) in every partition.

Case 2: Identical to case 1, except that itemsets X and Y may have been inserted into $\mathcal{G}$ not necessarily after processing the first partition itself, but perhaps after processing some *later* partition. It is also possible for X and Y to be removed from $\mathcal{G}$ and reinserted later. They *consistently* have ϵ-equal partial counts from the point they were last inserted into $\mathcal{G}$ till the end of the pass.

Case 3: Identical to case 2, except that itemsets X and Y may *not consistently* have ϵ-equal partial counts.

Note that in all the above cases, we consider itemsets X and Y to have ϵ-equal partial counts only if they were both last inserted into $\mathcal{G}$ *together*. Otherwise, it is infeasible to compare their partial counts as these counts would be over different portions of the database. This restriction is not likely to have much impact for small tolerances because then X and Y are likely to have ϵ-equal counts in every partition. It is therefore likely that they would always be inserted into $\mathcal{G}$ together. Even if they are not, it would only affect the performance and not the correctness of the algorithm.

Case 1

In Case 1, the partial counts of X and Y at the end of the first pass would be equal to their complete counts. In this case, the rule reduces to ϵ-equal support pruning while ensuring that the approximation error doesn't exceed ϵ.

Case 2

In Case 2, we know that the partial counts of X and Y are ϵ-equal *after* they were last inserted into $\mathcal{G}$. If their partial counts are also ϵ-equal *before* they are last inserted into

$\mathcal{G}$, then this case would reduce to Case 1. Let us consider the harder scenario when they are not ϵ-equal. This would be discovered while processing some partition P_i during the second pass of ARMOR, when the complete counts of X and Y are being obtained. Recall that we would have pruned all supersets of Y during the first pass because the partial counts of X and Y were ϵ-equal. All those supersets of Y now need to be regenerated with appropriate partial counts.

The modification to ARMOR that is required to regenerate these supersets is as follows: For every d-frequent itemset Z in $\mathcal{G}$ such that $Z \supset X \bigwedge Z \not\supseteq Y$, insert a new candidate $Y \cup Z$ into $\mathcal{G}$. The partial count of the new candidate should be set equal to the partial count of Z. Note that in ARMOR, the partial counts of each Z and the corresponding new candidates would not yet have been updated over the current partition P_i. It is clear that the partial counts of the new candidates at this stage, i.e. for the partition preceding P_i, are accurate w.r.t tolerance ϵ . Since their partial counts would be updated individually over partition P_i and all later partitions, their partial counts would be ϵ-equal to their complete counts by the end of the second pass.

We have shown above that the complete counts of candidates obtained at the end of the second pass of ARMOR are accurate w.r.t tolerance ϵ. We now proceed to show that there cannot be any frequent superset of Y that was not regenerated during the second pass. It is clear that any frequent superset of Y, say W, would have a subset $(W - Y) \cup X$ that is also frequent. This subset would be available in $\mathcal{G}$ during the second pass. Hence W would have been regenerated in ARMOR by forming the new candidate: $Y \cup [(W - Y) \cup X]$ as outlined in the previous paragraph.

Case 3

Moving over to Case 3, let us consider the following scenario: Itemsets X and Y are in $\mathcal{G}$ during the first pass at the end of some partition P_i and have ϵ-equal partial counts. Then, after processing the partition immediately after P_i, say P_j, the partial counts of X and Y are no longer ϵ-equal. Since the supersets of Y would have been removed from $\mathcal{G}$ earlier because the partial counts of X and Y were ϵ-equal, they would now have to be

regenerated. The modification to ARMOR that is required to regenerate these supersets is identical to that outlined for Case 2 above. Following the same reasoning as in Case 2, it is clear that the partial counts of the regenerated supersets would be accurate w.r.t tolerance ϵ and also that there cannot be any d-frequent superset of Y that is not regenerated.

5.7 Performance Study

In the previous sections, we have described the g-closed itemset framework along with the g-Apriori and g-ARMOR algorithms. We have conducted a detailed study to assess the utility of the framework in reducing both the output size and the response time of mining operations.

Our experiments cover a range of databases and mining workloads including the real datasets from the UC Irvine Machine Learning Database Repository and synthetic datasets from the IBM Almaden generator. These datasets are the same as those used in [ZH02]. Our experiments also include the real dataset, BMS-WebView-1 [ZKM01] from Blue Martini Software. This dataset originated from a dot-com company called Gazelle.com, a legwear and legcare retailer and contains several months of clickstream data. Table 5.2 shows the characteristics of the datasets used in our evaluation.

Database	#Items	Record Length	#Records
BMS-WebView-1	497	2.5	59,602
chess	76	37	3,196
connect	130	43	67,557
mushroom	120	23	8,124
pumsb*	7117	50	49,046
T10I4D100K	1000	10	100K
T10I4D10M	1000	10	10M

Table 5.2: Database Characteristics

We conducted four sets of experiments: In Experiment 1, we measure the output size reduction of the g-closed itemsets w.r.t frequent itemsets. In Experiment 2, we measure the response time reduction of the g-Apriori algorithm w.r.t Apriori. In Experiment 3,

we measure the response times of g-ARMOR and compare them with those of Apriori. Finally, in Experiment 4, we studied how the performance of the implemented algorithms scale with database size. All the algorithms were coded in C++ and the experiments were conducted on a 700-MHz Pentium III workstation running Red Hat Linux 6.2, configured with 512 MB main memory and a local 18 GB SCSI 10000 rpm disk.

The same data-structures (hashtrees [AS94]) and the same optimizations (using arrays to store itemset counters in the first two database passes) were used in both g-Apriori and Apriori to ensure that the experimental results are a good indication of the utility of the g-closed itemset framework, and not of any differences in the structure of the two algorithms.

We chose tolerance count values ranging from zero (corresponding to the exact closed itemset case) to 1000. While higher values of tolerance are uninteresting in themselves, their inclusion is useful in studying the effect of increasing tolerance on the output size.

5.7.1 Output Size Reduction

We now report on our experimental results. In our first experiment, we measure the output size reduction as a percentage of the frequent itemsets that are pruned to result in g-closed itemsets. The results of these experiments are shown in Figures 5.4a–f for the various databases. The x-axis in these graphs represents the tolerance count values, while the y-axis represents the percentage of frequent itemsets pruned. Each graph contains two curves corresponding to two different minimum support thresholds. We show only two curves per graph to avoid visual clutter.

In these graphs, we first see that the pruning achieved due to the g-closed itemset framework, in most cases, is significant. For example, on the chess dataset (Figure 5.4b) for a minimum support threshold of 80%, the percentage of pruned itemsets is only 38% at zero tolerance (closed itemset case). For the same example, at a tolerance count of 50 (corresponding to a maximum error of 1.5% in itemset supports), *the percentage of pruned itemsets increases to 97%!* The exact pruning achieved for each database for selected tolerance values is shown in Table 5.3, along with the number of frequent itemsets

and frequent closed itemsets.

The pruning achieved is significant even on the sparse datasets generated by the IBM Almaden generator. For example, on the T10I4D100K dataset (Figure 5.4f) for a minimum support threshold of 0.1%, the percentage of pruned itemsets is only 2.6% at zero tolerance, whereas it increases to 41.5% at a tolerance count of 10 (corresponding to a maximum error of 0.01% in itemset supports).

Database	Support	#Freq	#Closed	#g-Closed	Tolerance
BMS-WebView-1	0.08%	10286	9427	8138	0.008%
BMS-WebView-1	0.06%	461521	75653	55198	0.008%
chess	80%	8227	5083	398	1.5%
chess	70%	48969	23991	2107	1.5%
connect	97%	487	284	46	0.7%
connect	90%	27127	3486	793	0.7%
mushroom	40%	565	140	113	1.2%
mushroom	20%	53583	1197	1059	1.2%
pumsb*	60%	167	68	63	0.2%
pumsb*	40%	27354	2610	1309	0.2%
T10I4D100K	0.5%	1073	1073	979	0.01%
T10I4D100K	0.1%	27532	26806	16104	0.01%

Table 5.3: Output Size

Next, we notice that in some cases such as for the mushroom dataset (Figure 5.4d), the percentage of pruned itemsets increases only marginally with an increase in tolerance since even at zero tolerance itself (corresponding to the closed itemset case), the pruning is high. We hasten to add that at zero tolerance, the pruning depends critically on the dataset contents. We demonstrate this by adding a select 438 transactions to the 8,124 transaction mushroom dataset. The selection was made so as to break exact equalities. This caused the number of closed frequent itemsets at a minimum support threshold of 20% to increase from 1,390 to 15,541 – a factor of *11 times*! The number of frequent g-closed itemsets (at a tolerance count of 5) only increased from 1,386 to 2243.

An interesting trend that we noticed in all cases is that the percentage of pruned itemsets increases dramatically at low tolerances and then plateaus as the tolerance is increased further. This trend is significant as it indicates that the maximum benefit

attainable using the g-closed itemset framework is obtained at low tolerances. The reason for this trend is as follows:

As the length of an itemset, X, increases, the number of subsets that it has increases exponentially. This means that the chance of one of the subsets being ϵ-equal to one of its subsets becomes exponentially high. Hence most of the long generators get pruned at low tolerances itself. This accounts for the initial steep rise in the curve. Next, for very short generators (e.g., 1-itemsets and 2-itemsets), the difference in their supports from those of their supersets is typically large [ZG01]. Hence, most of these generators are not pruned even at high tolerances. Finally, those generators that are in the middle range (e.g. 3-itemsets) will get pruned at a slow pace with regard to the increase in tolerance. This accounts for the gradual upward slope in the curve after the initial exponential increase.

Another interesting trend that we notice in all cases is that the percentage of pruned itemsets is more for lower minimum support thresholds. The reason for this behaviour is that when the minimum support threshold is reduced, it is possible for longer itemsets to become frequent. As discussed above, longer itemsets are more likely to be pruned, thereby leading to more efficient pruning at lower minimum support thresholds. This trend is significant as it counteracts the exponential increase in output size of frequent itemset mining algorithms with decreasing minimum support thresholds.

A final point: in Figure 5.4c, we see that for the connect database, the percentage of pruned itemsets decreases marginally between tolerance counts of 500 and 1000. This unintuitive behaviour arises due to the following reason: In order to ensure that approximation error in itemset supports does not accumulate, not all itemsets having supersets with ϵ-equal supports are pruned. It is therefore possible that an itemset that was pruned at a tolerance count of 500 may not be pruned at a tolerance count of 1000. If such an itemset is short, then it may have many supersets that are not pruned as well. This phenomenon could at times cause the number of pruned itemsets to decrease marginally with an increase in tolerance.

5.7.2 Response Time Reduction

In our second experiment, we measure the performance gain obtained from the g-closed itemset framework. This is measured as the percentage reduction in response time of g-Apriori over Apriori. The results of these experiments are shown in Figures 5.5a–f. The x-axis in these graphs represents the tolerance count values, while the y-axis represents the performance gain of g-Apriori over Apriori. As in Experiment 1, each graph contains two curves corresponding to two different minimum support thresholds.

In all these graphs, we see that the performance gain of g-Apriori over Apriori is significant. In fact, the curves follow the same trend as in Experiment 1. This is expected because the bottleneck in Apriori (and other frequent itemsets mining algorithms) lies in the counting of the supports of candidates. Hence any improvement in pruning would result in a corresponding reduction in response-time.

5.7.3 Response Times of g-ARMOR

In our third experiment, we measure the response times of g-ARMOR and compare them against those of Apriori. The results of these experiments are shown in Figures 5.6a–f. The x-axis in these graphs represents the tolerance count values, while the y-axis (plotted on a **log-scale**) represents the response times of g-ARMOR and Apriori in seconds. Each graph contains two curves for each algorithm corresponding to two different minimum supports. The curves corresponding to Apriori are shown using a dashed line style.

In all these graphs, we see that the response times of g-ARMOR are over an *order of magnitude* faster than Apriori. We also notice that the response times become faster with an increase in tolerance count values. As in Experiment 2, this is expected because more candidates are pruned at higher tolerances. The reduction in response time is not as steep as in Experiment 2 due to the fact that g-ARMOR is much more efficient than g-Apriori and hence less responsive to a change in the number of candidates.

We do not show the response times of ARMOR in these graphs since it ran out of main memory for most of the datasets and support specifications used in our evaluation. This was because most of these datasets were dense, whereas ARMOR, as described in

Chapter 4, is designed only for sparse datasets and is memory intensive.

5.7.4 Scale-up Experiment

In our fourth (and final) experiment, we studied how the performance of the implemented algorithms scale with database size. This experiment was conducted on the T10I4D10M database that has 10 million records. The results of this experiment are shown in Figure 5.7. The x-axis in this graph represents the tolerance count values, while the y-axis (plotted on a **log-scale**) represents the response times of g-ARMOR, g-Apriori and Apriori in seconds. The graph contains two curves for each algorithm corresponding to two different minimum supports. The curves corresponding to Apriori are shown using a dashed line style.

In this graph, we notice that while the absolute times of the algorithms vary from the previous experiment, the shape of the curves remain the same. The performances of all three algorithms – g-ARMOR, g-Apriori and Apriori are seen to be linear w.r.t. database size. This behaviour of these algorithms is explained as follows: (1) The number of database passes for each of these algorithms depends only on the density of patterns in the database and not on the number of transactions. (2) The rate at which transactions are processed in each pass depends only on the distribution from which the transactions are derived, the number of candidate itemsets being counted and on the efficiency of the data-structure that holds the counters of candidates. It also does not depend on the number of transactions in the database. Due to these two reasons, it is expected that the response-time performances of the algorithms under study are linear w.r.t. database size.

5.8 Conclusions

In this chapter we proposed the generalized closed itemset framework (or g-closed itemset framework) in order to manage the information overload produced as the output of frequent itemset mining algorithms. This framework builds upon the original closed itemset concept over which it provides an order of magnitude improvement. This is achieved by

relaxing the requirement for exact equality between the supports of itemsets and their supersets. Instead, our framework accepts that the supports of two itemsets are equal if the difference between their supports is within a user-specified tolerance factor.

We also presented two algorithms – g-Apriori (based on the classical levelwise Apriori algorithm) and g-ARMOR (based on ARMOR, presented in Chapter 4) for mining the frequent g-closed itemsets. g-Apriori utilizes a new method for generating frequent g-closed itemsets from their generators. This new method avoids the costly additional pass that was required in the A-Close algorithm for mining frequent closed itemsets. g-Apriori is shown to perform significantly better than Apriori solely because the frequent g-closed itemsets are much fewer than the frequent itemsets. Finally, g-ARMOR was shown to perform over an order of magnitude better than Apriori over all databases and support specifications used in our experimental evaluation.

```
g-Apriori (D, I, minsup, tol)
Input: Database D, Set of Items I, Minimum Support minsup, Tolerance Count tol
Output: Generalized Closed Itemsets
1.      C_1 = set of all 1-itemsets;
2.      G = φ;
3.      for (k = 1; |C_k| > 0; k++)
4.          SupportCount(C_k, D); // Count supports of C_k over D
5.          G_k = Frequent itemsets in C_k
6.          Prune(G_k, G, tol);
7.          PropagatePruned(G_k, G, tol);
8.          C_{k+1} = AprioriGen(G_k);
9.          G = G ∪ G_k;
10.     for each itemset X in G
11.         Output (X ∪ X.pruned, X.count);
```

Figure 5.1: The g-Apriori Algorithm

```
Prune (G_k, G, tol)
Input: Frequent k-itemsets G_k, Generators G, Tolerance Count tol
Output: Remove non-generators from G_k
1.      for each itemset X in G_k
2.          for each (|X| − 1)-subset Y of X, in G
3.              debt = Y.count − X.count;
4.              if (debt + Y.debt ≤ tol)
5.                  G_k = G_k − {X}
6.                  Y.pruned = Y.pruned ∪ (X − Y)
7.                  Y.debt += debt
```

Figure 5.2: Pruning Non-generators from G_k

```
PropagatePruned (G_k, G, tol)
Input: Frequent k-itemsets G_k, Generators G, Tolerance Count tol
Output: Propagate pruned value to generators in G_k
1.      for each itemset X in G_k
2.          for each (|X| − 1)-subset Y of X, in G
3.              if (X.debt + Y.debt ≤ tol)
4.                  X.pruned = X.pruned ∪ Y.pruned
5.                  X.debt+ = Y.debt
```

Figure 5.3: Propagate Pruned Value to Supersets

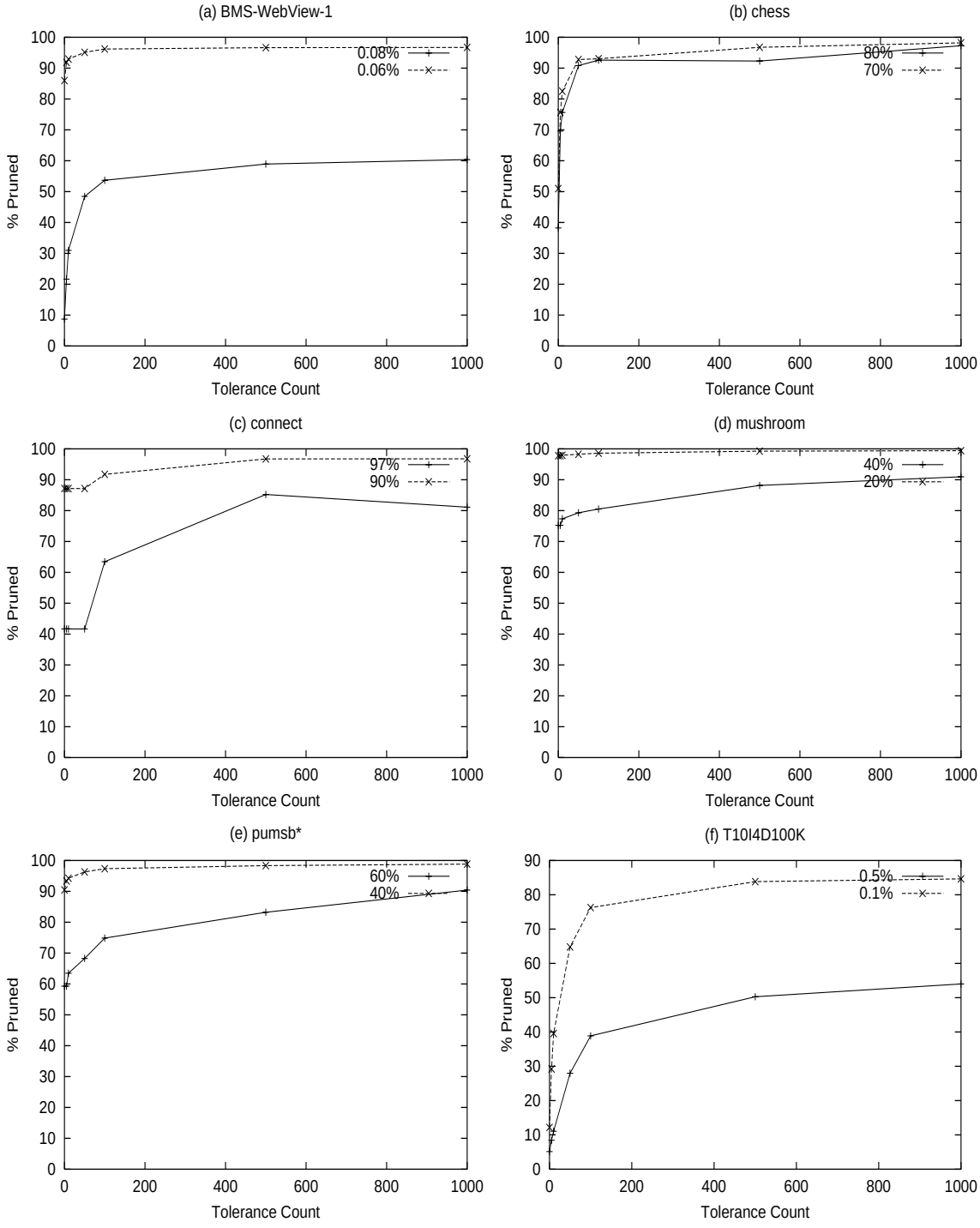

Figure 5.4: Output Size Reduction

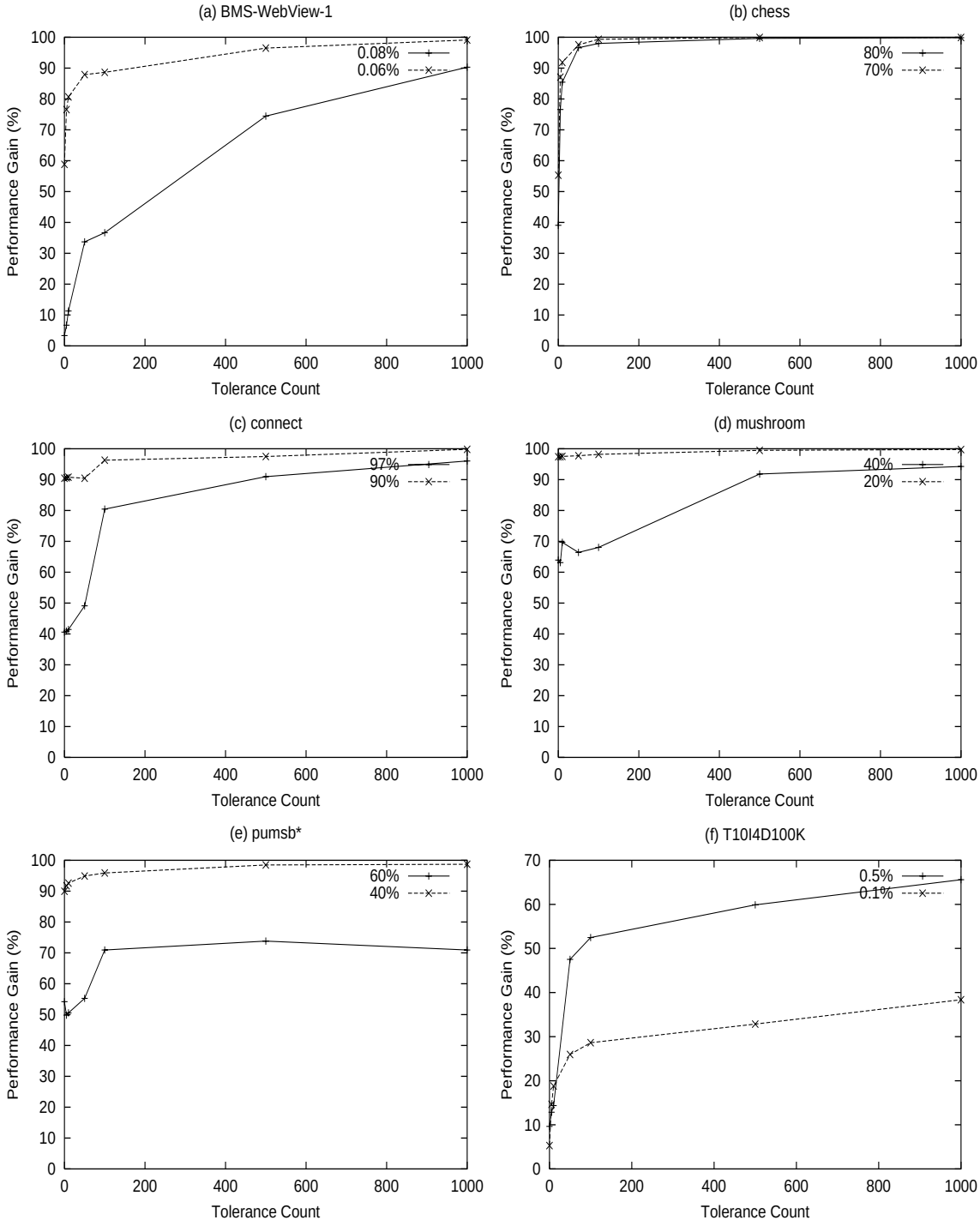

Figure 5.5: Response Time Reduction

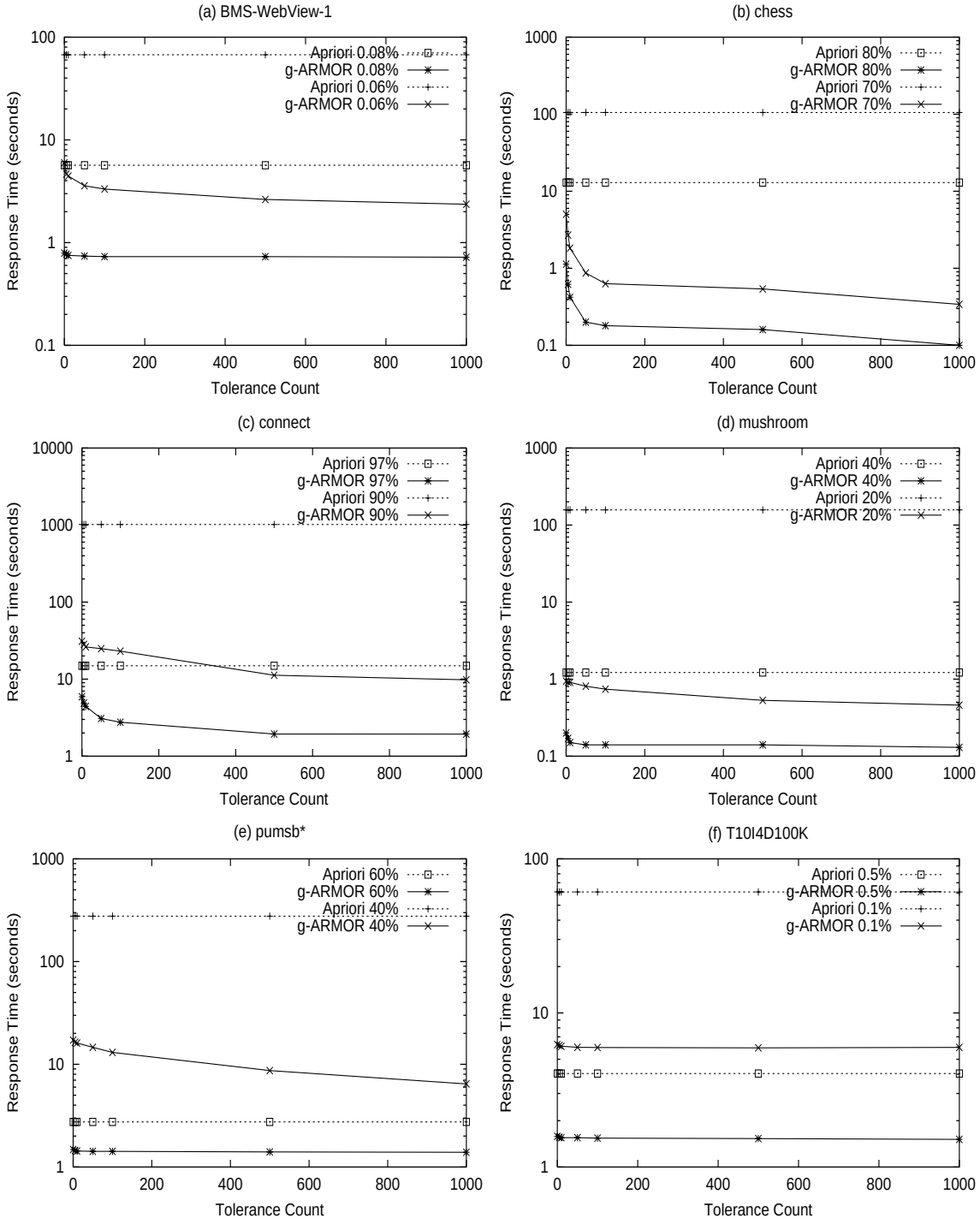

Figure 5.6: Response Times of g-ARMOR

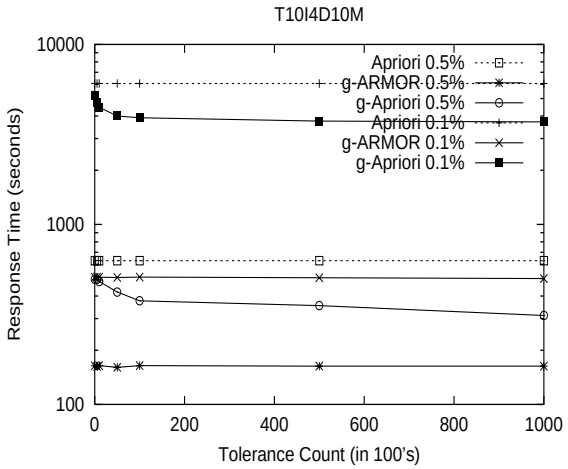

Figure 5.7: Scale-up Experiment

Chapter 6

Incremental Mining

6.1 Introduction

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 and $DB \cup db$. Mining db is necessary to evaluate the effects of business strategies; whereas mining $DB \cup db$ is necessary to maintain the updated set of mining rules. Such "incremental" mining is the focus of this chapter.

As mentioned in Chapter 1, our work on incremental mining presented in this thesis was actually done prior to our work on the other two issues, namely, on the efficiency

of BAR-mining algorithms and on the conciseness of results. However, for pedagogical reasons, we present it in the end.

6.1.1 The State-of-the-Art

The design of incremental mining algorithms for association rules has been considered earlier in [AFLM99, CHNW96, CLK97, CVB96, F+97, T+97]. While these studies were a welcome first step in addressing the problem of incremental mining, they also suffer from a variety of limitations that make their design and evaluation unsatisfactory from an "industrial-strength" perspective:

Effect of Skew: The effect of temporal changes (i.e. skew) in the distribution of database values between DB and db has not been considered. However, in practical databases, we should typically expect to see skew for the following reasons: (a) inherent seasonal fluctuations in the business process, and/or (b) effects of business strategies that have been put into place since the last mining. So, we expect that skew would be the norm, rather than the exception.

As we will show later in this chapter, the performance of the algorithms presented in [F+97, T+97] is sensitive to the skew 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 completely ignoring the previous mining results and applying Apriori from scratch on the entire current database.

Size of Database: The evaluations of the algorithms has been largely conducted on databases and increments that are small relative to the available main memory. For example, the standard experiment considered a database with 0.1 M tuples, with each tuple occupying approximately 50 bytes, resulting in a total database size of only 5 MB. For current machine configurations, this database would completely fit into memory with plenty still left to spare. Therefore, the ability of the algorithms to *scale* to the enormous disk-resident databases that are maintained by most business organizations, has not been clearly established.

Characterizing Efficiency: Apart from comparing their performance with that of Apriori, no quantitative assessment has been made of the *efficiency* of these algorithms in terms of their distance from the optimal, which would be indicative of the scope, if any, for further improvement in the design of incremental algorithms.

Incomplete Results: Almost all the algorithms fail to provide the mining results for solely the increment, db. As mentioned before, these results are necessary to help evaluate the effects of business strategies that have been put into place since the previous mining.

Changing User Requirements: It is implicitly assumed 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, given mining's exploratory nature, we could expect user requirements to change with time, perhaps resulting in different minimum support levels across mining operations. Extending the algorithms to efficiently handle such "multi-support" environments is not straightforward.

6.1.2 Contributions

In this chapter, we present and evaluate an incremental mining algorithm called **DELTA** (Differential Evaluation of Large iTemset Algorithm). The core of DELTA is similar to the previous algorithms but it also incorporates important design alterations for addressing their above-mentioned limitations. With these extensions, DELTA represents a practical algorithm that can be effectively utilized for real-world databases. The main features of the design and evaluation of DELTA are the following:

- DELTA guarantees that, for the entire mining process, at most *three passes* over the increment and *one pass* over the previous database may be necessary. We expect that such bounds will be useful to businesses for the proper scheduling of their mining operations.

- 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.

- For computing the negative border [Toi96] closure, a major performance-determining factor in the incremental mining process, a new hybrid scheme that combines the features of earlier approaches is implemented.

- DELTA provides complete mining results for both the entire current database as well as solely the increment.

- DELTA can handle multi-support environments, requiring only *one* additional pass over the current database to achieve this functionality.

- For the special case when the previous support threshold is so high that there are no frequent itemsets over DB, DELTA reduces to a re-mining of the entire database.

- By carefully integrating optimizations previously proposed for first-time hierarchical mining algorithms, the DELTA design has been extended to efficiently handle incremental mining of hierarchical association rules. As mentioned in Chapter 1, this illustrates the point that extensions to the basic association rule model including hierarchical, categorical and quantitative rules are finally reducible to BAR-mining.

- The performance of DELTA is evaluated on a variety of dynamic databases and compared with that of Apriori and the previously proposed incremental mining algorithms for boolean association rules. For hierarchical association rules, we compare DELTA against the Cumulate first-time mining algorithm presented in [SA95]. All experiments are made on databases that are significantly larger than the entire main memory of the machine on which the experiments were conducted. The effects of database skew are also modeled.

  The results of our experiments show that DELTA can provide significant improvements in execution times over the previous algorithms in all these environments.

Further, DELTA's performance is comparatively robust with respect to database skew.

- We also include in our evaluation suite the performance of an *an Oracle* that has complete apriori knowledge of the identities of all the frequent itemsets (and their associated negative border) both in the current database as well as in the increment and only requires to find their respective counts. Like in Chapter 4, modeling the Oracle's performance permits us to characterize the efficiency of practical algorithms in terms of their distance from the optimal.

  Our experiments show that DELTA's efficiency is *close to that obtained by the Oracle* for many of the workloads 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.

6.1.3 Organization

The remainder of this chapter is organized as follows: The DELTA algorithm for both boolean and hierarchical association rules is presented in Section 6.2 for the equi-support environment. The algorithm is extended to handle the multi-support case in Section 6.3. The performance model is described in Section 6.5 and the results of the experiments are highlighted in Section 6.6. Finally, in Section 6.7, we present the conclusions of our study and outline future research avenues.

6.2 The DELTA Algorithm

In this section, we present the design of the DELTA algorithm. For ease of exposition, we first consider the "equi-support" case, and then in Section 6.3, we describe the extensions required to handle the "multi-support" environment. In the following discussion and in the remainder of this chapter, we use the notation given in Table 2.1 of Chapter 2. The relevant part of this table has been reproduced in Table 6.1 for convenience. Also, we

DB, db, $DB \cup db$	Previous, increment, and current database
$minsup_{DB}$	Previous Minimum Support Threshold
$minsup_{DB \cup db}$	New Minimum Support Threshold
$minsup$	Minimum Support Threshold when $minsup_{DB} = minsup_{DB \cup db}$
F_{DB}, F_{db}, $F_{DB \cup db}$	Set of frequent itemsets in DB, db and $DB \cup db$
N_{DB}, N_{db}, $N_{DB \cup db}$	Negative borders of F_{DB}, F_{db} and $F_{DB \cup db}$
F_{known}	Set of known-frequent itemsets during algorithm execution: $F_{DB \cup db} \cap (F_{DB} \cup N_{DB})$
N_{known}	Negative border of F_{known}
$Infrequent$	Set of known-infrequent itemsets during algorithm execution
$Infrequent_{db}$	Set of known-infrequent (within db) itemsets during algorithm execution

Table 6.1: Notation (from Table 2.1)

use the terms "frequent", "infrequent", "count" and "support" with respect to the entire database $DB \cup db$, unless otherwise mentioned.

The input to the incremental mining process consists of the set of previous frequent itemsets F_{DB}, its negative border N_{DB}, and their associated supports. The output is the updated versions of the inputs, namely, $F_{DB \cup db}$ and $N_{DB \cup db}$ along with their supports. In addition, the mining results for solely the increment, namely, $F_{db} \cup N_{db}$, are also output.

6.2.1 The Mechanics of DELTA

The pseudo-code of the core DELTA algorithm for generating boolean association rules is shown in Figure 6.1 – the extension to hierarchical association rules is presented in Section 6.2.2. At most *three* passes over the increment and *one* pass over the previous database are made, and we explain below the steps taken in each of these passes. After this explanation of the mechanics of the algorithm, we discuss in Section 6.2.3 the rationale behind the design choices.

First Pass over the Increment

In the first pass, the counts of itemsets in F_{DB} and N_{DB} are updated over the increment db, using the function UpdateCounts (line 1 in Figure 6.1). By this, some itemsets in

DELTA $(DB, db, F_{DB}, N_{DB}, minsup)$
Input: Previous Database DB, Increment db, Previous Frequent Itemsets F_{DB}, Previous Negative Border N_{DB}, Minimum Support Threshold $minsup$
Output: Updated Set of Frequent Itemsets $F_{DB \cup db}$, Updated Negative Border $N_{DB \cup db}$
begin
1. UpdateCounts$(db, F_{DB} \cup N_{DB})$; // **first pass over** db
2. $F_{known} =$ GetFrequent$(F_{DB} \cup N_{DB}, minsup * |DB \cup db|)$;
3. $Infrequent = (F_{DB} \cup N_{DB}) - F_{known}$ // used later for pruning
4. **if** $(F_{DB} == F_{known})$
5. **return**(F_{DB}, N_{DB});
6. $N_{known} =$ NegBorder(F_{known});
7. **if** $(N_{known} \subseteq Infrequent)$
8. get supports of itemsets in N_{known} from $Infrequent$
9. **return**(F_{known}, N_{known});
10. $N^u = N_{known} - Infrequent$;
11. UpdateCounts(db, N^u); // **second pass over** db
12. $C =$ GetFrequent$(N^u, minsup * |db|)$;
13. $Infrequent_{db} = N^u - C$ // used later for pruning
14. **if** $(|C| > 0)$
15. $C = C \cup F_{known}$
16. ResetCounts(C);
17. **do** // compute negative border closure
18. $C = C \cup$ NegBorder(C);
19. $C = C - (Infrequent \cup Infrequent_{db})$ // prune
20. **until** C does not grow
21. $C = C - (F_{known} \cup N^u)$
22. **if** $(|C| > 0)$
23. UpdateCounts(db, C); // **third (and final) pass over** db
24. $ScanDB =$ GetFrequent$(C \cup N^u, minsup * |db|)$;
25. $N' =$ NegBorder$(F_{known} \cup ScanDB) - Infrequent$;
26. get supports of itemsets in N' from $(C \cup N^u)$
27. UpdateCounts$(DB, N' \cup ScanDB)$; // **first (and only) pass over** DB
28. $F_{DB \cup db} = F_{known} \cup$ GetFrequent$(ScanDB, minsup * |DB \cup db|)$;
29. $N_{DB \cup db} =$ NegBorder$(F_{DB \cup db})$;
30. get supports of $N_{DB \cup db}$ from $(Infrequent \cup N')$
31. **return**$(F_{DB \cup db}, N_{DB \cup db})$;
end

Figure 6.1: The DELTA Incremental Mining Algorithm

N_{DB} may become frequent and some itemsets in F_{DB} may become infrequent. Let the resultant set of frequent itemsets be F_{known}. These frequent itemsets are extracted using the function GetFrequent (line 2). The remaining itemsets are put in $Infrequent$ (line 3), and are later used for pruning candidates. The algorithm terminates if no itemsets have moved from N_{DB} to F_{known} (lines 4–5). This is valid due to the following Theorem presented in [T+97]:

Theorem 11 *If X is an itemset that is not in F_{DB} but is in $F_{DB \cup db}$, then there must be some subset x of X which was in N_{DB} and is now in $F_{DB \cup db}$.*

Hence, 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.

Second Pass over the Increment

On the other hand, if some itemsets do move from N_{DB} to F_{known}, then the negative border N_{known} of F_{known} is computed (line 6), using the AprioriGen [AS94] function. Itemsets in N_{known} with unknown counts are stored in a set N^u (line 10). The remaining itemsets in N_{known} i.e. with known counts, are all infrequent. Therefore, the only itemsets that may be frequent (and are not yet known to be so) are those in N^u and their extensions. If there are no itemsets in N^u, the algorithm terminates (lines 7-9).

Now, any itemset in N^u that is not locally frequent in db cannot be frequent in $DB \cup db$. Further, none of its extensions can be frequent as well. This is based on the following observation of [CHNW96]:[1].

Theorem 12 *An itemset can be present in $F_{DB \cup db}$ only if it is present in either F_{DB} or F_{db} (or both).*

[1] This observation applies only to the equi-support case

Therefore, a second pass over the increment is made to find the counts within db of N^u (line 11). Those itemsets that turn out to be infrequent in db are stored in a set called $Infrequent_{db}$ (line 13), which is later used for pruning candidates.

Third (and Final) Pass over the Increment

We then form all possible extensions of F_{known} which could be in $F_{DB \cup db} \cup N_{DB \cup db}$ and store them in set C. This is done by computing the remaining layers of the negative border closure of F_{known} (lines 15–20). (We expect that the remaining layers can be generated together since the number of 2-itemsets in F_{known} 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 $Infrequent$ and $Infrequent_{db}$ are removed so that none of their extensions are generated (line 19). After all the layers are generated, itemsets from F_{known} and N^u are removed from C since their counts within $DB \cup db$ and db respectively, are already available (line 21). The third (and final) pass over db is then made to find the counts within db 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 frequent in db need to be counted over DB as well to establish whether they are frequent 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($F_{known} \cup ScanDB$). From this the itemsets in $Infrequent$ 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 frequent itemsets from $ScanDB$ are gathered to form $F_{DB \cup db}$ (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 $Infrequent$ and N' (line 30). Thus we obtain the final set of frequent itemsets $F_{DB \cup db}$ and its negative border $N_{DB \cup db}$.

Results for the Increment

Performing the above steps results in the generation of $F_{DB \cup db}$ and $N_{DB \cup db}$ along with their supports. But, as mentioned earlier, we also need to generate the mining results for solely the increment, namely, $F_{db} \cup N_{db}$. To achieve this, the following additional processing is carried out during the above-mentioned passes:

After the first pass over the increment, we have the updated counts of all the itemsets in $F_{DB} \cup N_{DB}$. Therefore, the counts of these itemsets with respect to the increment alone is very easily determined by merely computing the differences between the updated counts and the original counts. After this computation, the itemsets that turn out to be frequent within *db* are gathered together and their negative border is computed.

If the counts within *db* of some itemsets in the negative border are unknown, these counts are determined during the second pass over the increment. Subsequently, the negative border closure of the resultant frequent itemsets (over *db*) is computed and the counts within *db* of the itemsets in the closure are determined during the third pass over the increment. Finally, the identities and counts within *db* of itemsets in $F_{db} \cup N_{db}$ are extracted from the closure.

In the above, note that a particular itemset could be a candidate for computing $F_{db} \cup N_{db}$, as well as $F_{DB \cup db} \cup N_{DB \cup db}$. To ensure that there is no unnecessary duplicate counting, all such common itemsets are identified and two counters are maintained for each of them: the first counter initially stores the itemset's support in DB, while the second stores the support in *db*. After the support in *db* is computed, the first counter is incremented by this value – it then reflects the support in $DB \cup db$.

6.2.2 Generating Hierarchical Association Rules

The processing steps described in the previous sub-section are completely sufficient to deliver the desired mining outputs for boolean databases. We now move on to describing how it is easily possible to extend the DELTA design to also handle the generation of association rules for *hierarchical* databases.

The hierarchical rule mining problem is to find association rules between items at any

level of a given taxonomy graph (*is-a* hierarchy). An obvious but inefficient solution to this problem is to reduce it to a boolean mining context using the following strategy: While reading each transaction from the database, dynamically create an "augmented" transaction that also includes all the ancestors of all the items featured in the original transaction. Now, any of the boolean mining algorithms can be applied on this augmented database.

A set of optimizations to improve upon the above scheme were introduced in [SA95] as part of the Cumulate (first-time) hierarchical mining algorithm. Interestingly, we have found that these optimizations can be utilized for incremental mining as well, and in particular, can be cleanly integrated in the core DELTA algorithm. In the remainder of this sub-section, we describe the optimizations and their incorporation in DELTA.

Cumulate Optimizations

Cumulate's optimizations for efficiently mining hierarchical databases are the following:

- **Pre-computing ancestors.** Rather than finding the ancestors for each item by traversing the taxonomy graph, the ancestors for each item are precomputed and stored in an array.
- **Filtering the ancestors added to transactions.** While reading a transaction from the database, it is not necessary to augment it with *all* ancestors of items in that transaction. Only ancestors of items in the transaction that are also present in some candidate itemset are added.
- **Pruning itemsets containing an item and its ancestor.** A candidate itemset that contains both an item and its ancestor may be pruned. This is because it will have exactly the same support as the itemset which doesn't contain that ancestor and is therefore redundant.

Incorporation in DELTA

The above optimizations are incorporated in DELTA in the following manner:

1. The first optimization is performed only in routines that access the database and therefore do not affect the structure of the DELTA algorithm.

2. The second optimization is performed before each pass over the increment or previous database. Ancestors of items that are not part of any candidate are removed from the arrays of ancestors that were precomputed during the first optimization.

3. The third optimization is performed only once and that is at the end of the first pass over the increment. At this stage the identities of all potentially frequent 2-itemsets (over $DB \cup db$) are known, and hence no further candidate 2-itemsets will be generated. Among the potentially frequent 2-itemsets, those that contain an item and its ancestor are pruned. It follows that candidates generated from the remaining 2-itemsets will also have the same property, i.e. they will not contain an item and its ancestor. Hence this optimization does not need to be applied again.

As a side-note, we add here that due to the generic nature of the above optimizations, they could be incorporated into other frequent itemset generation algorithms such as ARMOR, g-ARMOR and g-Apriori discussed in previous chapters.

6.2.3 Rationale for the DELTA Design

Having described the mechanics of the DELTA design, we now provide the rationale for its construction:

Let F_{known} be the set of frequent itemsets in $F_{DB} \cup N_{DB}$ that survive the support requirement after their counts have been updated over db, and N_{known} be its negative border. Now, if the counts of all the itemsets in N_{known} are available, then the final output is simply $F_{known} \cup N_{known}$. Otherwise, the only itemsets that may be frequent (and are not yet known to be so) are those in N_{known} with unknown counts and their extensions – by virtue of Theorem 11. At this juncture, we can choose to do one of the following:

Complete Closure: Generate the complete closure of the negative border, that is, all extensions of the itemsets in N_{known} with unknown counts. While generating the

extensions, itemsets that are known to be infrequent may be removed so that none of their extensions are generated. After the generation process is over, find the counts of all the generated itemsets by performing one scan over $DB \cup db$. We now have all the information necessary to first identify $F_{DB \cup db}$, and then the associated $N_{DB \cup db}$.

Layered Closure: Instead of generating the entire closure at one shot, generate the negative border "a layer at a time". After each layer is computed, update the counts of the itemsets in the layer by performing a scan over $DB \cup db$. Use these counts to prune the set of itemsets that will be used in the generation of the next layer.

Hybrid Closure: A combination of the above two schemes, wherein the closure is initially generated a layer at a time, and after a certain number of layers are completed, the remaining complete closure is computed. The number of layers upto which the closure is generated in a layered manner is a design parameter.

The first scheme, Complete Closure, appears infeasible because it could generate *a very large number of candidates* if the so-called "promoted borders" [F+97], that is, itemsets that were in N_{DB} but have now moved to $F_{DB \cup db}$, 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 $2^{p_1}(|F_{known}| - p_1)$. 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 frequent itemset in F_{known} to form candidates. Therefore, even for moderate values of p_1, the number of candidates generated could be extremely large.

The second strategy, Layered Closure, avoids the above candidate explosion problem since it brings a pruning step into play after the computation of each layer. However, it has its own performance problem in that it may require several passes over the database, one per layer, and this could turn out to be very costly for large databases. Further, it becomes impossible to provide bounds on the number of passes that would be required for the mining process.

Therefore, in DELTA, we adopt the third hybrid strategy, wherein an initial Layered Closure approach is followed by a Complete Closure strategy. In particular, the Layered Closure is used only for the *first* layer, and then the Complete Closure is brought into play. This choice is based on the well-known observation that pruning typically has the maximum impact for itemsets of length two – that is, the number of 2-itemsets that turn out to be frequent is usually a small fraction of the possible 2-itemset candidates [PCY95a]. In contrast, the impact of pruning at higher itemset lengths is comparatively small.

To put it in a nutshell, the DELTA design endeavors to achieve a reasonable compromise between the number of candidates counted and the number of database passes, since these two factors represent the primary bottle-necks in association rule generation. That our choice of compromise results in good performance is validated in the experimental study described in Section 6.6.

6.3 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 previously considered in the literature. We expect 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., $minsup_{DB \cup db} > minsup_{DB}$), and *Weaker*, where the current threshold is lower (i.e., $minsup_{DB \cup db} < minsup_{DB}$). We now address each of these cases separately:

6.3.1 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 optimization is incorporated:

Initially, all itemsets which are not frequent w.r.t. $minsup_{DB \cup db}$ are removed from F_{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 frequent after counting over the increment *db*. They may also be part of the computed negative border closure (lines 15-20 in Figure 6.1). If so, then during the pass over DB their counts are not measured since they are already known. If the counts of all the itemsets in the closure are known, *the pass over DB becomes unnecessary.*

6.3.2 Weaker Support Threshold

The weaker support case is much more difficult to handle since the F_{DB} set now needs to be *expanded* but the identities of these additional sets cannot be deduced from the increment *db*. In particular, note that Theorem 12, which DELTA relied on for pruning candidates in the equi-support case, *no longer holds* when the support threshold is lowered since we cannot deduce that a candidate is infrequent over DB just because it is not present in $F_{DB} \cup N_{DB}$.

However, it is easy to observe that the output required in the weaker threshold case is a *superset* of what would be output had the support threshold not changed. This observation suggests a strategy by which the DELTA algorithm is executed as though the support threshold *had not changed*, while at the same time making suitable alterations to handle the support threshold change.

In DELTA, the above strategy is incorporated by generating extra candidates (as described below) based on the lowered support threshold. It is only for these candidates that Theorem 12 does not hold. Hence, it is necessary to find their counts over the entire database $DB \cup db$. This is done *simultaneously* while executing equi-support DELTA.

The pseudo-code for the complete algorithm is given as function **DeltaLow** in Figure 6.2, and is described in the remainder of this section. The important point to note here is that the enhanced DELTA requires only *one* additional pass over the entire database to produce the desired results.

First Pass over the Increment

As in the equi-support case, the counts of itemsets in F_{DB} and N_{DB} are updated over the increment db (line 1 in Figure 6.2). By this, some itemsets in N_{DB} may become frequent and some itemsets in F_{DB} may become infrequent. Let the resultant set of frequent itemsets (w.r.t. $minsup_{DB \cup db}$) be F_{known}. These frequent itemsets are extracted using the function GetFrequent (line 2). Itemsets in the negative border of F_{known} with unknown counts are computed as $\mathsf{NegBorder}(F_{known}) - (F_{DB} \cup N_{DB})$. We refer to this set as $NBetween$ since these itemsets are likely to have supports *between* $minsup_{DB}$ and $minsup_{DB \cup db}$ (line 3). For these itemsets, Theorem 12 does not hold due to the lowered support threshold.

Remaining Passes of Equi-Support DELTA

The remaining passes of equi-support DELTA are executed for the previous support $minsup_{DB}$. A difference, however, is that the counts of itemsets in $NBetween$ over $DB \cup db$ are simultaneously found (line 4).

Among the candidates generated during the remaining passes of equi-support DELTA, some may already be present in $NBetween$. To ensure that there is no unnecessary duplicate counting, all such common itemsets are identified and only one copy of each is retained during counting.

Additional Pass over the Entire Database

At the end of the above passes, the counts of all 1-itemsets and 2-itemsets of $F_{DB \cup db} \cup N_{DB \cup db}$ are available. The counts of 1-itemsets are available because $F_{DB} \cup N_{DB}$ contains all possible 1-itemsets [T+97], while the counts of all required 2-itemsets are available because F_{known} contains all frequent 1-itemsets in $DB \cup db$ and $NBetween$ contains the immediate extensions of F_{known} that are not already in $(F_{DB} \cup N_{DB})$. Therefore, it becomes possible to generate the negative border closure of all known frequent itemsets without encountering the "candidate explosion" problem described for the Complete Closure approach in Section 6.2.3.

Let F' be the set of all frequent itemsets whose counts are known (line 5), and let $Infrequent$ be the set of itemsets with known counts which are not in F' (line 6). If the counts of the negative border of F' are already known, then the algorithm terminates (lines 7–9). Otherwise, all the remaining extensions of F' that could become frequent 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 F' 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 frequent itemsets and its negative border are determined (lines 18–20).

When $minsup_{DB \cup db} << minsup_{DB}$

We discuss here the behaviour of DELTA when $minsup_{DB \cup db}$ is *much less* than $minsup_{DB}$. We expect this case to be *especially troublesome* because the new mining results would be a much larger set than the previous mining results. This means that the previous mining results would not be very useful in determining the new results. To simplify the discussion, let us consider the extreme case when $minsup_{DB}$ is so high that $F_{DB} = \phi$. We show that in this case, DELTA reduces to Apriori with the modification that all database scans of Apriori beyond the second pass are combined. Since, as discussed in Section 6.2.3, the impact of pruning for itemsets of length greater than two is relatively small, it follows that DELTA reduces to a *re-mining* of the entire database when the previous mining results are not useful.

For the above scenario, DELTA first updates the counts of itemsets in N_{DB} (which consists of all possible 1-itemsets) over db (line 1 in Figure 6.2). This is equivalent to (a part of) the first pass of Apriori. After the frequent 1-itemsets are obtained (line 2), candidate 2-itemsets are generated (line 3). Next, the counts of these candidate 2-itemsets over $DB \cup db$ are found (line 4). Note that while performing line 4, some candidate 2-itemsets may be generated; but these would be a subset of the candidates generated in line 3. Since these candidates are combined before their counts over $DB \cup db$ are determined,

no redundant work is done.

After gathering the frequent 2-itemsets (line 5), their negative border closure is computed (lines 10–16) and the counts of itemsets in this closure over $DB \cup db$ are determined (line 17). This corresponds to combining all passes of Apriori beyond the second pass. Finally, the frequent itemsets and negative border information is gathered and output (lines 18–21).

6.4 Integrating ARMOR & g-ARMOR with DELTA

As mentioned earlier, our work on incremental mining was actually done prior to our work on the other two issues addressed in this thesis. However, we have presented it in the end for pedagogical reasons. In this Section, we provide a sketch of how the techniques presented in earlier chapters can be integrated into the DELTA scheme.

Notice that the counting technique in DELTA has been abstracted using the function UpdateCounts. Hence, any efficient data-structure could be implemented to store the counters of itemsets. For instance, we could use the counting scheme developed in Chapter 4 for the Oracle and ARMOR algorithms that uses a DAG data-structure. The only modification that this would require to the DELTA scheme is that like in ARMOR it would be necessary to compute and store the supports of marginally more candidates than the frequent itemsets and its negative border. As mentioned in Chapter 4, the maximum number of additional candidates for all the databases and support specifications considered in our empirical study was only about *ten percent* more.

With the above modifications in place, it is possible to integrate g-ARMOR with DELTA in a manner similar to integrating the g-closed itemset framework into ARMOR. The input to DELTA now contains only the frequent g-closed itemsets, its negative border and the marginally additional candidates that would be generated by g-ARMOR (by virtue of it being a generalization of ARMOR). During the execution of this modified DELTA, for any candidates X, Y where $Y \supset X$, if $support(X) \approx support(Y)$, then prune all supersets of Y. Alternatively, for any X, Y where $Y \supset X$, if $support(X) \approx support(Y)$ holds only upto some partition and then no longer holds, then regenerate all supersets of

Y using the technique described in Section 5.6 of Chapter 5.

6.4.1 Multi-Tolerance Incremental Mining

In Section 6.3, we have extended the DELTA algorithm to handle *multi-support* environments, where the minimum support specified by the user for the current database differs from that specified for the previous database. A related issue arises when the *tolerance* threshold specified by the user for the current database differs from that specified for the previous database. For convenience, similar to multi-support mining, we break up the multi-tolerance problem into two cases: *Stronger*, where the current threshold is higher, and *Weaker*, where the current threshold is lower. We now address each of these cases separately:

Stronger Tolerance Threshold

The stronger tolerance case is handled almost exactly the same way as the equi-tolerance case, that is, as though the threshold has *not changed*. The only difference is that the following optimization is incorporated: Initially, for all available itemsets $X, Y : Y \supset X$, if $support(X) \approx support(Y)$ w.r.t. the new tolerance, then prune all supersets of Y.

Weaker Tolerance Threshold

The weaker tolerance case is much more difficult to handle since the previous mining results now need to be *expanded* but the *supports* of the additional required itemsets cannot be estimated with the desired accuracy. In DELTA, this case is handled as follows: For all available itemsets X, Y where $Y \supset X$, check if $support(X) \approx support(Y)$ w.r.t. the old tolerance and *not* w.r.t. the new tolerance. This indicates that supersets of Y had been pruned in the previous mining using ϵ-equal support pruning. Therefore, we now regenerate all such supersets of Y using the technique described in Section 5.6 of Chapter 5. However, we cannot estimate the supports of these itemsets even approximately (w.r.t. the new tolerance) by utilizing the currently mined results. Hence, these supports

are obtained by making a pass over DB after which, the processing continues as for the equi-tolerance case.

6.5 Performance Study

In the previous sections, we presented the **FUP**, **Borders** and **TBAR** incremental mining algorithms, apart from our new **DELTA** algorithm. To evaluate the relative performance of these algorithms and to confirm the claims that we have 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. (Note that, as mentioned in Section 3.3 of Chapter 3, both FUP and Borders do not compute the mining results for solely the increment, and hence their execution times do not include the additional processing required to generate these results.)

6.5.1 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 frequent itemsets (and the associated negative border) in the current database and increment and only needs to gather their corresponding supports. Note that this idealized incremental algorithm represents the *absolute minimal amount* of processing that is necessary and therefore represents a lower bound[2] on the (execution

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

time) performance.

The Oracle algorithm operates as follows: For those itemsets in $F_{DB \cup db} \cup N_{DB \cup db}$ 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 $F_{DB \cup db} \cup N_{DB \cup db}$. During the pass over db, it also determines the counts within db of itemsets in $F_{db} \cup N_{db}$. Duplicate candidates are avoided by retaining only one copy of each of them. So, in the worst case, it needs to make one pass over the previous database and one pass over the increment.

For evaluating the performance of DELTA on hierarchical databases, we compared it with **Cumulate** and Oracle as no previous incremental algorithms are available for comparison. We chose Cumulate among the algorithms proposed in [SA95] since it performed the best on most of our workloads. The hierarchical databases were generated using the same technique as in [SA95].

6.5.2 Database Generation

Parameter	Meaning	Values
N	Number of items	1000
T	Mean transaction length	10
P	Number of potentially frequent itemsets	2000
I	Mean length of potentially frequent 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 frequent itemset identity	0.33 (for Skewed)
p_{it}	Prob. of changing item identity	0.50 (for Skewed)

Table 6.2: Parameter Table

The databases used in our experiments were synthetically generated using the technique described in [AS94] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator are described in Table 6.2. These are similar to those used in [AS94] except that the size and skew of the

Parameter	Value
Number of roots	250
Number of levels	4
Fanout	5
Depth-ratio	1

Table 6.3: Taxonomy Parameter Table

increment are two additional parameters. Since the generator of [AS94] does not include the concept of an increment, we have taken the following approach, similar to [CHNW96]: The increment is produced by first generating the entire $DB \cup db$ and then dividing it into DB and db.

Additional parameters required for the taxonomy in our experiments on hierarchical databases are shown in Table 6.3. The values of these parameters are identical to those used in [SA95].

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 frequent 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 frequent itemsets are changed. A potentially frequent 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 frequent itemsets are changed in this manner, d number of transactions are produced with the new modified set of potentially frequent itemsets.

6.5.3 Itemset Data Structures

In our implementation of the incremental mining algorithms, we generally use the *hashtree* data-structure [AS94] as a container for itemsets. However, like in Chapter 4, the 2-itemsets are not stored in hashtrees but instead in a 2-dimensional array which is indexed by the frequent 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.

6.5.4 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 6.2 for our experiments on non-hierarchical (boolean) databases.

The parameters settings used in our experiments on hierarchical databases are identical except for the number of items (N) and the number of potentially frequent itemsets (P) which were both set to 10000. The specific values of additional parameters required for the taxonomy are shown in Table 6.3.

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%. For our experiments on hierarchical databases, 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.

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 6.2. With these settings, at the 0.5 percent support threshold and a 10% increment, for example, there are over 700 frequent itemsets in db which are not frequent in DB, and close to 500 frequent itemsets in DB that are not frequent 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 6.2 are considered in these experiments.

6.6 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.

6.6.1 Experiment 1: Flat / Equi-support / Identical Distribution

Our first experiment considers the equi-support situation with identical distribution between DB and db on boolean databases. For this environment, the execution time performance of all the mining algorithms is shown in Figures 6.3a–d for increment sizes ranging from 1% to 100%.

Focusing first on FUP, we see in Figure 6.3 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 $F_{DB} \cup N_{DB}$ 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 6.3a, 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 3.3. 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 6.3b–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 frequent 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 6.3c), 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.

6.6.2 Experiment 2: Flat / Equi-support / Skewed Distribution

Our next experiment considers 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 6.4a–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 infrequent 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 frequent 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* frequent itemsets, and not by the frequent 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 infrequent either over $DB \cup db$ or over db are pruned away during closure generation, and not merely those candidates which are infrequent over db. Hence, DELTA is relatively stable with respect to data skew. As in the Identical distribution case, it can be seen in Figures 6.4a–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 $F_{DB} - F_{DB \cup db}$ increases, resulting in more unnecessary candidates being updated over db, and (2) the number of itemsets in $F_{DB \cup db} - F_{DB}$ 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.

6.6.3 Experiment 3: Flat / Multi-Support / Identical Distribution

The previous 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 6.5a–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.

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 6.5a–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 frequent 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 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 frequent and hence both algorithms perform almost the same amount of processing.

6.6.4 Experiment 4: Flat / Multi-Support / Skewed Distribution

Our next experiment evaluates the same environment as that of the previous experiment, except that the distribution of the increments is Skewed with respect to the original database. The execution time performance for this case is shown in Figures 6.6a–d. We see here that the relative performance of the algorithms is very similar to that seen for the Identical workload environment. Further, as in the equi-support skewed case (Experiment 2), DELTA is stable with respect to skew since it uses information from both DB and db to prune away candidates. Only when the increment size is 100% do we notice some degradation in the performance of DELTA. However, it performs slightly better than Apriori even for this large increment.

6.6.5 Experiment 5: Hierarchical / Equi-support / Identical Distribution

The previous experiments were conducted on boolean databases. We now move on to experiments conducted on *hierarchical* databases. In these experiments, we compare the performance of DELTA with that of Cumulate and Oracle only since, as mentioned earlier, no incremental algorithms are available for comparison. The execution time performance of the various algorithms for this case is shown in Figures 6.7a–d. Note that the time taken to complete mining is measured in *hours* here as compared to the *minutes* taken in the previous experiments. The reason for this large increase is that the number of frequent itemsets is much more (about 10–15 times) – this is because itemsets can be formed both within and across levels of the item taxonomy graph.

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 6.7a–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 because the set of frequent itemsets with its negative border is relatively stable, and DELTA prunes away most of the unnecessary candidates in its second pass over the increment.

6.6.6 Experiment 6: Hierarchical / Equi-support / Skewed Distribution

Our next experiment considers 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 6.8a–d.

As in the Identical distribution case, it can be seen in Figures 6.8a–b that for small increment sizes, the performance of DELTA almost coincides with that of Oracle. The stability of DELTA with regard to data skew is again attributed to the fact that all itemsets that are known to be infrequent either over $DB \cup db$ or over db are pruned away during closure generation, and not merely those candidates which are infrequent over db. The performance of DELTA however degrades to some extent for large skewed increments. This is because of the same reasons as in the Flat/Equi-support/Skewed Distribution case (Experiment 2): (1) the number of itemsets in $F_{DB} - F_{DB \cup db}$ increases, resulting in more unnecessary candidates being updated over db, and (2) the number of itemsets in $F_{DB \cup db} - F_{DB}$ 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 Cumulate. For example, for a minimum support of 0.75% and an increment of 100%, its performance is more than 35% as good as that of Cumulate.

6.6.7 Experiment 7: Hierarchical / Multi-support / Identical Distribution

The previous two experiments modeled the equi-support environment for mining over hierarchical databases. We now move on to considering *multi-support* environments over

these databases.

In this experiment, we fixed the initial support to be 1.5% and the new support was varied between 0.75% and 2.5%, thereby covering both the Weaker Threshold and Stronger Threshold possibilities. For this environment, Figures 6.9a–d show the performance of DELTA relative to that of Cumulate and Oracle for the databases where the distribution of the increments is Identical to that of the previous database.

We note here that at either end of the support spectrum, DELTA performs very similarly to Cumulate whereas in the "middle band" it does noticeably better, especially for moderate increment sizes (Figures 6.9a–b). This is similar to the relationship between DELTA and Apriori in the Flat/Multi-support/Identical Distribution case (Experiment 3). 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 frequent 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 Cumulate. Most of the candidates will have support less than the previous minimum threshold, and hence all of them have to be counted over the previous database. Therefore, the performance of DELTA approaches that of Cumulate in the low support region. In the high support region, on the other hand, most of the candidates do not turn out to be frequent and hence both algorithms perform almost the same amount of processing.

6.6.8 Experiment 8: Hierarchical / Multi-support / Skewed Distribution

Our next experiment evaluates the same environment as that of the previous experiment, except that the distribution of the increments is Skewed with respect to the original database. The execution time performance for this case is shown in Figures 6.10a–d. We see here that the relative performance of the algorithms is very similar to that seen for the Identical workload environment. Further, as in the Hierarchical/Equi-support/Skewed case (Experiment 6), DELTA is stable with respect to skew since it uses information from

both DB and db to prune away candidates. Only for large increment sizes (Figures 6.10c–d) do we notice some degradation in the performance of DELTA. However, it performs slightly better than Cumulate even for these increments.

6.7 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 frequent 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 which is the result of a synthesis of existing algorithms, designed to address each of their specific limitations. It 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 frequent itemset. Also, DELTA does not suffer from the candidate explosion problem associated with the Borders algorithm owing to its better pruning strategy.

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 hierarchical association rules, while maintaining its performance close to Oracle. No prior work exists on extending incremental mining algorithms to handle hierarchical rules.

In summary, DELTA is a practical, robust and efficient incremental mining algorithm.

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

Figure 6.2: DELTA for Weaker Support Threshold (DeltaLow)

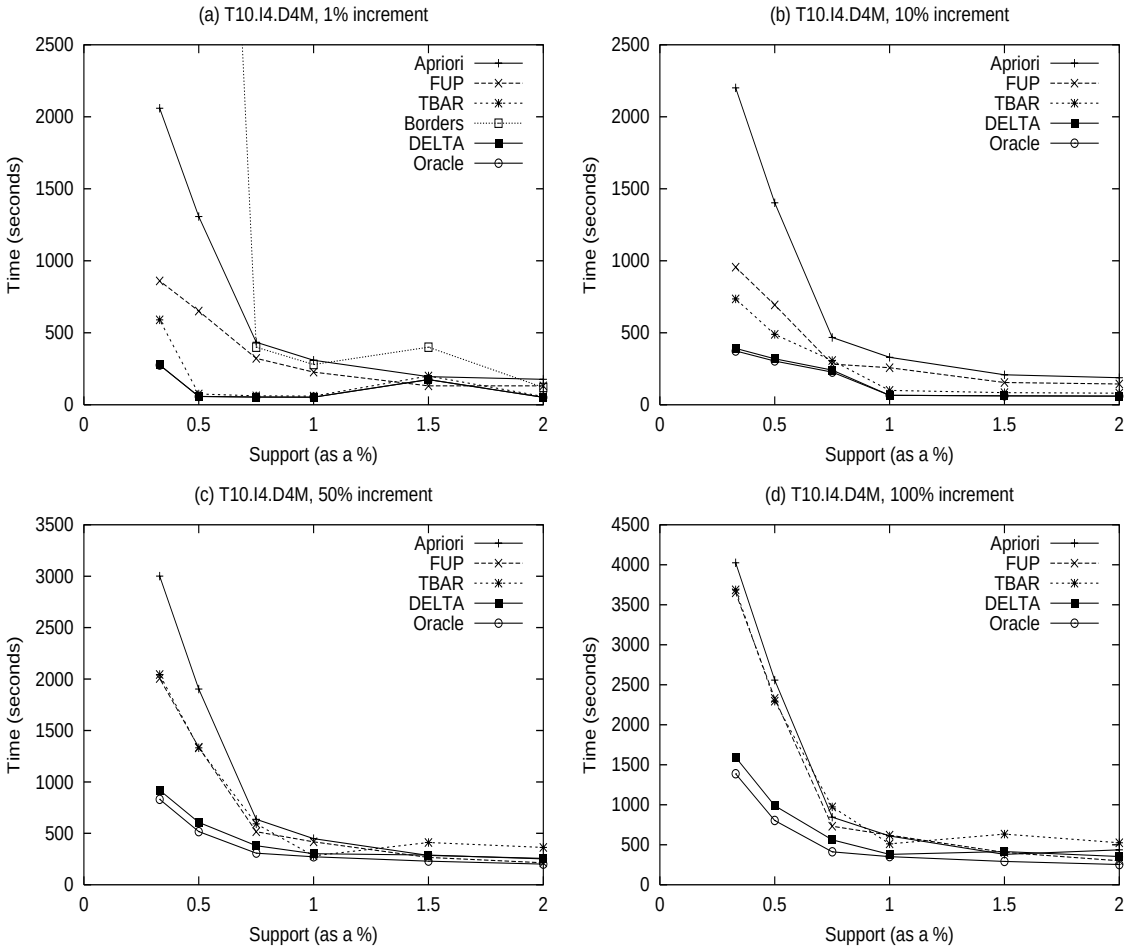

Figure 6.3: Flat / Equi-support / Identical Distribution

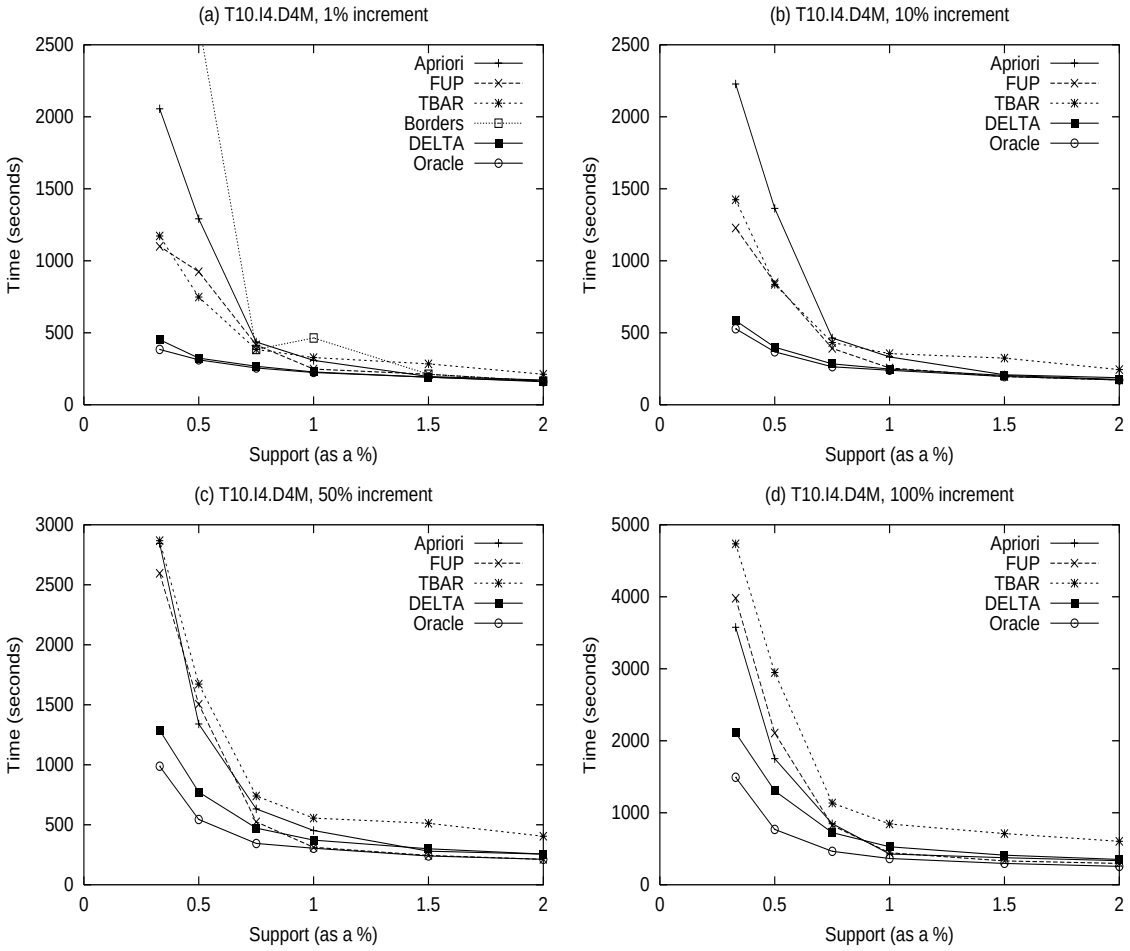

Figure 6.4: Flat / Equi-support / Skewed Distribution

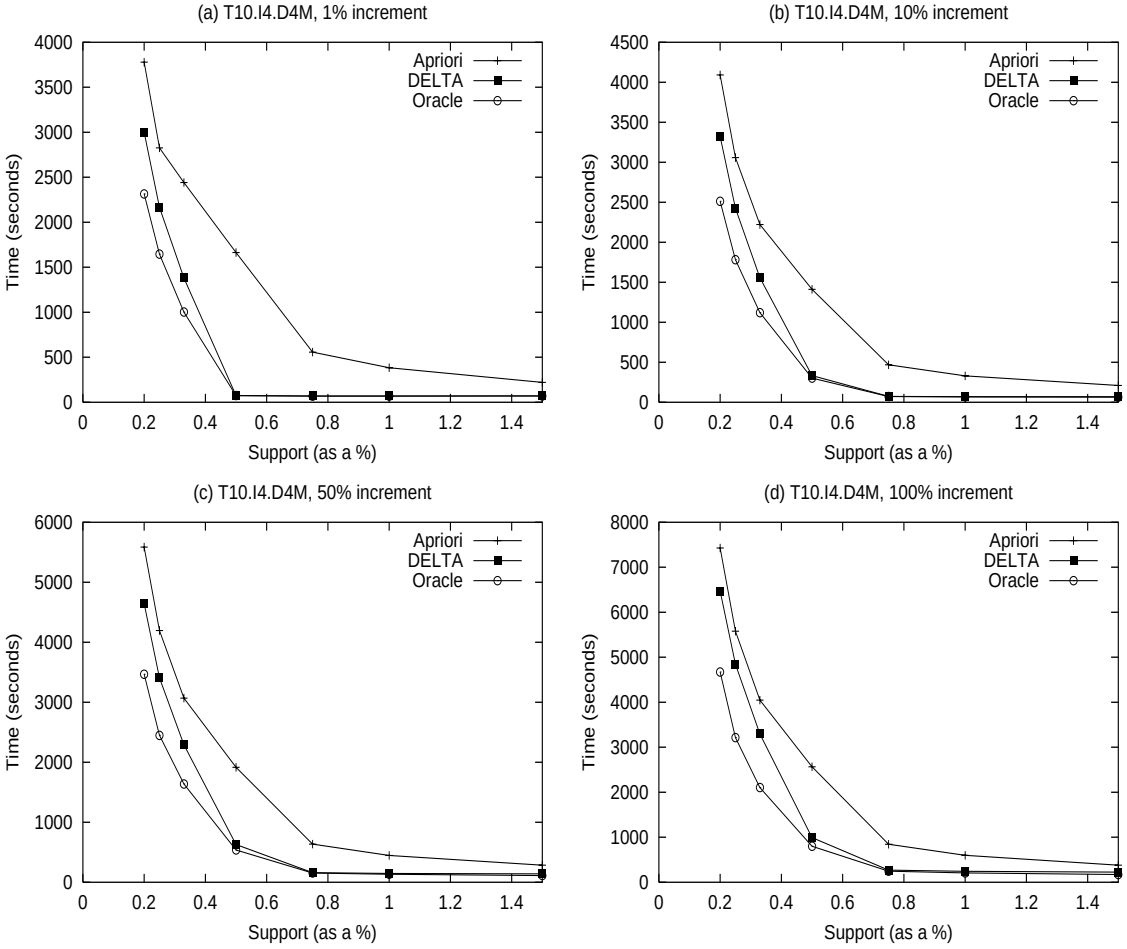

Figure 6.5: **Flat / Multi-Support / Identical Distribution [Previous Support = 0.5%]**

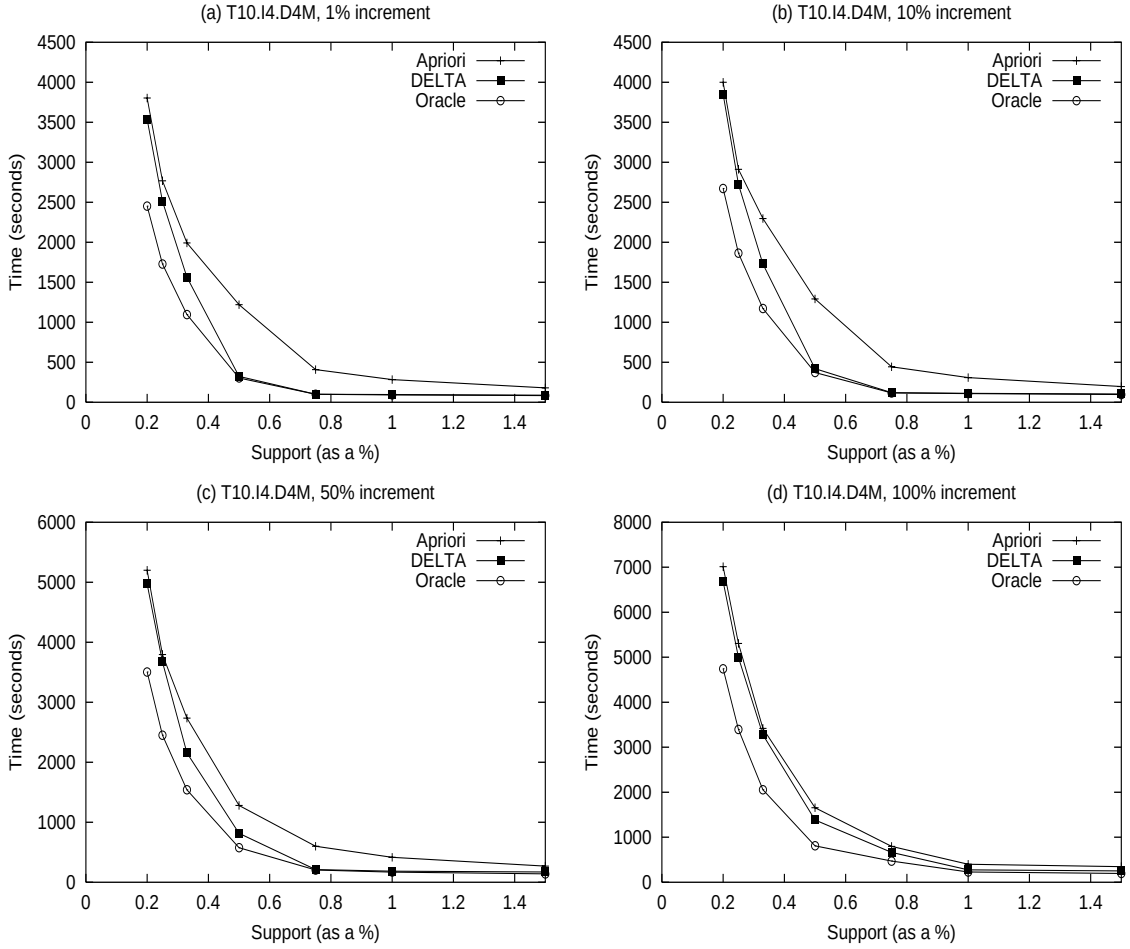

Figure 6.6: **Flat / Multi-Support / Skewed Distribution [Previous Support = 0.5%]**

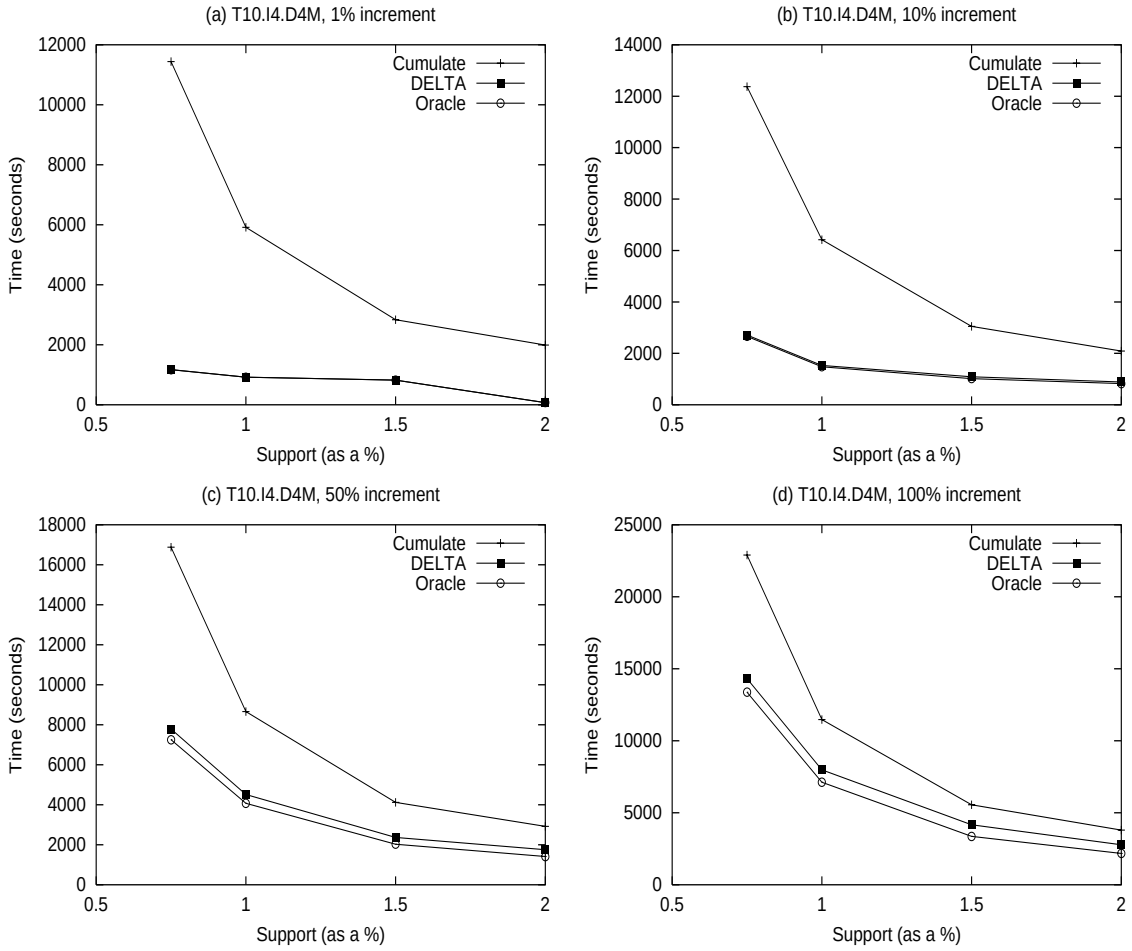

Figure 6.7: Hierarchical / Equi-support / Identical Distribution

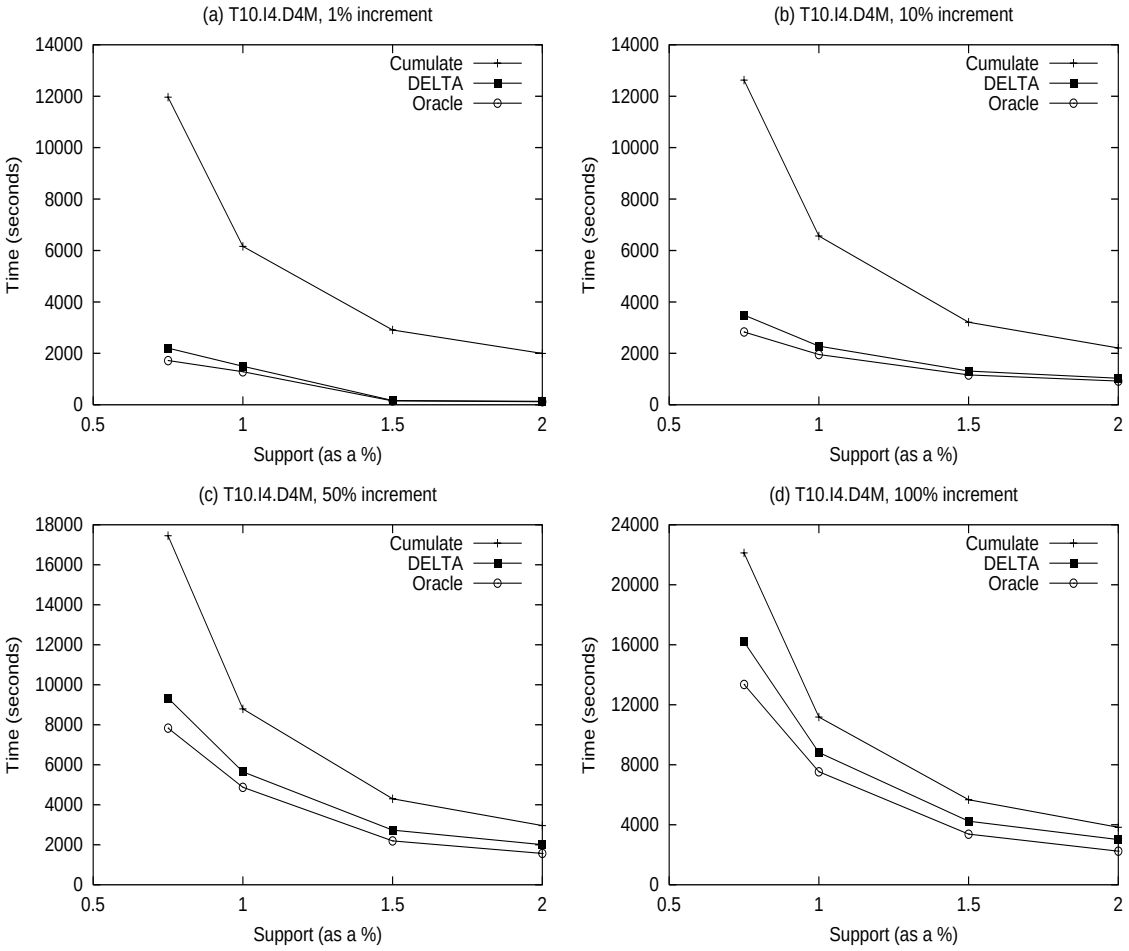

Figure 6.8: Hierarchical / Equi-support / Skewed Distribution

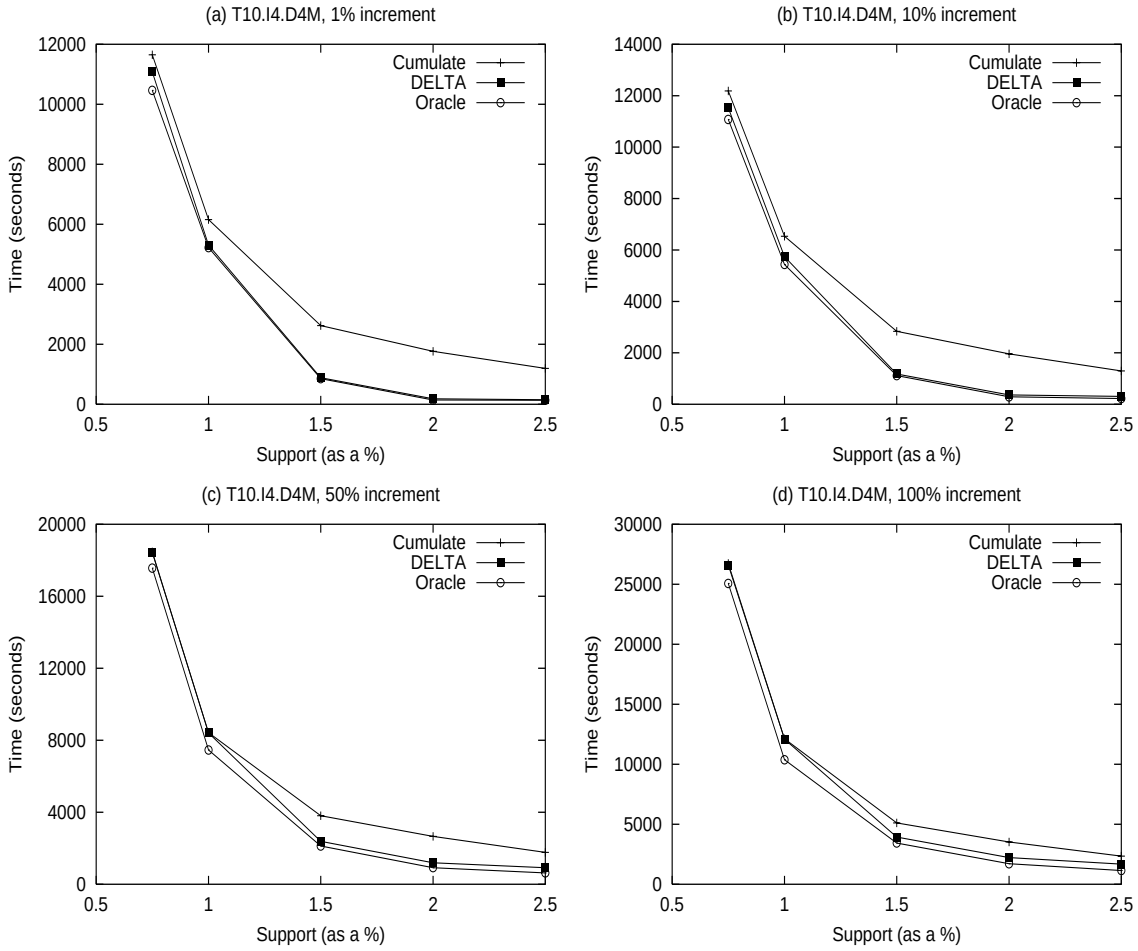

Figure 6.9: **Hierarchical / Multi-support / Identical Distribution [Previous Support = 1.5%]**

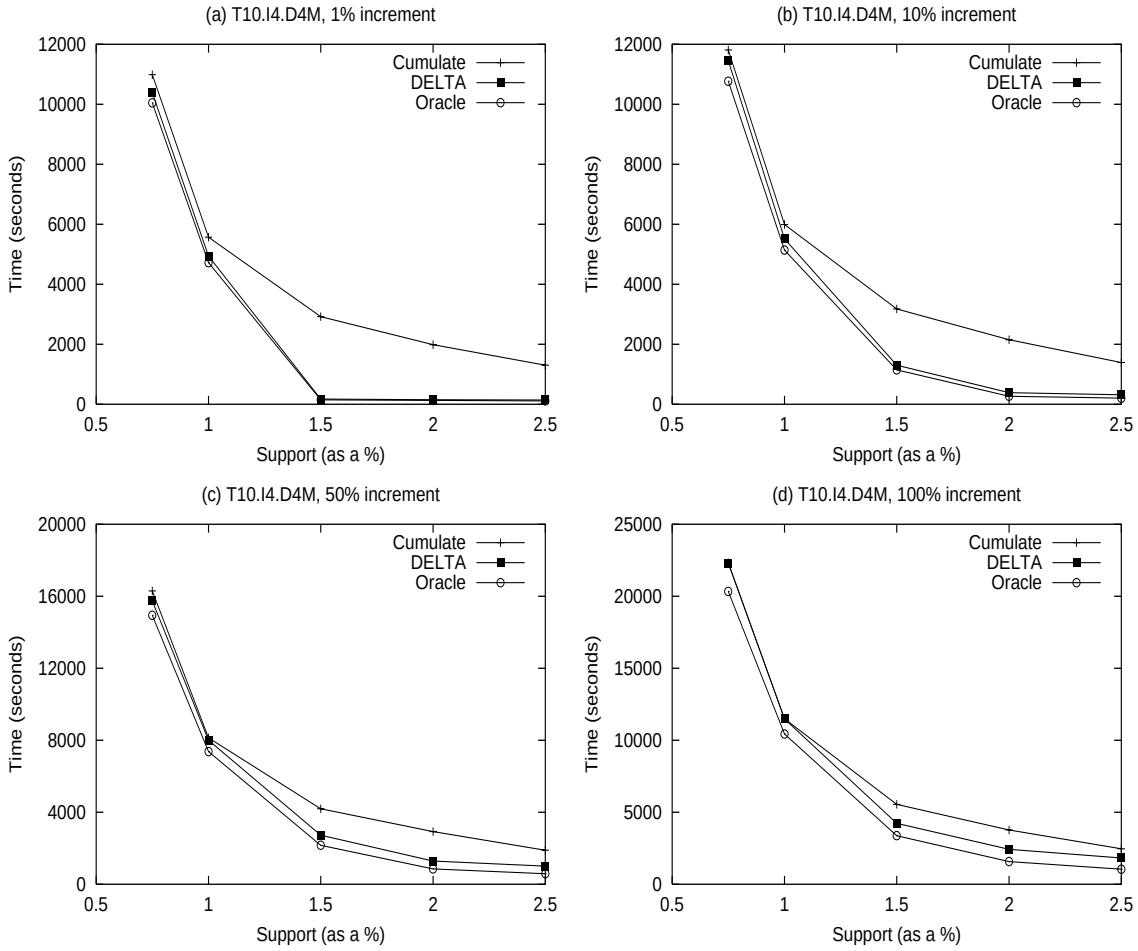

Figure 6.10: **Hierarchical / Multi-support / Skewed Distribution [Previous Support = 1.5%]**

Chapter 7

Conclusions and Future Research

7.1 Summary of Contributions

In this thesis, we have investigated three issues in association rule mining – the efficiency of algorithms, the conciseness of results and the problem of re-mining. We summarize our contributions in each of these areas below.

7.1.1 Issue 1: Efficiency of Algorithms

A variety of novel algorithms have been proposed in the recent past for the efficient mining of association rules, each in turn claiming to outperform its predecessors on a set of standard databases. In this thesis, our approach was to quantify the algorithmic performance of association rule mining algorithms with regard to an idealized, but practically infeasible, "Oracle". The Oracle algorithm utilizes a partitioning strategy to determine the supports of itemsets in the required output. It uses direct lookup arrays for counting singletons and pairs and a DAG data-structure for counting longer itemsets. We have shown that these choices are optimal in that only required itemsets are enumerated and that the cost of enumerating each itemset is $\Theta(1)$. Our experimental results showed that there was a substantial gap between the performance of current mining algorithms and that of the Oracle.

We also presented a new online mining algorithm called ARMOR (Association Rule

Mining based on ORacle), that was constructed with minimal changes to Oracle to result in an online algorithm. ARMOR utilizes a new method of candidate generation that is dynamic and incremental and is guaranteed to complete in two passes over the database. Our experimental results demonstrate that ARMOR performs within a *factor of two* of Oracle.

7.1.2 Issue 2: Conciseness of Results

In this thesis we proposed the generalized closed itemset framework (or g-closed itemset framework) in order to manage the information overload produced as the output of frequent itemset mining algorithms. This framework builds upon the original closed itemset concept over which it provides an order of magnitude improvement. This is achieved by relaxing the requirement for exact equality between the supports of itemsets and their supersets. Instead, our framework accepts that the supports of two itemsets are equal if the difference between their supports is within a user-specified tolerance factor.

We also presented two algorithms – g-Apriori (based on the classical levelwise Apriori algorithm) and g-ARMOR (based on our ARMOR algorithm) for mining the frequent g-closed itemsets. g-Apriori utilizes a new method for generating frequent g-closed itemsets from their generators. This new method avoids the costly additional pass that was required in the A-Close algorithm for mining frequent closed itemsets. g-Apriori is shown to perform significantly better than Apriori solely because the frequent g-closed itemsets are much fewer than the frequent itemsets. Finally, g-ARMOR was shown to perform over an order of magnitude better than Apriori over all databases and support specifications used in our experimental evaluation.

7.1.3 Issue 3: Re-mining

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 frequent 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 which is the result of a synthesis of existing algorithms, designed to address each of their specific limitations. It 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 frequent itemset. Also, DELTA does not suffer from the candidate explosion problem associated with the Borders algorithm owing to its better pruning strategy.

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

We showed empirically that for a variety of increment sizes, increment distributions and support thresholds, DELTA performs significantly better than the previously proposed incremental algorithms – FUP, TBAR and Borders. 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.

7.1.4 Overall Architecture

In this thesis, we have presented and evaluated three algorithms: ARMOR, g-ARMOR and DELTA for efficiently discovering association rules, generating concise rule summaries and maintaining discovered rules, respectively. In summary, the overall scheme for BAR-mining that we advocate is shown in Figure 7.1. The user inputs the database and the following mining parameters – minimum support, minimum confidence and the tolerance factor for support approximation. Next, the mining system applies the g-ARMOR

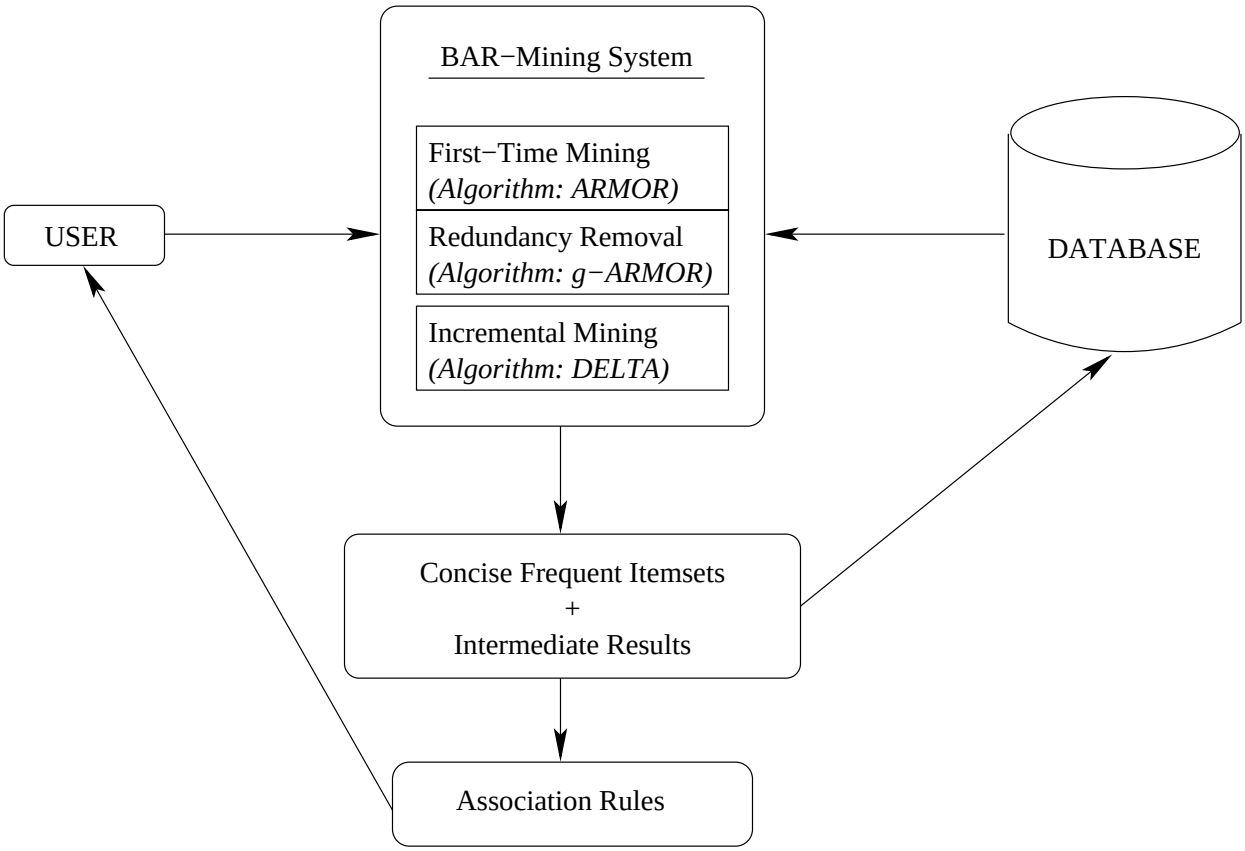

Figure 7.1: Architecture for BAR-mining

algorithm to produce concise frequent itemsets, which are then used to form the association rule that are presented to the user. For future mining runs, the system applies the DELTA algorithm, which utilizes previous mining results to efficiently re-mine the updated database.

7.2 Future Work

The work that we have presented in this thesis can be extended in the following ways:

1. **Backend:** From the point of view of building a commercial package for BAR-mining, our implementation of mining algorithms needs to be extended to handle commercial database backends. Since BAR-mining algorithms require only *sequential* access to the database, this extension is conceptually straight-forward. Various

alternative ways of integrating BAR-mining with relational database systems are presented and evaluated in [STA98, RCIC99, NT99].

2. **Frontend:** The algorithms designed and evaluated in this thesis address the computationally expensive step of BAR-mining, namely, to produce frequent itemsets. A commercial BAR-mining system would need to utilize the output of this step to produce association rules and present them to the end user using an intuitive and appealing frontend.

3. **Integration:** In this thesis, we have provided techniques for efficiently discovering association rules, generating concise rule summaries and maintaining discovered rules. We have also provided a sketch of how these various techniques could be combined into a single framework (in Section 6.4 of Chapter 6). It would be necessary to implement this integrated framework for a commercial package that utilizes the techniques presented in this thesis.

References

[AA01] D. Agrawal and C. C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proc. of ACM Principles of Database Systems (PODS)*, 2001.

[AAP98] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[AAP01] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Parallel and Distributed Computing*, March 2001.

[ABE+99] M. Atallah, E. Bertino, A. Elmagarmid, M. Ibrahim, and V. Verykios. Disclosure limitation of sensitive rules. In *Proc. of IEEE Knowledge and Data Engineering Exchange Workshop*, November 1999.

[AFLM99] Y. Aumann, R. Feldman, O. Lipsttat, and H. Mannila. Borders: An efficient algorithm for association generation in dynamic databases. *Journal of Intelligent Information Systems*, pages 61–73, April 1999.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, September 1994.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, March 1995.

[AS96] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Eng.*, December 1996.

[AS00] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[AY98] C. C. Aggarwal and P. S. Yu. Online generation of association rules. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, February 1998.

[BAG99] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, February 1999.

[Bay98] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[BB00] J-F. Boulicaut and A. Bykowski. Frequent closures as a concise representation for binary data mining. In *Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, April 2000.

[BBR00] J-F. Boulicaut, A. Bykowski, and C. Rigotti. Approximation of frequency queries by means of free-sets. In *Proc. of European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, September 2000.

[BM98] C. L. Blake and C. J. Merz. UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository. html, 1998.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1997.

[BMUT97] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1997.

[CHNW96] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, February 1996.

[CLK97] D. Cheung, S. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. of Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, April 1997.

[com97] Eachmovie collaborative filtering data set. http://www.research.compaq.com/SRC/eachmovie/, 1997.

[CR73] S. Cook and R. Reckhow. Time bounded random access machines. *Computer and System Sciences*, 7, 1973.

[CS02] L. Cristofor and D. Simovici. Generating an informative cover for association rules. Technical report, University of Massachusetts at Boston, 2002.

[CSD98] S. Chakrabarti, S. Sarawagi, and B. Dom. Mining surprising patterns using temporal description length. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, September 1998.

[CVB96] D. Cheung, T. Vincent, and W. Benjamin. Maintenance of discovered knowledge: A case in multi-level association rules. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1996.

[DL98] G. Dong and J. Li. Interestingness of discovered association rules in terms of neighborhood-based unexpectedness. In *Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, 1998.

[DVEB01] E. Dasseni, V. Verykios, A. Elmagarmid, and E. Bertino. Hiding association rules by using confidence and support. In *Proc. of Intl. Information Hiding Workshop*, April 2001.

[EGSA02] A. Evfimievski, J. Gehrke, R. Srikant, and R. Agrawal. Privacy preserving mining of association rules. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2002.

[F+97] R. Feldman et al. Efficient algorithms for discovering frequent sets in incremental databases. In *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[GRRS99] M. Garofalakis, S. Ramaswamy, R. Rastogi, and K. Shim. Of crawlers, portals, mice, and men: Is there more to mining the web? In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1999.

[GZ01] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proc. of Intl. Conf. on Data Mining (ICDM)*, November 2001.

[Hid99] C. Hidber. Online association rule mining. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1999.

[HK01] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[HKK97] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1997.

[HKKM97] E. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering based on association rule hypergraphs. In *Workshop on Research Issues on Data Mining and Knowledge Discovery*, August 1997.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[KC02] M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. In *Proc. of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2002.

[KMR+94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. of Intl. Conf. on Information and Knowledge Management (CIKM)*, November 1994.

[LD98] J. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, 1998.

[LHM98] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1998.

[LHM99] B. Liu, W. Hsu, and Y. Ma. Pruning and summarizing the discovered association rules. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1999.

[LK98] D. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *Proc. of Intl. Conf. on Extending Database Technology*, March 1998.

[LP02] Y. Lindell and B. Pinkas. Privacy preserving data mining. *Journal of Cryptology*, 15(3):177–206, 2002.

[MGKS97] H. Mannila, D. Gunopulos, R. Khardon, and S. Saluja. Data mining, hypergraph transversals, and machine learning. In *Proc. of ACM Principles of Database Systems (PODS)*, 1997.

[MM02] G. Manku and R. Motwani. Approximate frequency counts over streaming data. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, August 2002.

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. Technical report, University of Helsinki, 1997.

[NLHP98] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[NT99] S. Nestorov and S. Tsur. Integrating data mining with relational dbms: A tightly-coupled approach. In *Intl. Workshop on Next Generation Information Technologies and Systems*, July 1999.

[ORS98] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, February 1998.

[P+01] J. Pei et al. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. of Intl. Conf. on Data Mining (ICDM)*, December 2001.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of Intl. Conf. on Database Theory (ICDT)*, January 1999.

[PCY95a] J. Park, M. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1995.

[PCY95b] J. S. Park, M. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, November 1995.

[PH00] V. Pudi and J. Haritsa. Quantifying the utility of the past in mining large databases. *Information Systems*, July 2000.

[PH02a] V. Pudi and J. Haritsa. How good are association-rule mining algorithms? In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, February 2002.

[PH02b] V. Pudi and J. Haritsa. On the efficiency of association-rule mining algorithms. In *Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, May 2002.

[PH03a] V. Pudi and J. Haritsa. Generalized closed itemsets: A technique for improving the conciseness of rule covers. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, March 2003.

[PH03b] V. Pudi and J. Haritsa. Reducing rule covers with deterministic error bounds. In *Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, May 2003.

[PZOL01] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, February 2001.

[RCIC99] K. Rajamani, A. Cox, B. Iyer, and A. Chada. Efficient mining for association rules with relational database systems. In *Proc. of Intl. Database Engineering and Applications Symposium (IDEAS)*, August 1999.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, September 1995.

[SA96] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1996.

[SAM99] D. K. Subramanian, V. S. Ananthanarayana, and M. N. Murty. Generation of inter-database association rules based on semantic networks. Technical Report IISc-CSA-1999-6, CSA, Indian Institute of Science, 1999.

[SHS+00] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[SLR99] D. Shah, L. V. S. Lakshmanan, and K. Ramamritham. Interestingness and pruning of mined patterns. In *Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1999.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 1995.

[STA98] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1997.

[SVC01] Y. Saygin, V. Verykios, and C. Clifton. Using unknowns to prevent discovery of association rules. In *ACM SIGMOD Record*, 2001.

[SVE02] Y. Saygin, V. Verykios, and A. Elmagarmid. Privacy preserving association rule mining. In *Research Issues in Data Engineering (RIDE)*, February 2002.

[T+97] S. Thomas et al. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1997.

[TK00] P-N. Tan and V. Kumar. Interestingness measures for association patterns : A perspective. Technical report, University of Minnesota, 2000.

[TKR+95] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Hatonen, and H. Mannila. Pruning and grouping discovered association rules. In *ECML-95 Workshop*

on Statistics, Machine Learning and Knowledge Discovery in Databases, April 1995.

[Toi96] H. Toivonen. Sampling large databases for association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[TPBL00] R. Taouil, N. Pasquier, Y. Bastide, and L. Lakhal. Mining basis for association rules using closed sets. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, February 2000.

[VC02] J. Vaidya and C. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2002.

[XD99] Y. Xiao and M. H. Dunham. Considering main memory in mining association rules. In *Proc. of Intl. Conf. on Data Warehousing and Knowledge Discovery (DAWAK)*, 1999.

[Zak00] M. J. Zaki. Generating non-redundant association rules. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 2000.

[ZG01] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Rensselaer Polytechnic Institute, 2001.

[ZH02] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proc. of SIAM Intl. Conf. on Data Mining*, 2002.

[ZKM01] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 2001.

[ZPOL97a] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1997.

[ZPOL97b] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal (DMKD)*, December 1997.