PROJECT TECHNICAL REPORT

for

Sponsored Project HITA/ESE/JRH/001

Title	:	**Design and Analysis of Database Mining Algorithms**
Sponsor	:	**HITACHI Ltd., Japan**
Duration	:	**December 1997 – May 1998**
Principal Investigator	:	**Dr. Jayant Haritsa** **Database Systems Lab** **Supercomputer Education and Research Centre**

July 1998

Centre for Sponsored Schemes and Projects

Indian Institute of Science

Bangalore 560012, INDIA

SUMMARY

In December 1997, the Indian Institute of Science undertook a six-month sponsored project for Hitachi Ltd, Japan, under contract HITA/ESE/JRH/001. The project investigator was Dr. Jayant Haritsa of the Supercomputer Education & Research Centre at the Indian Institute of Science. The goal of the project was to develop efficient database mining algorithms.

Over the last six months, a variety of data mining algorithms to discover association rules have been developed by Dr. Jayant Haritsa and his students, and these algorithms are described in this report. A summary of the new algorithms is given below:

Initial Mining : There have been been many algorithms proposed in the literature for efficiently mining (for the first time) large historical databases. Virtually all of these algorithms require making multiple scans over the entire database with the number of scans required being proportional to the number of items in the biggest frequent itemset. We present here **TWOPASS**, a new algorithm that is guaranteed to provide all frequent itemsets in exactly **two** passes over the database.

Incremental Mining : For most business organizations, data mining is not a one-time operation, but a recurring process, especially if the database has been significantly updated since the previous mining exercise. Therefore, 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. We present here **DELTA**, a new algorithm for efficient incremental mining which finds the frequent itemsets of both the current and incremental databases, handles "multi-support" environments where there may be changes in the frequency thresholds between the original and the current database, and also computes the new negative border.

Rule Generation : After frequent itemsets are discovered in the first stage of mining, the second step is to derive "rules" from these frequent itemsets. We present here **RULEGEN**, a new algorithm for efficient rule derivation that is linear in the size of the biggest itemset, rather than the exponential cost incurred by previous algorithms.

Contents

Chapter 1

Problem Statement

1.1 Introduction

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

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

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

For large commercial organizations, where thousands of transactions are executed every

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

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

1.2 Database Mining

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

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

databases.

1.3 Rules

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

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

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

Formal Definition

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

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

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

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

Table 1.1: Vegetable Purchase Database

expressed in terms of their equivalent percentages.

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

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

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

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

A word of caution: The rules derived in the above example are not truly valid since the database used in the example is a "toy" database that has only ten customer records – it was provided only for illustrative purposes. For rules to be meaningful, they should be derived from large databases that have thousands of customer records, thereby indicating consistent patterns, not transient phenomena.

1.4 Rule Discovery

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

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

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

Frequent Itemset Generation

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

Strong Rule Derivation

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

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

1.5 Frequent Itemset Generation

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

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

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

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

measure only those itemsets that are potentially frequent and the rare itemsets are discarded from consideration.

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

1.6 Strong Rule Derivation

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

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

1.7 Classification and Sequence Rules

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

The classification problem involves finding rules that *partition* the data into disjoint groups. For example, the courses offered by a college may be categorized into good, average

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

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

1.8 Summary

The goal of Database Mining is to discover information from historical organizational databases that can be used to improve their business decisions. Developing efficient algorithms for mining has become an active area of research in the database community in the last few years. Although commercial data mining packages are not yet available, there are several research prototypes that have been developed. Examples are QUEST, constructed at IBM's Almaden Research Center in San Jose, California, U.S.A., and DISCOVER, available from the Hong Kong University of Science and Technology. We expect that sophisticated database mining packages will be available soon and that they will become essential software for all organizations within a few years.

1.9 Our Work

In the remainder of this report, we present algorithms for efficiently mining association rules on both fresh databases and on incremental databases.

Chapter 2

Initial Mining

2.1 Introduction

There have been been many algorithms proposed in the literature for efficiently mining (for the first time) large historical databases, especially with regard to deriving association rules (e.g. [AIS93, AS94, SON95, CHN+96, HKK97]). Virtually all of these algorithms require making multiple scans over the entire database with the number of scans required being proportional to the number of items in the biggest frequent itemset (one such algorithm was described in the previous chapter). In the remainder of this chapter, we present **TWOPASS**, a new algorithm that is guaranteed to provide all frequent itemsets in exactly **two** passes over the database.

Before we describe the TWOPASS algorithm, we wish to mention that apart from computing the set of frequent itemsets, an extremely useful auxiliary information that is usually produced out of the mining process is the *negative border* of the set of frequent itemsets [Toi96]. The negative border contains the *minimal* itemsets that are not frequent, and is used, for example, in sampling-based approaches to mining [Toi96]. This negative border idea is also used in the TWOPASS algorithm.

2.2 The TWOPASS Algorithm

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

Each partition is read one by one. For each partition, we need to find all its local frequent itemsets. This can be done efficiently as the partition is in main memory. In our implementation, we used the *Apriori* [AS94] algorithm to find the frequent itemsets in the first partition. For the remaining partitions, we use L, and its negative border N, as a set of candidate itemsets. The counts in N are set to zero. If an itemset in N becomes frequent, then a *miss* has occurred. The negative border closure of $L \cup N$ is generated and its counts over the partition are found. The partition fields of the new itemsets are set to the current partition. If the partial_support of any itemset in L becomes less than sup_{min}, it is removed from L. At the end of the partition, L will contain (atleast) all the local frequent itemsets and N will contain its negative border. Thus we don't need to calculate the negative border of L again for the next partition.

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

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

second pass. By the end of this pass, all frequent itemsets will be output.

In Figure 2.1, we show the generic algorithm, and in Figure 2.2, we show the detailed steps of the TWOPASS algorithm.

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

Figure 2.1: The Generic Algorithm

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

Figure 2.2: The TWOPASS Algorithm

2.3 Proof of Correctness

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

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

Proof:

Let $count(i,n)$, $region(i,n)$ and $partial\_support(i,n)$ denote the count field, region field and partial_support of itemset i respectively, at the end of handling the n^{th} partition during the first pass. *local_count(i,n)* is the number of occurrences of itemset i within the n^{th} partition and $part\_size(n)$ is the size of the n^{th} partition.

If an itemset i is frequent within the n^{th} partition, then either it was present in L at the beginning of the partition or not. If it was present, then,

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

Since i is frequent in this partition,

$$local\_count(i,n) \geq sup_{min} \times part\_size(n)$$

Therefore,

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

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

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

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

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

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

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

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

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

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

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

2.4 Negative Border Closure

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

2.5 Size of Partitions

One of the design issues for the TWOPASS algorithm is the number of partitions. However, the performance is not sensitive to the size of partitions except for extreme cases. Very small partitions may result in too many candidates from each partition which are not globally frequent. But these candidates will be removed from L after a few more partitions are traversed. Hence the algorithm is *relatively free from skew* as compared to the Partition algorithm [SON95] which attempted to use the partition idea for *parallel* mining. Very large partitions don't cause any performance degradation, provided they can fit in main memory. Note that the size of a partition must be atleast 1 / sup_{min} since otherwise there will not be enough tuples in each partition for the definition of sup_{min} to hold. A reasonable partition size would be about ten to twenty times this minimum size. However some performance improvement may be noticeable with very large partition sizes.

2.6 Number of Candidate Itemsets

It is easy to see that an itemset will be in L only if it is frequent in atleast one partition. Hence the number of candidates which are generated is not more than that in Partition [SON95]. However since itemsets are continually removed from L, the number of candidates is actually much less. Another advantage over Partition is due to the way we mine each partition. Here, the information gathered from previous partitions is used to reduce processing during mining each partition. Therefore, no tid-lists are maintained and no level-wise mining of individual partitions is necessary.

2.7 Incorporating Sampling

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

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

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

Chapter 3

Incremental Mining

3.1 Introduction

A common feature of most of the mining algorithms that have been previously proposed is that they are designed for use on historical databases that are being mined *for the first time*. In most business organizations, however, the historical database is "dynamic" that is, it is periodically updated with fresh data. For such environments, mining is not a one-time operation but may be a recurring process, especially if the database has been significantly updated since the previous mining exercise. One obvious way to approach the dynamic mining problem is to mine the entire database from scratch on each occasion, that is, as if it were being mined for the first time. This approach would lead to considerable wasted effort since much of the processing would be a repetition of what had already been done before. Therefore, 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. Such "incremental" mining is the focus of this chapter.

The dynamic database mining problem can be formulated as the following:

Requirement 1 : Given a previously mined database DB and a subsequent increment db to this database, find the frequent itemsets in $DB \cup db$.

The important point to note here is that itemsets that were frequent in DB may no longer be so in $DB \cup db$ and, similarly, itemsets that turn out to be frequent in $DB \cup db$ may not

have been frequent in DB, making the incremental mining to be a non-trivial process.

We also anticipate that users will also often want to identify those rules that apply *locally to just the database increment*, since comparing these rules with those that applied on the original database provides insight into the temporal variations of customer purchase behavior. This additional requirement can be formulated as follows:

Requirement 2 : Apart from finding the frequent itemsets of $DB \cup db$, also identify the frequent itemsets that are local to the increment db.

An implicit assumption in the above formulations is that the minimum frequency threshold specified by the user for the current database ($DB \cup db$) is the *same* as that used for the originally mined database (DB). However, this need not always be the case, thereby further complicating the incremental mining effort. This leads to our third requirement:

Requirement 3 : The incremental mining algorithm should be able to efficiently handle "multi-support" environments where there may be changes in the frequency thresholds between DB and $DB \cup db$.

Next, from a use-ability viewpoint, we would like the performance of the incremental mining algorithm to not be highly sensitive to the characteristics of the incremental database. In particular, we would expect that:

Requirement 4 : The performance of the incremental mining algorithm should be reasonably robust with respect to (a) the *skew* between the data distributions in the original database and in the increment database, (b) the *size* of the increment relative to the size of the original database, and (c) the *type* of the increment – addition or deletion (or a combination of these operations).

Finally, apart from computing the set of frequent itemsets, the new negative border of the set of frequent itemsets has to also be computed.

Requirement 5 : The incremental mining algorithm should efficiently compute the new negative border of the set of frequent itemsets.

Within the framework of the above requirements, our goal ideally is to design incremental mining algorithms that need to access, for computing the rules of the current database ($DB \cup db$), *only the results of the previous mining but not the corresponding database (DB) itself.* That is, in essence, we should be able to "throw away" the original database and rely only on its mined results for the current processing. As can easily be anticipated, achieving this ideal is possible only under a very few restricted circumstances, which we identify here. In the more general case, we would like the incremental mining algorithm to *minimize* the amount of reprocessing performed on the original database.

Previous Work

Although database mining research has been underway for over five years now, work in the area of incremental mining algorithms has begun only lately. The first work that we are aware of appeared in [CHNW96], where an algorithm called FUP (Fast UPdate) was proposed. The FUP algorithm is similar in structure to Apriori [AS94]. FUP operates on an iterative basis and in each iteration makes a complete scan of the current database. At the end of the k-th iteration, all the frequent itemsets of size k are derived. The mining program terminates if the current iteration does not return any new frequent itemsets. A performance evaluation of FUP showed it to perform much better than a direct application of Apriori on the entire database, thereby confirming the utility of the incremental approach.

As described above, the FUP algorithm requires k passes over the current database. Last year, more efficient *one-pass* incremental algorithms, based on the negative border concept, were proposed independently in [TBAR97] and [FAAM97] and their performance studies showed that these algorithms performed noticeably better than Apriori. They also provided arguments for why they expected their algorithms to perform better than FUP.

The above studies were a welcome first step in addressing the problem of incremental mining. However, they also have the following limitations with regard to the desired features of incremental mining algorithms listed earlier:

1. The FUP algorithm design does not meet Requirement 2 (identifying the rules local to the increment). If the user needs this information, then the increment has to be mined

separately.

2. All the algorithms assume that the frequency threshold specified by the user for the current database is the same as that for the original database. Therefore, Requirement 3 is not addressed at all.

3. The performance experiments only considered situations where DB and db were identically distributed. However, as noted earlier, in practice the increment may have considerable variation with respect to the original database. In fact, it is these temporally varying databases that are typically of most interest to users. Further, most of their experiments considered increments that were only a small percentage of the original database in size. In practice, however, increments may be of arbitrary size depending on the environment in which data is being produced. In short, Requirement 4 is not fully explored.

4. The one-pass algorithms generate *complete closures* of the negative border, and under certain situations this may result in either memory space problems or wasted computational effort due to generating too many candidate itemsets. Therefore, Requirement 4 may not always be met in this regard.

5. The one-pass algorithms compute the new negative border "from scratch" without utilizing the already available negative border of the original database. This means that Requirement 5 is violated.

Contributions

In this chapter, we present a new incremental mining algorithm called **DELTA** (Differential EvaLuation of Frequent iTemset Algorithm) for the incremental mining of association rules. DELTA has been designed to largely satisfy the above-mentioned requirements on incremental mining algorithms.

Using a synthetic database generator, we analyze the performance characteristics of DELTA on a variety of dynamic databases and compare it with that of Apriori, FUP and the one-pass algorithms. Two kinds of dynamic databases are evaluated in our experiments:

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

The above database workloads were created by *extending* the synthetic database generator described in [AS94] to generate increments of the required sizes and skews. The results of our experiments show that DELTA can provide significant improvements in execution times over the previously proposed algorithms in all these environments.

3.2 Incremental Mining

In this section, we analyze the incremental approach to mining the basket database environment for association rules. For ease of exposition, we will assume for now that the support threshold specified by the user (sup_{min}) for the current database is the same as that used in mining the original database, and that the increment is a set of insertions – later, in Section 3.5, we will return to these issues. We also assume that the list of all frequent itemsets in the original database and their associated supports are available, as also the negative border of the set of frequent itemsets. In the following discussion and in the remainder of this chapter, we use the following notation:

I	Set of items in the database
DB, db, $DB \cup db$	Original, increment, and current database
$\mid DB \mid$, $\mid db \mid$, $\mid DB \cup db \mid$	Sizes of the original, increment and current database
$count_X^{DB}$, $count_X^{db}$, $count_X^{DB \cup db}$	counts of itemset X in DB, db and $DB \cup db$
sup_X^{DB}, sup_X^{db}, $sup_X^{DB \cup db}$	supports of itemset X in DB, db and $DB \cup db$
L_k^{DB}, L_k^{db}, $L_k^{DB \cup db}$	Set of frequent k-itemsets in DB, db and $DB \cup db$
L^{DB}, L^{db}, $L^{DB \cup db}$	Set of frequent itemsets in DB, db and $DB \cup db$
N^{DB}, N^{db}, $N^{DB \cup db}$	Negative borders of L^{DB}, L^{db} and $L^{DB \cup db}$
C_k^{DB}, C_k^{db}, $C^{DB \cup db}$	Set of candidate k-itemsets in DB, db and $DB \cup db$
$ScanDB$	Set of itemsets that need scanning against DB

From the standard definition of frequent itemsets, it is straightforward to prove the following observations [TBAR97]: (1) An itemset in $DB \cup db$ can be frequent only if it is frequent in either DB or db, and (2) If an itemset is frequent in both DB and in db, then it is also frequent in $DB \cup db$. Based on these observations, we can identify three different categories of frequent itemsets in $DB \cup db$, that model different aspects of customer behavior:

Persistent : Itemsets frequent in $DB \cup db$, in DB and in db – these itemsets are always popular.

Waning : Itemsets frequent in $DB \cup db$, in DB but not in db – these itemsets are fading from the market.

Waxing : Itemsets frequent in $DB \cup db$, in db but not in DB – these itemsets are gaining in popularity.

The first two categories of frequent itemsets (Persistent and Waning) can be generated *without having to access the original database* DB by using the following strategy: Mine the increment database db using one of the "first-time" mining algorithms (e.g. Apriori) after incorporating the following modification – add each element X in L^{DB} to the set of candidate itemsets while mining db and store all the resulting counts, $count_X^{db}$. Then, calculate $count_X^{DB \cup db}$ as the sum of $count_X^{DB}$ and $count_X^{db}$. If the resulting $sup_X^{DB \cup db}$ is greater than sup_{min}, include X in $L^{DB \cup db}$.

In the above mining of the increment, note that all the optimizations developed for first-time algorithms such as "remaining tuples optimization" [AIS93], "pruning function optimization" [AIS93], etc., can be applied to minimize the processing for itemsets that are known early on to definitely not be frequent.

Generating the third category (X frequent in db but not in DB) of frequent itemsets in $DB \cup db$ causes problems, however, since the associated $count_X^{DB}$ values are not available. In this case, it becomes necessary to access the original database. Note that this access is necessary *even* in the special case where $count_X^{db}$ is so frequent that even if $count_X^{DB}$ were to be zero, $\mathcal{X}$ would be a frequent itemset in $DB \cup db$. The reason for this is twofold: First, users expect the *actual* rule supports to be provided to them, not just the information that the rule exceeds the threshold support. Second, in order to compute the *confidence* of each rule, it is necessary to know the exact support values of each frequent itemset.

From the above discussion, we conclude that in general it is not possible to fully achieve the ideal goal of completely dispensing with the original database, but that reprocessing of the original database is necessary for some of the rules. Our goal then shifts to minimizing the extent of this processing. In the following sections, we present algorithms that attempt to achieve this goal.

3.3 Previous Incremental Algorithms

In this section, we describe two representative algorithms, FUP and TBAR, from the family of incremental algorithms that have been developed over the last two years [CLK97, FAA+97, FAAM97, LC97, TBAR97].

3.3.1 The FUP Algorithm

As mentioned in the Introduction, the **FUP** (Fast UPdate) algorithm [CHNW96] is the first algorithm proposed for incremental mining of association rules. FUP operates on an iterative basis and in each iteration makes a *complete scan of the current database*, beginning with the increment and then moving on to the original database. The detailed description of the algorithm is shown in the Figure 3.1).

/* find $L_1{}^{DB \cup db}$ the set of all large 1-itemsets in $DB \cup db$ */
$C_1^{DB \cup db} = \phi$; $L_1^{DB \cup db} = \phi$; $PS = \phi$; /* PS is the prune set */
for all $t \in db$ **do** /* scan db */
for all 1-itemset $X \in t$ **do**
if $X \in L_1^{DB}$ **then** $count_X^{db}++$;
else if $X \notin C_1^{DB \cup db}$ **then** { $C_1^{DB \cup db} = C_1^{DB \cup db} \cup \{X\}$; $count_X^{db} = 1$; }
else $count_X^{db}++$;
for all $X \in L_1^{DB}$ **do** /* put winners into $L_1^{DB \cup db}$ */
if $count_X^{DB \cup db} \geq sup_{min} * \mid DB \cup db \mid$ **then** $L_1^{DB \cup db} = L_1^{DB \cup db} \cup \{X\}$;
for all $X \in C_1^{DB \cup db}$ **do** /* prune candidate sets in $C_1^{DB \cup db}$ */
if $count_X^{db} < sup_{min} * \mid db \mid$ **then** { $C_1^{DB \cup db} = C_1^{DB \cup db} - \{X\}$; $PS = PS \cup \{X\}$; }
for all $t \in DB$ **do** /* scan DB */
for all 1-itemset $X \subseteq t$ **do**
if $X \in C_1^{DB \cup db}$ **then** $count_X^{DB}++$;
if $X \in PS$ **then** remove X from t; /* Transaction t is reduced */
for all $X \in C_1^{DB \cup db}$ **do** /* put winners into $L_1^{DB \cup db}$ */
if $count_X^{DB \cup db} \geq sup_{min} * \mid DB \cup db \mid$ **then** $L_1^{DB \cup db} = L_1^{DB \cup db} \cup \{X\}$;
return $L_1^{DB \cup db}$ /* end of the 1st iteration */
/* for $k \geq 2$ repeat to find $L_k^{DB \cup db}$ until either $L_k^{DB \cup db} = \phi$ or $db = \phi$ */
$L_k^{DB \cup db} = \phi$; $C_k^{DB \cup db} = \text{AprioriGen}(L_{k-1}^{DB \cup db}) - L_k^{DB}$;
for all k-itemset $X \in L_k^{DB}$ **do** /* prune losers in L_k^{DB} */
for all $(k-1)$-itemset $Y \in L_{k-1}^{DB} - L_{k-1}^{DB \cup db}$ **do**
if $Y \subseteq X$ **then** { $L_k^{DB} = L_k^{DB} - \{X\}$; break; }
for all $t \in db$ **do** /* scan db */
for all $X \in Subset(L_k^{DB}, t)$ **do** $count_X^{db}++$;
/* Subset(X,t) returns all sets in X that are contained in t */
for all $X \in Subset(C_k^{DB \cup db}, t)$ **do** $count_X^{db}++$; /* find counts of all $X \in C_k^{DB \cup db}$ */
Reduce_db (t); /* some items in db can be removed */
for all $X \in L_k^{DB}$ **do** /* put winners from L_k^{DB} into $L_k^{DB \cup db}$ */
if $count_X^{DB \cup db} \geq sup_{min} * \mid DB \cup db \mid$ **then** $L_k^{DB \cup db} = L_k^{DB \cup db} \cup \{X\}$
for all $X \in C_k^{DB \cup db}$ **do** /* prune candidate sets in $C_k^{DB \cup db}$ */
if $count_X^{db} < sup_{min} * \mid db \mid$ **then** $C_k^{DB \cup db} = C_k^{DB \cup db} - \{\text{X}\}$;
for all $t \in DB$ **do** /* scan DB */
for all $X \in Subset(C_k^{DB \cup db}, t)$ **do** $count_X^{DB}++$;
Reduce_DB (t); } /* some items in DB can be removed */
for all $X \in C_k^{DB \cup db}$ **do** /* put winners from $C_k^{DB \cup db}$ into $L_k^{DB \cup db}$ */
if $count_X^{DB \cup db} \geq sup_{min} * \mid DB \cup db \mid$ **then** $L_k^{DB \cup db} = L_k^{DB \cup db} \cup \{X\}$
return $L_k^{DB \cup db}$ /* end of the k-th iteration */

Figure 3.1: The FUP Incremental Mining Algorithm

In this algorithm, itemsets that are frequent in $DB \cup db$ are termed "winners" while itemsets that were frequent in DB but cease to be frequent in $DB \cup db$ are termed "losers". During the first iteration, the set of all frequent 1-itemsets in $DB \cup db$ is found. This is done by first scanning db for all 1-itemsets $X \in L_1^{DB}$ and updating their counts $count_X^{DB \cup db}$. The winners among these are included in $L_1^{DB \cup db}$. During the scan of db, the counts for all candidate 1-itemsets, $C_1^{DB \cup db}$, that are not part of L_1^{DB}, are also evaluated. Among these, only those whose *local* support is greater than sup_{min} can potentially be frequent in the overall database (by virtue of Observation 1 in Section 3.2). For this restricted set of itemsets, the original database DB is scanned to derive the overall support. The winners among these are included in $L_1^{DB \cup db}$.

We now move on to describing how FUP behaves in the k-th ($k > 1$) iteration, whose goal is to compute $L_k^{DB \cup db}$. Here, some of the losers are identified even before scanning db. That is, based on the set of losers $L_{k-1}^{DB} - L_{k-1}^{DB \cup db}$ identified in the previous $(k-1)$ iteration, all itemsets in L_k^{DB} that contain these loser itemsets are removed from L_k^{DB} (since any subset of a frequent itemset must also be frequent [AIS93]). For this updated set L_k^{DB}, the increment db is scanned to evaluate the counts. During the scan of db, the counts for all candidate k-itemsets $C_k^{DB \cup db}$ that are not part of L_k^{DB} are also evaluated. This set is generated by applying the AprioriGen function [AS94] on $L_{k-1}^{DB \cup db}$. Then, just like in the first iteration, after pruning $C_k^{DB \cup db}$ to eliminate those itemsets that are not frequent locally, the original database DB is scanned to find the counts for all $X \in C_k^{DB \cup db}$. The winners among these are included in $L_k^{DB \cup db}$.

3.3.2 Database Size Reduction

FUP employs the techniques proposed for the DHP (Direct Hashing and Pruning) mining algorithm [PCY95] to reduce the size of both the original and the increment databases. In the first iteration, all items that do not have enough support in db are stored in a "prune-set" PS. During the subsequent scan of DB, all items in PS are removed from all the transactions in DB.

In the k-th ($k \geq 2$) iteration, during the scan of db, for each item purchased in transaction t, the number of sets which contain this item is calculated. If this number is smaller than k,

then I cannot possibly belong to any frequent $(k+1)$-itemset. Hence, I is removed from all the transactions in db. Further, any item in DB which does not belong to any set in L_k^{DB} or $C_k^{DB \cup db}$ will not belong to any frequent $(k+1)$-itemset. Therefore, in the scanning of DB to compute the supports of sets in $C_k^{DB \cup db}$, all items that do not belong to any set in L_k^{DB} or $C_k^{DB \cup db}$ are also removed.

3.3.3 The TBAR Algorithm

As described above, the FUP algorithm requires k passes over the current database. More efficient *one-pass* algorithms, based on the negative border concept, were proposed independently in [TBAR97] and [FAAM97]. Since these algorithms are very similar, we restrict our attention to the description given in [TBAR97] – we will use TBAR to refer to this algorithm in the sequel.

In the TBAR algorithm, first the frequent itemsets in the increment db are found. Simultaneously, the counts of itemsets in L^{DB}, and their negative border N^{DB}, are updated over db. All the frequent itemsets that are in L^{DB} and N^{DB} are added to $L^{DB \cup db}$. If $L^{DB \cup db}$ is not the same as L^{DB}, then the new negative border $N^{DB \cup db}$ is calculated, otherwise, the new negative border is the same as N^{DB}. Now, if there are itemsets in $L^{DB \cup db}$ or $N^{DB \cup db}$ whose counts are unknown, new candidates are generated by computing the *negative border closure* of $L^{DB \cup db}$. During the computation of the closure, any itemsets that are not frequent in db are omitted. The counts of the candidates are then obtained by making a pass over $DB \cup db$. Thus all itemsets in $L^{DB \cup db}$ and $N^{DB \cup db}$ are finally obtained.

While the above approach is attractive by virtue of being a one-pass algorithm, a significant drawback is that *too many candidates* may be generated in the negative border closure resulting in and increase in the overall mining time. Also, the negative borders are computed *from scratch*, by applying the AprioriGen function [AS94] repeatedly. Further, the TBAR algorithm does not address the multi-support threshold problem, where the new minimum support threshold may be different from that used over DB.

We address all the above-mentioned limitations of FUP and TBAR in our design of the DELTA algorithm. Before we describe DELTA itself, we first present techniques for incremental computation of the negative border in the following section. These techniques

are subsequently used in the DELTA algorithm.

3.4 Incremental Computation of the Negative Border

The negative border N_L of a collection of itemsets L can be defined as follows: an itemset I is present in N_L iff it is not in L but all its subsets are in L. In [TBAR97], the negative border for the current database is computed by repeated applications of the AprioriGen algorithm of [AS94]. This approach is inefficient, however, in the context of incremental mining since it essentially computes the negative border "from scratch" without using the already available L^{DB} and its negative border N^{DB}.

We present now a technique for efficiently computing the incremental negative border. Our technique is captured in the NBGen function presented in Figure 3.2, the inputs to which are L^{DB} and $MoveNL$, the set of itemsets which have moved from the negative border N^{DB} to $L^{DB \cup db}$.

```
NBGen(L^DB, MoveNL )
begin
    Candidate = φ
    for all X ∈ MoveNL
        for all Y ∈ L
            if (| X |=| Y |) and (X and Y differ in exactly 1 item) then
                c = X ∪ Y
                if c ∉ L and if all (| c | −1)-subsets of c are in L then
                Candidates = Candidates ∪ { c }
    return Candidates
end
```

Figure 3.2: Generating Negative Border Incrementally

Theorem: The NBGen function computes the new negative border $N^{DB \cup db}$.

Proof: Let I be an itemset which has to be added to $N^{DB \cup db}$, i.e. it was not in N^{DB} previously. Then, all (| I | −1)-subsets of I must now be in $L^{DB \cup db}$. Since I was not in N^{DB} previously, it means that there must be some (| I | −1)-subset of I which has just been added to $L^{DB \cup db}$. That subset must therefore be in $MoveNL$. Hence it is sufficient to consider all 1-extensions of itemsets in $MoveNL$.

Now consider some 1-extension I' of any itemset X in $MoveNL$. I' can be in $N^{DB \cup db}$ only if all its subsets are in $L^{DB \cup db}$. Hence there would be some itemset Y in $L^{DB \cup db}$ which is a $(\mid I' \mid -1)$-subset of I' and which differs from X in exactly one item. Since the NBGen function seeks all such itemsets X and Y, it is guaranteed to find all those itemsets that have to be added to N. □

3.4.1 Computing the Negative Border Closure

One method for computing the negative border *closure* is to repeatedly apply the NBGen function on $L^{DB \cup db}$. At each stage of the computation, the result obtained is added to $L^{DB \cup db}$ and the NBGen function is applied again, and this process is continued until $L^{DB \cup db}$ does not grow. This approach is taken in the TBAR incremental algorithm as well as in earlier sampling-based algorithms [Toi96].

A problem with the above approach is that the negative border closure of a set of frequent itemsets *can grow exponentially* when the itemsets in $MoveNL$, have been added to it. For example, consider the case when a significant number, say n, of 1-itemsets are in $MoveNL$. This is possible since we would expect that most 1-itemsets would, if they are not already frequent, in course of time move from the negative border to the set of frequent itemsets. In such a case, there would be 2^n itemsets in the closure due to these n 1-itemsets, which may turn out to be too large to process efficiently.

A related problem is that generating too many candidate itemsets can *slow down the pass* over the database, as has been pointed out by the performance evaluation in [AS94]. It is due to this reason that the Apriori algorithm chooses to perform multiple passes over the database, generating only the next layer of candidate itemsets in each pass, rather than generating candidate itemsets for all the layers at one go. Therefore, even if it is possible to fit the complete negative border closure in main memory, it may be better to generate *only a few layers*, especially in the initial passes. However in later passes, when the number of candidates generated is small, the complete closure may be small as well. This is especially because 1-itemsets will not be candidates in the later passes. In such circumstances, it would be feasible and efficient to generate the complete closure.

While performing incremental mining, the initial set of passes would have been executed

anyway *during the original mining*. Hence the closure would typically not be subject to the exponential explosion scenario described above. However, it is important that if the situation does happen to arise, the incremental mining algorithm should continue to perform well and not suffer drastic degradation.

3.5 The DELTA Algorithm

In this section, we present our new incremental algorithm, **DELTA** (Differential EvaLuation of Frequent iTemset Algorithm), for identifying association rules in basket databases. The detailed description of the algorithm is shown in Figure 3.3.

The DELTA algorithm operates in *three phases*. At the outset, two variables L and N are initialized to L^{DB} and N^{DB}, respectively. These variables will finally contain the itemsets of $L^{DB \cup db}$ and $N^{DB \cup db}$, respectively. In the first pass of the first phase, the counts of itemsets in L and N are updated with respect to the increment db. At the end of this pass, the itemsets in N that have now become frequent are added to both $MoveNL$ and to L. Also those itemsets in L which have become small are removed from L and added to N if all their subsets are in L.[1] Extensions of itemsets in $MoveNL$ are found using the NBClosureGen function, shown in Figure 3.4. This function takes a limit ($ClosureLmt$) and if the number of candidates generated at any layer of the closure exceeds this limit, no further layers are generated. This is meant to check the explosive growth of the closure. Interestingly, note that even if some itemsets move from N to L, it is still possible that no new candidates may be generated [TBAR97].

In the other passes of the first phase, the candidates generated in the previous pass are counted over db. The candidates are separated into two sets: the last layer of candidates generated are placed in a set called $LastNB$ while the other candidates are placed in $NBClosure$. This is done because only the candidates that become frequent in $LastNB$ can be extended to form new candidates. By maintaining those itemsets separately, the next layers of candidates can be computed more efficiently. The candidates which become frequent are added to L', a temporary copy of L, because these candidates don't have the

[1] In the context of sampling-based mining, itemsets that move to L from N are called "misses" [Toi96].

/* Initialize */
$L = L^{DB}$ $N = N^{DB}$

/* First Phase: Passes over db */
for ($k = 1$; k==1 or ($LastNB \cup NBClosure \neq \phi$ **and** $\mid ScanDB \mid < ScanLmt$); k++)
for all $t \in db$
for all $X \subseteq t$ **do**
if $k = 1$ **then**
if $X \in L \cup N$ **then** $count_X^{DB \cup db}$++;
if $k \neq 1$ **or** $sup_{min}^{DB} < sup_{min}^{DB \cup db}$ **then**
if $X \in NBClosure \cup LastNB$ **then** $count_X^{db}$++;
if $k = 1$ **then** /* First Pass */
$L' =$ getFrequent($L, sup_{min} * \mid DB \cup db \mid$);
$Small = L - L'$ $L = L - Small$ /* Negative Border from $Small$ */
$N = N \cup ExtractNB(Small, L)$;
$MoveNL =$ getFrequent($N, sup_{min} * \mid DB \cup db \mid$);
$L = L \cup MoveNL$ $L' = L' \cup MoveNL$ $N = N$ - $MoveNL$
if $sup_{min}^{DB \cup db} < sup_{min}^{DB}$ **then** /* Weaker Threshold */
$ScanDB = ScanDB \cup NBClosure \cup LastNB$
$L' = L' \cup$ getFrequent($NBClosure \cup LastNB, sup_{min} * \mid db \mid$);
$MoveNL = MoveNL \cup$ getFrequent($LastNB, sup_{min} * \mid db \mid$);
else /* Other Passes */
$L' = L' \cup$ getFrequent($NBClosure \cup LastNB, sup_{min} * \mid db \mid$);
$MoveNL = MoveNL \cup$ getFrequent($LastNB, sup_{min} * \mid db \mid$);
$Small = NBClosure \cup LastNB - L'$
$ScanDB = ScanDB \cup$ ExtractNB($Small, L'$);
$ScanDB = ScanDB \cup$ getFrequent($NBClosure \cup LastNB, sup_{min} * \mid db \mid$);
NBClosureGen($L', MoveNL, NBClosure, LastNB, ClosureLmt$);

/* Second Phase: Pass over DB */
for all $t \in DB$
for all $X \subseteq t$ **and** $X \in ScanDB$ **do** $count_X^{DB \cup db}$++;
$MoveNL =$ getFrequent($ScanDB, sup_{min} * \mid DB \cup db \mid$);
$L = L \cup MoveNL$ $N = N \cup$ ExtractNB($ScanDB, L$);

/* Third Phase: Passes over $DB \cup db$ if necessary */
while $MoveNL \neq \phi$ **do**
NBClosureGen($L, MoveNL, NBClosure, LastNB, ClosureLmt$);
if $NBClosure \cup LastNB \neq \phi$ then
for all $t \in db \cup DB$
for all $X \subseteq t$ **and** $X \in NBClosure \cup LastNB$ **do** $count_X^{DB \cup db}$++;
$L = L \cup$ getFrequent($NBClosure \cup LastNB, sup_{min} * \mid db \cup DB \mid$);
$MoveNL =$ getFrequent($LastNB, sup_{min} * \mid db \cup DB \mid$);
$Small = NBClosure \cup LastNB - L$
$N = N \cup$ExtractNB($Small, L$);

Figure 3.3: The DELTA Incremental Mining Algorithm

```
NBClosureGen( L, MoveNL, NBClosure, LastNB, ClosureLmt )
begin
        NBClosure = φ
        LastNB = φ
        L' = L /* make a duplicate of L */
        do
                NBClosure = NBClosure ∪ LastNB
                L' = L' ∪ LastNB
                MoveNL = L
        while MoveNL ≠ φand | MoveNL |< ClosureLmt
end
```

Figure 3.4: Generating the Negative Border Closure

complete counts over the entire database. They are also added to $ScanDB$, which is the set of itemsets whose counts have to be updated over DB. The itemsets which are part of the new negative border are also added to $ScanDB$.

In principle, passes over the increment db during the first phase could go on till there are no candidates left. However, we have included another termination condition by which the iteration would stop when the size of $ScanDB$ becomes too large. The reason for this is that there may cases in which many candidates turn out to be frequent in the increment but are not actually frequent overall. Such would be the case, for example, when both the increment and the support are rather small resulting in *almost every itemset from the increment being identified as frequent*. If the termination condition is activated, the remaining candidates that have been left out from $ScanDB$ are generated later and counted in the third phase described below.

In the second phase, the candidates in $ScanDB$ are counted over DB. Those that are evaluated to be frequent are added to L, whereas among the others, those that have all their subsets in L are added to N.

As mentioned above, the third phase is necessary only if the first phase was terminated due to an oversized $ScanDB$.[2] The candidates which turned out to be frequent at the end of the second phase are used as the seed to generate new candidates using the NBClosureGen function. The new candidates that are generated are counted over the entire database. The

[2]This is true for the equi-support case; for the multi-support environment, as discussed in Section 3.5.1 the third phase becomes necessary in other circumstances as well.

candidates which become frequent are again used to generate new candidates for the next pass.

3.5.1 Change in Support Threshold

Until now, we have considered incremental mining in the context of "equi-support" environments, that is, where sup_{min}, the support threshold, is the *same* for both DB and $DB \cup db$. In practice, however, it is quite possible that the support levels required by the user currently may differ from that utilized originally. We can support such "multi-support" environments too by incorporating the enhancements described below in the DELTA algorithm design.

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

Stronger Threshold : This case is handled almost exactly the same way as the equi-support case, that is, as though *the threshold had not changed.* The only difference is that more itemsets will move from L to N than the other way, as they no longer remain frequent. Hence it is unlikely that many itemsets would move from N to L, resulting in much fewer candidates.

Yet another optimization is to identify all locally frequent itemsets X such that $count_X^{db} + sup_{min}^{DB} * \mid DB \mid < sup_{min}^{DB \cup db} * \mid DB \cup db \mid$. It is easy to see that it is impossible for such sets to become frequent in $DB \cup db$ and therefore these sets can also be removed from the $ScanDB$ set.

Finally, note that if $db + sup_{min}^{DB} * DB < sup_{min}^{DB \cup db} * \mid DB \cup db \mid$, there is *no need to scan DB at all*, since there will be no Category 3 (Waxing) frequent itemsets appearing in $L^{DB \cup db}$. In this situation, we can achieve our ideal goal of "throwing away" the original database DB.

It is easy to show that for the above condition to be true, $\frac{\mid db \mid}{\mid DB \mid}$, the size of the increment relative to the original, should be less than $(\frac{1 - sup_{min}^{DB}}{1 - sup_{min}^{DB \cup db}} - 1)$. For example,

if $sup_{min}^{DB} = 0.1$ and $sup_{min}^{DB \cup db} = 0.2$, all increments that are *less than 12 percent* of the original database can be processed without accessing DB.

Weaker Threshold : This case is much more difficult to handle since the L^{DB} set now needs to be *expanded* but the identities of these additional sets cannot be deduced from the increment *db*. However, any globally frequent itemset must be in L^{DB} or in N^{DB} or must be an extension of some itemset in N^{DB}. Hence it is sufficient to find extensions of itemsets that move from N^{DB} to L^{DB}. Such extensions are identified by incorporating the code segment shown in Figure 3.5 *before* the first pass of the first phase in DELTA. The extensions are obtained again using the NBClosureGen function. Note that this function may not generate the complete closure. It generates layers of the closure as long as the number of itemsets in the current layer is less than a limit. Having the limit is especially important here because under weaker thresholds, the closure may explode exponentially. This is because there can be a large number of itemsets which move from N to L.

```
if sup_min^{DB∪db} < sup_min^{DB} then
/* Extract frequent itemsets from N */
      NewFrequent = getFrequent(N, sup_min^{DB} * | DB | );
      L = L ∪ NewFrequent
      N = N - NewFrequent
      NBClosureGen(L, NewFrequent, NBClosure, LastNB, ClosureLmt);
```

Figure 3.5: Addition to DELTA for Multi-Support Case

3.5.2 Transaction Deletion

Till now, we have considered incremental mining in the context of *insertion increments*, where new transactions are added to the database. In practice, however, it is quite possible that the update to the database may be in the form of *deletion* of transactions. We can support such decrement environments too by making minor modifications, described below, in the DELTA algorithm design (these modifications suffice for both the equi-support and the multi-support cases).

For each itemset X in L^{DB} and N^{DB}, we compute its new support by making one pass over the decrement and reducing the original count by the count in the decrement.[3] That is, $count_X^{DB \cup db} = count_X^{DB} - count_X^{db}$. All itemsets whose count is still above the minimum are retained in L^{DB}. Some itemsets from L^{DB} may move to N^{DB} and vice versa. For those itemsets that move from N^{DB} to L^{DB}, we apply the NBClosureGen function on them to get their extensions. It is easy to see that at this point the state of the system is the same *as if there were no deletions*, except that the counts of some extensions have to be found. These extensions are added to $LastNB$ and $NBClosure$, just as in the weaker threshold case, shown in Figure 3.5.

3.5.3 Advantages of DELTA

We now qualitatively outline the reasons due to which the design of DELTA appears to have several advantages over that of FUP and TBAR:

1. As described in the previous sub-section, DELTA can handle the multi-support environment whereas this issue is not addressed by both FUP and TBAR.

2. DELTA requires much fewer passes than FUP over the original database DB. Although a single pass would suffice under most conditions, just like TBAR, there could be cases when such a single pass algorithm results in too many candidates and lot of wasted effort. Under some conditions, especially when considering multi-support environments, such a single pass approach will not even be feasible due to exponential memory requirements. Therefore, DELTA judiciously chooses the appropriate number of passes required to properly handle the specific increment on which it is operating.

3. DELTA incorporates techniques to ensure that only as many candidate itemsets as can be handled efficiently are generated.

4. DELTA computes new negative borders incrementally and efficiently by utilizing information from the already existing negative border.

[3] Information about the decrement may be obtained from the log files [LC97].

Parameter	Meaning	Values
N	Number of items	1000
$\mid I \mid$	Mean size of potentially frequent itemsets	4
$\mid L \mid$	Number of potentially frequent itemsets	10
$\mid T \mid$	Mean size of the transaction	1000
$\mid DB \mid$	Number of transactions in database DB	100000
$\mid db \mid$	Number of transactions in increment db	1000, 10000, 50000, 100000
S	Skew of increment db (w.r.t. DB)	Identical, Skewed

Table 3.1: Parameter Table

5. DELTA handles deletion of transactions from the original database in both the multi-support and equi-support environments.

3.6 Performance Study

In the previous sections, we presented the FUP, TBAR and DELTA incremental mining algorithms and informally motivated as to why we expect DELTA to perform better than the other algorithms. To confirm this expectation and to quantify the improvement obtained, we conducted a set of experiments that covered a range of database and mining workloads. We also included the Apriori algorithm in our algorithm 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 algorithm. The performance metric is the *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 are described in Table 3.1. These are similar to those used in [AS94] except that the size and skew of the increment are two additional parameters. We do not describe the basic data generation process here but refer the reader to [AS94, Shri97] for the details. 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.

The above method will produce data that is *identically* distributed in both DB and db. As mentioned in the Introduction, databases often exhibit temporal trends resulting in the increment perhaps having a different distribution than the original database. That is, there may be quite a few changes between the number and identities of the frequent itemsets in DB and db. To model this effect, we modified the generator in the following manner: After DB 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, it is decided whether the itemset has to be changed or not. If change is decided, each item in the itemset is changed to its "mirror image" with a certain probability (the mirror image of an item i is the item $N - i$). After the frequent itemsets are changed in this manner, db number of transactions are produced with these changed frequent itemsets.

For our study, the specific values chosen for the various generator parameters are given in Table 3.1. The database generator was written in C++ and the experiments were conducted on UltraSparc 170E workstations running Solaris 2.5. A range of rule support threshold values between 0.75% and 6% were considered. (For the databases used in our experiments support thresholds greater than 6 percent resulted in the mining usually stopping after the very first iteration itself since there were very few frequent itemsets that satisfied this minimum frequency).

Along with varying the support thresholds, we also varied the size of the increment db from 1000 transactions to 100000 transactions. Since the original database size was always kept fixed at 100000 transactions, these increment values represented a increment-to-original ratio that ranged from 0.01 to 1.

Finally, two types of increment distributions were considered: *Identical* where both DB and db had the same itemset distribution, and *Skewed* where the distributions were noticeably different.

To help characterize the inputs completely from the mining perspective, we counted for each input workload the frequent itemsets based on the Persistent, Waxing and Waning categories described in the Introduction. In addition, we also included the following two additional categories of frequent itemsets:

New: Itemsets that are frequent only in db – these itemsets have recently become popular.

Old: Itemsets frequent only in DB – these itemsets have almost disappeared from the market.

Finally, on a related dimension, we also counted the number of frequent itemsets in $L^{DB \cup db}$, in L^{DB} and in L^{db}, respectively.

We also conducted experiments wherein the new minimum support threshold is different from that used in the original mining. The original threshold was varied from 0.6% to 6% and for each value of the original threshold, the new threshold was also varied in the same range. Therefore, both the Stronger Threshold and Weaker Threshold cases outlined in Section 3.5 were considered in these experiments.

The limit for the size of the $ScanDB$ set ($ScanLmt$) is set to 10000. If this limit is exceeded during the first phase of the DELTA algorithm, no more itemsets are added to $ScanDB$ and the second phase of the DELTA algorithm is activated. The limit used in the NBClosureGen function ($ClosureLmt$) is set to 50 in our experiments. This means that if the number of itemsets generated at any layer is more than 50, the next layer is not generated.

3.7 Results

In this section, we report on the results of our experiments comparing the performance of the various mining algorithms for the dynamic basket database model described in the previous section. We conducted two sets of experiments, one with Identical itemset distribution and the other with Skewed itemset distribution for the increment, and their results are analyzed in the remainder of this section. We first discuss these results in the context of the "equi-support" environment and subsequently for the "multi-support" environment.

3.7.1 Identical Distribution

For the Identical distribution environment, Table 3.2 gives the breakdown of the various kinds of frequent itemsets, for original-to-increment ratios of 100 : 1, 100 : 10, 100 : 50 and 100 : 100, and support thresholds of 0.75, 1, 2, 4, and 6 percent. As can be seen from this table, the number of frequent itemsets decreases exponentially with increase in

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

Table 3.2: Itemset distribution (Identical)

support threshold, as should be expected. Further, with increasing increment sizes, there is an increase in the Waxing set and a decrease in the Waning set, since the increment plays a larger role in determining the frequent itemsets. Similarly there is an increase in the Old set and a decrease in the New set, for the same reason.

The execution time performance of the Apriori, FUP and DELTA mining algorithms for the above set of input workloads is shown in Figure 3.6 (we defer the discussion of TBAR's performance to later). We first notice here that for all the increment sizes and all the support factors, FUP outperforms Apriori and *DELTA outperforms both FUP and Apriori.* At low support thresholds especially, there is a considerable difference between the performance of DELTA and that of FUP – this is because these support values result in higher values of k, the maximal frequent itemset size, leading to correspondingly more iterations for FUP over the original database DB.

We also notice that the difference between the performance of the incremental algorithms

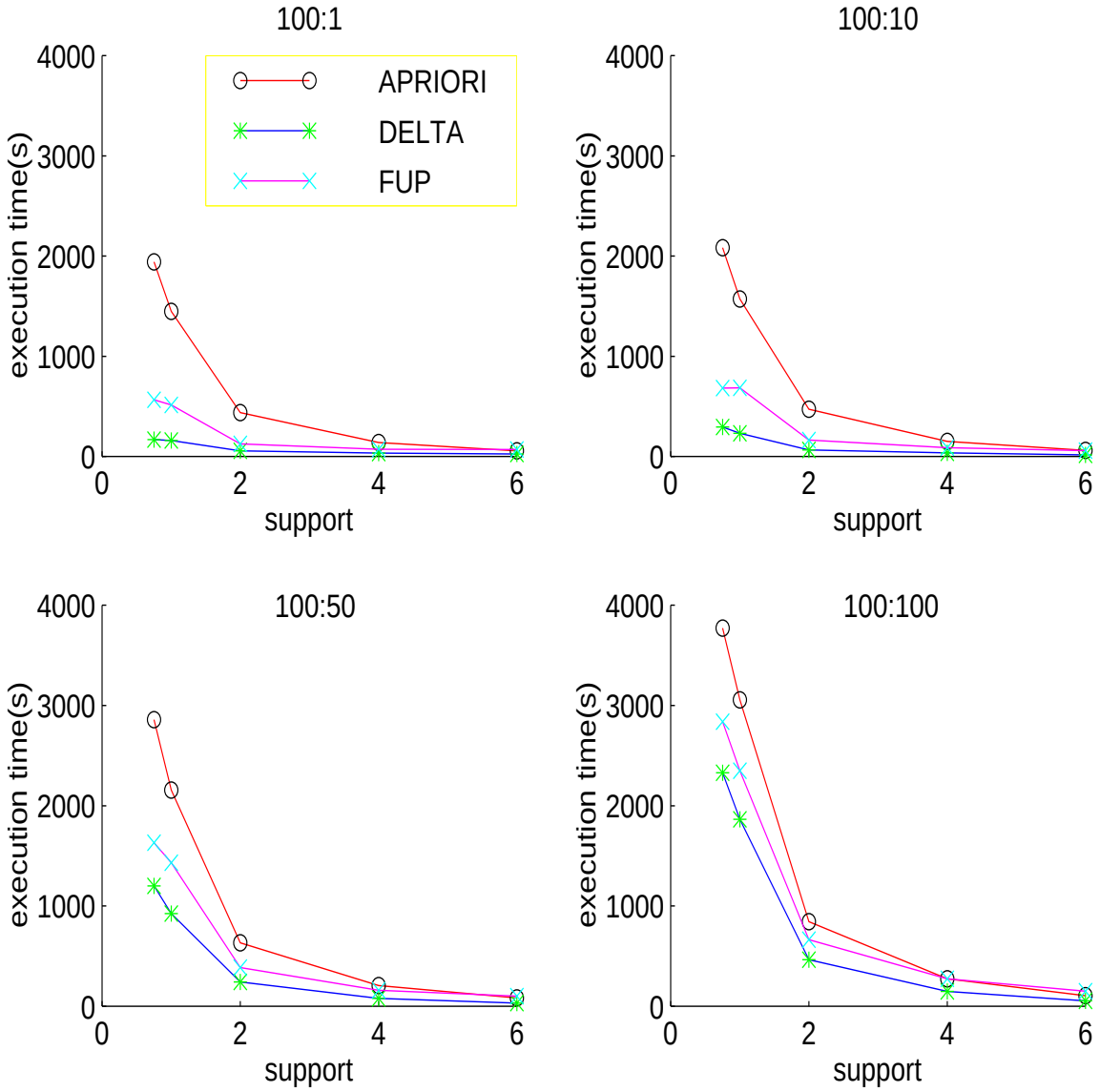

Figure 3.6: Execution time in seconds (Identical)

100:1		
Support	DELTA	TBAR
0.75	11.6	8.5
1.0	9.1	5.1
2.0	7.7	5.2

100:10		
Support	DELTA	TBAR
0.75	7.1	3
1.0	2.6	2.6
2.0	2.6	2.6

Table 3.3: DELTA vs TBAR

(FUP and DELTA) and that of Apriori decreases with increasing increment size. This is because all three algorithms have to do essentially similar multiple iterations over the increment *db* and the more the size of *db*, the lesser will the effect of optimization on *DB* processing be felt on the overall performance. That is, the mining of *db* becomes a major factor in the overall performance determination. But what is interesting to note is that even when the increment is as large as the original database (the 100:100 case), FUP and DELTA still do noticeably better than Apriori.

We have seen in the above that DELTA can perform significantly better than FUP. We now move on to comparing DELTA's performance to that of TBAR based on their published results [TBAR97]. In Table 3.3 we show the relative speedups of DELTA and TBAR with respect to that of Apriori for the 100:1 and 100:10 increment databases. We see here that DELTA performs noticeably better than TBAR, especially for small-sized increments and low supports. Further, its performance never becomes worse than that of TBAR.

3.7.2 Skewed Distribution

For the Skewed environment, Table 3.4 gives the breakdown of the various kinds of frequent itemsets for the same increment and support values as those used for the previous experiment. Note that there are many more Old and New category itemsets with the Skewed distribution as compared to that for the Identical distribution. For example, with an increment size 100:100 and support of 0.75%, the New and Old values for Skewed are 117 and 219, respectively, whereas for Identical they were 25 and 26, respectively. This is only to be expected since having a skew implies significant change in the frequent itemset elements between the original and the increment. The same effect is also observed by comparing the

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

Table 3.4: Itemset distribution (Skewed)

number of Persistent itemsets between the Identical and Skewed environments – at the 0.75 percent support threshold, for example, the Skewed setup always has about 200 itemsets less.

The execution time performance of the Apriori, FUP and DELTA algorithms for the Skewed workload is shown in Figure 3.7. In an overall sense, their relative behavior is very similar to that seen for the Identical distribution – again for all the increment sizes and all the support factors, FUP outperforms Apriori and DELTA outperforms both FUP and Apriori. So, while the previous experiment had shown the robustness of DELTA's improved performance with regard to increment size, this experiment confirms the robustness of DELTA's improved performance with regard to increment distribution.

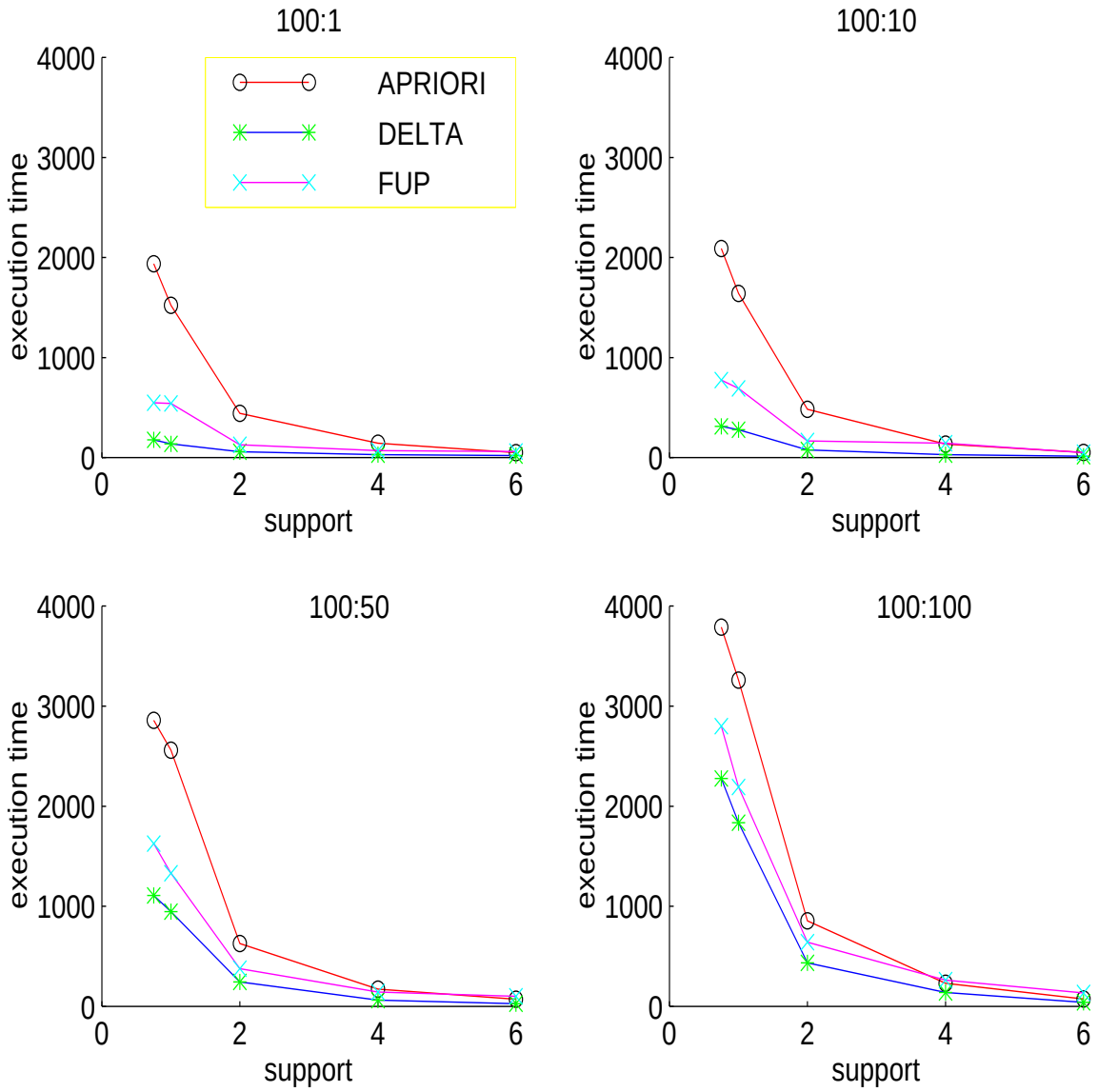

Figure 3.7: Execution time in seconds (Skewed)

3.7.3 Multi-Support Experiments

The previous set of experiments modeled equi-support environments. We now move on to considering *multi-support* environments. In these experiments, we compare the performance of DELTA to Apriori only since FUP and TBAR, as mentioned before, do not handle the multi-support case. Although we conducted these experiments for a variety of increments and support changes, due to space constraints we show the performance graphs here for only a sample set of increments and change in support values (the performance behavior in the other experiments was similar). In particular we consider two cases, one where the original support threshold is high and the other where the original support threshold is low.

High Original Support

In this experiment, we fixed the initial support to be 4.0% and the new support was varied between 1.0% and 7.0%. thereby covering both the Weaker Threshold and Stronger Threshold possibilities. For this environment, Figures 3.8a and 3.8b show the performance of DELTA relative to that of Apriori for the 100:1 and 100:50 increment ratios, respectively, where the distribution of the increments is Identical to that of the original.

We see in Figure 3.8a that DELTA exhibits a huge performance gain in the Stronger Threshold region ($\geq$ 4.0%), upto *as much as 25 times*. Further, in the Weaker Threshold region, although DELTA's performance drops with decreasing threshold, the improvement relative to Apriori is still observable. For example, for a new support of 1.0% percent, DELTA takes about *25%* less time than Apriori. In Figure 3.8b, a similar qualitative behavior is seen but the magnitude of improvement is somewhat reduced. This is expected since both DELTA and Apriori have to execute passes over a larger increment resulting in more commonality in the work that is performed.

Low Original Support

In our second experiment, we fixed the initial support to be 0.8% and the new support was varied between 0.6 and 1.0. For this environment, Figures 3.9a and 3.9b show the performance of DELTA relative to that of Apriori for the same databases as those used in

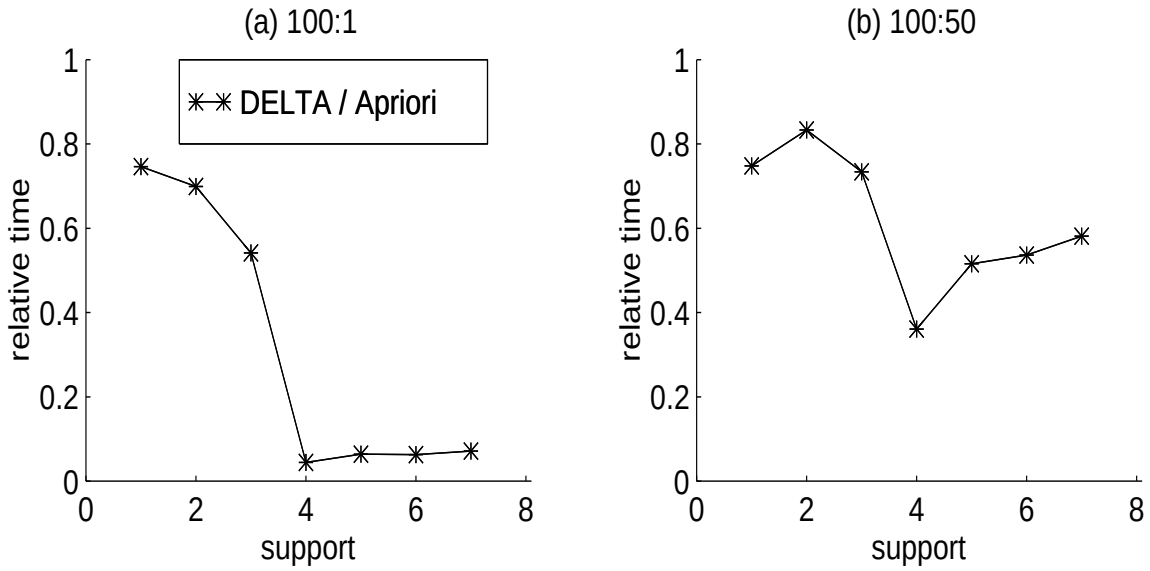

Figure 3.8: Multi-Support Experiment (High Original Support)

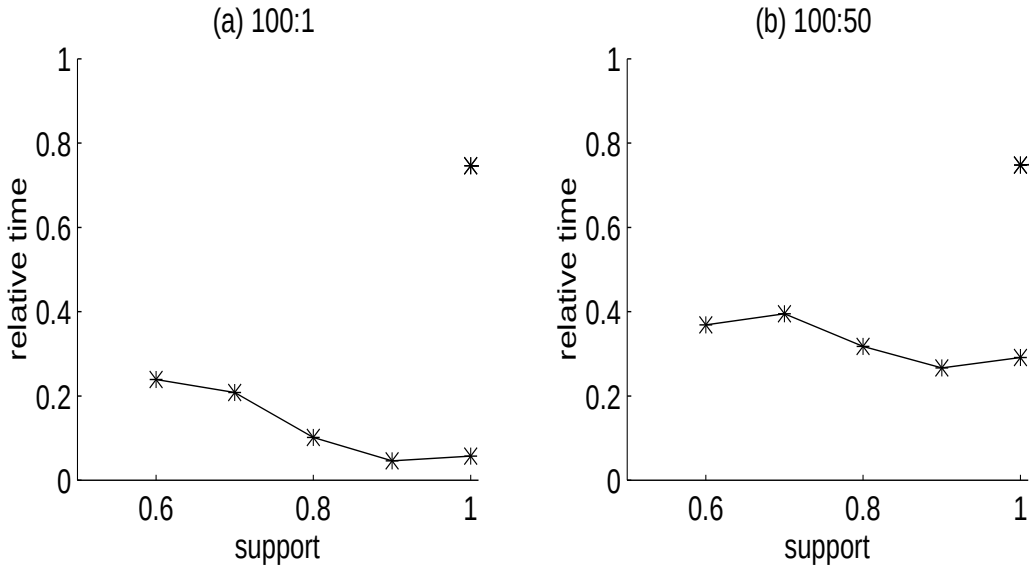

Figure 3.9: Multi-Support Experiment (Low Original Support)

the High Support experiment,

We see in Figure 3.9a that DELTA exhibits large performance gains in the Stronger Threshold region ($\geq 0.8\%$), again upto as much as 25 times. Further, in the Weaker Threshold region, although DELTA's performance drops with decreasing threshold, the improvement relative to Apriori is still quite significant. For example, for a new support of 0.6 % percent, DELTA takes *one-fifth* the time taken by Apriori. A similar good performance is seen in Figure 3.9b but with reduced magnitude – the reduction is for the same reason as explained in the High Support experiment.

We now comment on why the performance advantage of DELTA increased in the Low Support case as compared to the High Support case. The reason is the following: In the low support case, much work has already been done in the previous mining and so DELTA performs better. This is confirmed by the fact that DELTA and Apriori converge in performance when the original threshold is high, say 6% and the present threshold is low such as 1%. This is expected since very few itemsets become frequent at 6% and DELTA and Apriori have to do almost the same amount of work. It is to be noted that under such circumstances, a large number of 1-itemsets from the negative border become frequent, 350 in this case. It is impractical to generate the closure of all these itemsets as that would be 2^{350} itemsets, and this problem is avoided by DELTA by virtue of the limits incorporated in its design.

3.8 Conclusions

We considered the problem of incrementally mining association rules on 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 try to use the previous mining results, that is, knowledge of the itemsets which are frequent in the original database, their negative border, and their associated counts, to identify the frequent itemsets in the updated database. A list of desirable features for incremental mining algorithms was outlined and a new incremental mining algorithm called DELTA, which largely achieves these requirements, was presented.

DELTA requires much fewer passes over the original database in contrast to previously

proposed algorithms such as FUP [CHNW96] that require as many passes as the number of elements in the biggest rule in the current database. In addition, DELTA handles situations where the rule frequency threshold required for the current database is different than that for the original database, and handles this for both insertion and deletion increments. Unlike the previously proposed one-pass TBAR algorithm, DELTA is designed to efficiently computation of negative borders and also to generate only as many candidates as can be handled efficiently.

The performance of DELTA was compared against that of Apriori and FUP using a synthetic database generator. Our experiments showed that for a variety of increment sizes, increment distributions, and support thresholds, DELTA performs much better than both Apriori and FUP. Further, a comparison with previously published results show DELTA to perform visibly better than TBAR.

DELTA represents a first attempt at a "grand unified" incremental algorithm, as it incorporates the positive features present in previously proposed incremental algorithms and, further, handles all kinds of increments and support thresholds. In summary, DELTA is a practical, versatile, robust and efficient incremental mining algorithm.

In our future work, we plan to study the growth of the negative border closure and find bounds on its size. We also plan to study how the speed of a pass varies with respect to the number of candidates to be counted. This will enable us to find a cost-benefit approach to deciding on how many candidates to generate for a pass and thereby automatically set the limit factors used in the DELTA algorithm.

Chapter 4

Rule Generation

4.1 Introduction

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

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

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

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

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

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

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

To address the above issue, we present the **RULEGEN** algorithm in Figure 4.1 – this algorithm efficiently finds all *minimal* rules, given a set of frequent itemsets, L. The RULEGEN algorithm closely resembles the *Apriori* algorithm in its structure.

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

Figure 4.1: The RULEGEN Algorithm

Chapter 5

Conclusions

In this project, we have attempted to design new algorithms for efficiently mining huge historical databases. In particular, we have presented three new algorithms, **TWOPASS**, **DELTA** and **RULEGEN**, which together address both the frequent itemset generation and the rule generation problems for both fresh databases and incremental databases.

We recommend the above algorithms to the scientists at Hitachi since they could considerably reduce the tremendous computational expense normally associated with mining operations.

Bibliography

[AIS93] R. Agrawal, T. Imielinski and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *Proc. of 22nd SIGMOD Conf.*, May 1993.

[AS94] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proc. of 20th VLDB Conf.*, September 1994.

[CHN+96] D. Cheung, J. Han, V. Ng, A. Fu and Y. Fu, "A Fast Distributed Algorithm for Mining Association Rules", *Proc. of 4th PDIS Conf.*, December 1996.

[CHNW96] D. Cheung, J. Han, V. Ng and C. Wong, "Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique", *Proc. of 12th ICDE Conf.*, February 1996.

[CLK97] D. Cheung, S. Lee and B. Kao, "A General Incremental Technique for Maintaining Discovered Association Rules" *Proc. of 5th DASFAA Conf.*, April 1997.

[FAA+97] R. Feldman, A. Amir, Y. Aumann, A. Zilberstein and H. Hirsh, "Incremental Algorithms for Association Generation", *Proc. of 1st PAKDD Conf.*, February 1997.

[FAAM97] R. Feldman, Y. Aumann, A. Amir and H. Mannila, "Efficient Algorithms for Discovering Frequent Sets in Incremental Databases", *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[HKK97] E. Han, G. Karypis and V. Kumar, "Scalable Parallel Data Mining for Association Rules", *Proc. of 26th SIGMOD Conf.*, June 1997.

[LC97] S. Lee and D. Cheung, "Maintenance of Discovered Association Rules: When to Update?", *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[PCY95] J. Park, M. Chen and P. Yu, "An Effective Hash-Based Algorithm for Mining Association Rules", *Proc. of 24th SIGMOD Conf.*, May 1995.

[TBAR97] S. Thomas, S. Bodagala, K. Alsabti and S. Ranka, "An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases", *Proc. of 3rd KDD Conf.*, August 1997.

[Shri97] Shrividya, "DELTA: A Fast Algorithm for Incremental Mining of Association Rules", *M.E. Thesis*, Dept. of Computer Science & Automation, Indian Institute of Science, June 1997.

[SON95] A. Savasere, E. Omiecinski and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases", *Proc. of 21st VLDB Conf.*, September 1995.

[Toi96] H. Toivonen, "Sampling Large Databases for Association Rules", *Proc. of 22nd VLDB Conf.*, September 1996.