Pergamon

Information Systems Vol. 25, No. 5, pp. 323–343, 2000
© 2000 Elsevier Science Ltd. All rights reserved
Printed in Great Britain
0306-4379/00 $20.00

PII: S0306-4379(00)00021-1

QUANTIFYING THE UTILITY OF THE PAST IN MINING LARGE DATABASES†

VIKRAM PUDI[1] and JAYANT R. HARITSA[1,2]

[1]Database Systems Lab, SERC, Indian Institute of Science, Bangalore 560012, INDIA
[2]Database Systems Research Department, Lucent Bell Labs, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

(Received 6 February 2000; in final revised form 8 April 2000)

Abstract — Incremental mining algorithms that can efficiently derive the current mining output by utilizing previous mining results are attractive to business organizations since data mining is typically a resource-intensive recurring activity. In this paper, we present the **DELTA** algorithm for the robust and efficient incremental mining of association rules on large market basket databases. DELTA guarantees efficiency by ensuring that, for any dataset, at most three passes over the increment and one pass over the previous database are required to generate the desired rules. Further, it handles "multi-support" environments where the support requirements for the current mining differ from those used in the previous mining, a feature in tune with the exploratory nature of the mining process. We present a performance evaluation of DELTA on large databases over a range of increment sizes and data distributions, as well as change in support requirements. The experimental results show that DELTA can provide significant improvements in execution times over previously proposed incremental algorithms in all these environments. In fact, for many workloads, its performance is close to that achieved by an optimal, but practically infeasible, algorithm. © 2000 Elsevier Science Ltd. All rights reserved

Key words: Data Mining, Association Rule, Hierarchical Association Rule, Incremental Mining

1. INTRODUCTION

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

In the above context, it is attractive to consider the possibility of using the results of the previous mining operations to minimize the amount of work done during each new mining operation. That is, given a previously mined database DB and a subsequent increment db to this database, to efficiently mine db and $DB \cup db$. Mining db is necessary to evaluate the effects of business strategies; whereas mining $DB \cup db$ is necessary to maintain the updated set of mining rules. Such "incremental" mining is the focus of this paper. Practical applications where incremental mining techniques are especially useful include data warehouses and Web mining since these systems are constantly updated with fresh data – on the web, for instance, about one million pages are added daily [12].

We consider here the design of incremental mining algorithms for databases that can be represented as a two-dimensional matrix, with one dimension being a (fixed) set of possibilities, and the other dimension being a dynamic history of instances, with each instance recording a joint occurrence of a subset of these possibilities. A canonical example of such a matrix is the "market basket" database [1], wherein the possibilities are the items that are on sale by the store, and the instances record the specific set of items bought by each customer. For ease of exposition, we will assume a market basket database in the remainder of this paper.

Within the above framework, we focus in particular on the identification of *association rules* [1], that is, rules which establish interesting correlations about the joint occurrence of items in customer

†Recommended by Felipe Carino

purchases – for example, "Eighty percent of the customers who bought milk also bought cereal". Association rules have been productively used in a variety of applications ranging from bookstores (e.g. when you purchase a book at Amazon.com, the site informs you of the other books that have been typically bought by earlier purchasers of your book) to health care (e.g. in detecting shifts in infection and antimicrobial resistance patterns in intensive care units [7]).

Since the association rule problem is well-established in the database research literature (see [1, 19] for complete details), we assume hereon that the reader is familiar with the concepts and algorithmic techniques underlying these rules. In particular, we assume complete knowledge of the classical *Apriori* algorithm [3].

1.1. The State-of-the-Art

The design of incremental mining algorithms for association rules has been considered earlier in [5, 8, 9, 10, 11, 18]. While these studies were a welcome first step in addressing the problem of incremental mining, they also suffer from a variety of limitations that make their design and evaluation unsatisfactory from an "industrial-strength" perspective:

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

As we will show later in this paper, the performance of the algorithms presented in [11, 18] is sensitive to the skew factor. In fact, their sensitivity is to the extent that, with significant skew and substantial increments, they may do worse than even the naive approach of completely ignoring the previous mining results and applying Apriori from scratch on the entire current database.

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

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

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

Changing User Requirements It is implicitly assumed that the minimum support specified by the user for the current database ($DB \cup db$) is the *same* as that used for the previously mined database (DB). However, in practice, given mining's exploratory nature, we could expect user requirements to change with time, perhaps resulting in different minimum support levels across mining operations. Extending the algorithms to efficiently handle such "multi-support" environments is not straightforward.

Hierarchical Databases Virtually all the previous work is applicable only to "flat" databases where the items in the database have independent characteristics. In reality, however, databases are often "hierarchical", that is, there exists a *is-a hierarchy* over the set of items

in the database[†]. For example, sweaters and ski jackets are both instances of winter wear. As pointed out in [16], such taxonomies are valuable since rules at lower levels may not satisfy the minimum support threshold and hence not many rules are likely to be output if they are not considered. They are also useful to prune away many of the uninteresting rules that would be output at low minimum support thresholds, since a significant fraction of the rules at lower levels may be subsumed by those at higher levels.

1.2. Contributions

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

- DELTA guarantees that, for the entire mining process, at most *three passes* over the increment and *one pass* over the previous database may be necessary. We expect that such bounds will be useful to businesses for the proper scheduling of their mining operations.
- For the special case where the new results are a *subset* of the old results, and therefore in principle requiring no processing over the previous database, DELTA is optimal in that it requires only a *single pass* over the increment to complete the mining process.
- For computing the negative border [19] closure, a major performance-determining factor in the incremental mining process, a new hybrid scheme that combines the features of earlier approaches is implemented.
- DELTA provides complete mining results for both the entire current database as well as solely the increment.
- DELTA can handle multi-support environments, requiring only *one* additional pass over the current database to achieve this functionality.
- By carefully integrating optimizations previously proposed for first-time hierarchical mining algorithms, the DELTA design has been extended to efficiently handle incremental mining of hierarchical association rules.
- The performance of DELTA is evaluated on a variety of dynamic databases and compared with that of Apriori and the previously proposed incremental mining algorithms for flat association rules. For hierarchical association rules, we compare DELTA against the Cumulate first-time mining algorithm presented in [16]. All experiments are made on databases that are significantly larger than the entire main memory of the machine on which the experiments were conducted. The effects of database skew are also modeled.

  The results of our experiments show that DELTA can provide significant improvements in execution times over the previous algorithms in all these environments. Further, DELTA's performance is comparatively robust with respect to database skew.
- We also include in our evaluation suite the performance of an *an oracle* that has *complete apriori* knowledge of the identities of all the large itemsets (and their associated negative border) both in the current database as well as in the increment and only requires to find their respective counts. It is easy to see that this algorithm, although practically infeasible, represents the optimal since all other algorithms will have to do at least the same amount of work. Therefore, modeling the oracle's performance permits us to characterize the efficiency of practical algorithms in terms of their distance from the optimal.

[†]Flat and hierarchical databases are also referred to as "boolean" and "generalized" databases, respectively, in the mining literature.

DB, db, $DB \cup db$	Previous, increment, and current database
sup_{min}^{DB}, $sup_{min}^{DB \cup db}$	Previous and New Minimum Support Thresholds
sup_{min}	Minimum Support Threshold when $sup_{min}^{DB} = sup_{min}^{DB \cup db}$
L^{DB}, L^{db}, $L^{DB \cup db}$	Set of large itemsets in DB, db and $DB \cup db$
N^{DB}, N^{db}, $N^{DB \cup db}$	Negative borders of L^{DB}, L^{db} and $L^{DB \cup db}$
L	$L^{DB \cup db} \cap (L^{DB} \cup N^{DB})$
N	Negative border of L
Small	Set of small itemsets with known counts (during program execution)

Table 1: Notation

Our experiments show that DELTA's efficiency is *close to that obtained by the oracle* for many of the workloads considered in our study. This shows that DELTA is able to extract *most of the potential* for using the previous results in the incremental mining process.

1.3. Organization

The remainder of this paper is organized as follows: The DELTA algorithm for both flat and hierarchical association rules is presented in Section 2 for the equi-support environment. The algorithm is extended to handle the multi-support case in Section 3. The previously proposed incremental mining algorithms are summarized in Section 4. The performance model is described in Section 5 and the results of the experiments are highlighted in Section 6. Finally, in Section 7, we present the conclusions of our study and outline future research avenues.

2. THE DELTA ALGORITHM

In this section, we present the design of the DELTA algorithm. For ease of exposition, we first consider the "equi-support" case, and then in Section 3, we describe the extensions required to handle the "multi-support" environment. In the following discussion and in the remainder of this paper, we use the notation given in Table 1. Also, we use the terms "large", "small", "count" and "support" with respect to the entire database $DB \cup db$, unless otherwise mentioned.

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

2.1. The Mechanics of DELTA

The pseudo-code of the core DELTA algorithm for generating flat association rules is shown in Figure 1 – the extension to hierarchical association rules is presented in Section 2.2. At most *three* passes over the increment and *one* pass over the previous database are made, and we explain below the steps taken in each of these passes. After this explanation of the mechanics of the algorithm, we discuss in Section 2.3 the rationale behind the design choices.

2.1.1. First Pass over the Increment

In the first pass, the counts of itemsets in L^{DB} and N^{DB} are updated over the increment db, using the function UpdateCounts (line 1 in Figure 1). By this, some itemsets in N^{DB} may become large and some itemsets in L^{DB} may become small. Let the resultant set of large itemsets be L. These large itemsets are extracted using the function GetLarge (line 2). The remaining itemsets are put in *Small* (line 3), and are later used for pruning candidates. The algorithm terminates if no itemsets have moved from N^{DB} to L (lines 4–5). This is valid due to the following Theorem presented in [18]:

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

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

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

24.   ScanDB = GetLarge(C ∪ N^u, sup_min * |db|);
25.   N' = NegBorder(L ∪ ScanDB) − Small;
26.   get supports of itemsets in N' from (C ∪ N^u)
27.   UpdateCounts(DB, N' ∪ ScanDB);      // first (and only) pass over DB
28.   L^{DB∪db} = L ∪ GetLarge(ScanDB, sup_min * |DB ∪ db|);
29.   N^{DB∪db} = NegBorder(L^{DB∪db});
30.   get supports of N^{DB∪db} from (Small ∪ N')
31.   return(L^{DB∪db}, N^{DB∪db});
end
```

Fig. 1: The DELTA Incremental Mining Algorithm

Theorem 1 *If X is an itemset that is not in L^{DB} but is in $L^{DB\cup db}$, then there must be some subset x of X which was in N^{DB} and is now in $L^{DB\cup db}$.*

Hence, for the special case where the new results are a *subset* of the old results, and therefore in principle requiring no processing over the previous database, DELTA is optimal in that it requires only a single pass over the increment to complete the mining process.

2.1.2. Second Pass over the Increment

On the other hand, if some itemsets do move from N^{DB} to L, then the negative border N of L is computed (line 6), using the AprioriGen [3] function. Itemsets in N with unknown counts are stored in a set N^u (line 10). The remaining itemsets in N i.e. with known counts, are all small.

Therefore, the only itemsets that may be large (and are not yet known to be so) are those in N^u and their extensions. If there are no itemsets in N^u, the algorithm terminates (lines 7-9).

Now, any itemset in N^u that is not locally large in *db* cannot be large in $DB \cup db$. Further, none of its extensions can be large as well. This is based on the following observation of [8]†:

Theorem 2 *An itemset can be present in $L^{DB \cup db}$ only if it is present in either L^{DB} or L^{db} (or both).*

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

2.1.3. Third (and Final) Pass over the Increment

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

2.1.4. First (and Only) Pass over the Previous Database

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

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

2.1.5. Results for the Increment

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

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

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

†This observation applies only to the equi-support case.

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

2.2. Generating Hierarchical Association Rules

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

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

A set of optimizations to improve upon the above scheme were introduced in [16] as part of the Cumulate (first-time) hierarchical mining algorithm. Interestingly, we have found that these optimizations can be utilized for incremental mining as well, and in particular, can be cleanly integrated in the core DELTA algorithm. In the remainder of this sub-section, we describe the optimizations and their incorporation in DELTA.

2.2.1. Cumulate Optimizations

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

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

2.2.2. Incorporation in DELTA

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

1. The first optimization is performed only in routines that access the database and therefore do not affect the structure of the DELTA algorithm.
2. The second optimization is performed before each pass over the increment or previous database. Ancestors of items that are not part of any candidate are removed from the arrays of ancestors that were precomputed during the first optimization.
3. The third optimization is performed only once and that is at the end of the first pass over the increment. At this stage the identities of all potentially large 2-itemsets (over $DB \cup db$) are known, and hence no further candidate 2-itemsets will be generated. Among the potentially large 2-itemsets, those that contain an item and its ancestor are pruned. It follows that candidates generated from the remaining 2-itemsets will also have the same property, i.e.

they will not contain an item and its ancestor. Hence this optimization does not need to be applied again.

2.3. Rationale for the DELTA Design

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

Let L be the set of large itemsets in $L^{DB} \cup N^{DB}$ that survive the support requirement after their counts have been updated over *db*, and N be its negative border. Now, if the counts of all the itemsets in N are available, then the final output is simply $L \cup N$. Otherwise, the only itemsets that may be large (and are not yet known to be so) are those in N with unknown counts and their extensions – by virtue of Theorem 1. At this juncture, we can choose to do one of the following:

Complete Closure Generate the complete closure of the negative border, that is, all extensions of the itemsets in N with unknown counts. While generating the extensions, itemsets that are known to be small may be removed so that none of their extensions are generated. After the generation process is over, find the counts of all the generated itemsets by performing one scan over $DB \cup db$. We now have all the information necessary to first identify $L^{DB \cup db}$, and then the associated $N^{DB \cup db}$.

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

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

The first scheme, Complete Closure, appears infeasible because it could generate *a very large number of candidates* if the so-called "promoted borders" [11], that is, itemsets that were in N^{DB} but have now moved to $L^{DB \cup db}$, contain more than a few 1-itemsets. This is because if p_1 is the number of 1-itemsets in the promoted borders, a lower bound on the number of candidates is $2^{p_1}(|L| - p_1)$. This arises out of the fact that every combination of the p_1 1-itemsets is a possible extension, and all of them can combine with any other large itemset in L to form candidates. Therefore, even for moderate values of p_1, the number of candidates generated could be extremely large.

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

Therefore, in DELTA, we adopt the third hybrid strategy, wherein an initial Layered Closure approach is followed by a Complete Closure strategy. In particular, the Layered Closure is used only for the *first* layer, and then the Complete Closure is brought into play. This choice is based on the well-known observation that pruning typically has the maximum impact for itemsets of length two – that is, the number of 2-itemsets that turn out to be large is usually a small fraction of the possible 2-itemset candidates [14]. In contrast, the impact of pruning at higher itemset lengths is comparatively small.

To put it in a nutshell, the DELTA design endeavors to achieve a reasonable compromise between the number of candidates counted and the number of database passes, since these two factors represent the primary bottle-necks in association rule generation. That our choice of compromise results in good performance is validated in the experimental study described in Section 6.

3. MULTI-SUPPORT INCREMENTAL MINING IN DELTA

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

For convenience, we break up the multi-support problem into two cases: *Stronger*, where the current threshold is higher (i.e., $sup_{min}^{DB \cup db} > sup_{min}^{DB}$), 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:

3.1. Stronger Support Threshold

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

Initially, all itemsets which are not large w.r.t. $sup_{min}^{DB \cup db}$ are removed from L^{DB} and the corresponding negative border is then calculated. The itemsets that are removed are not discarded completely, but are retained separately since they may become large after counting over the increment *db*. They may also be part of the computed negative border closure (lines 15-20 in Figure 1). If so, then during the pass over DB their counts are not measured since they are already known. If the counts of all the itemsets in the closure are known, *the pass over DB becomes unnecessary.*

3.2. Weaker Support Threshold

The weaker support case is much more difficult to handle since the L^{DB} set now needs to be *expanded* but the identities of these additional sets cannot be deduced from the increment *db*. In particular, note that Theorem 2, which DELTA relied on for pruning candidates in the equi-support case, *no longer holds* when the support threshold is lowered since we cannot deduce that a candidate is small over DB just because it is not present in $L^{DB} \cup N^{DB}$.

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

In DELTA, the above strategy is incorporated by generating extra candidates (as described below) based on the lowered support threshold. It is only for these candidates that Theorem 2 does not hold. Hence, it is necessary to find their counts over the entire database $DB \cup db$. This is done *simultaneously* while executing equi-support DELTA.

The pseudo-code for the complete algorithm is given as function **DeltaLow** in Figure 2, and is described in the remainder of this section. The important point to note here is that the enhanced DELTA requires only *one* additional pass over the entire database to produce the desired results.

3.2.1. First Pass over the Increment

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

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

Fig. 2: DELTA for Weaker Support Threshold (DeltaLow)

3.2.2. Remaining Passes of Equi-Support DELTA

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

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

3.2.3. Additional Pass over the Entire Database

At the end of the above passes, the counts of all 1-itemsets and 2-itemsets of $L^{DB \cup db} \cup N^{DB \cup db}$ are available. The counts of 1-itemsets are available because $L^{DB} \cup N^{DB}$ contains all possible 1-itemsets [18], while the counts of all required 2-itemsets are available because L contains all large 1-itemsets in $DB \cup db$ and $NBetween$ contains the immediate extensions of L that are not already in $(L^{DB} \cup N^{DB})$. Therefore, it becomes possible to generate the negative border closure of all known large itemsets without encountering the "candidate explosion" problem described for the Complete Closure approach in Section 2.3.

Let L' be the set of all large itemsets whose counts are known (line 5), and let $Small$ be the set of itemsets with known counts which are not in L' (line 6). If the counts of the negative border

of L' are already known, then the algorithm terminates (lines 7–9). Otherwise, all the remaining extensions of L' that could become large are determined by computing the negative border closure (lines 10–16). (As in the equi-support case, we expect that the remaining layers of the closure can be generated together since the number of 2-itemsets in L' is typically much smaller than the overall number of all possible 2-itemset pairs.) The itemsets of the closure are counted over the entire database (line 17), and the final set of large itemsets and its negative border are determined (lines 18–20).

4. PREVIOUS INCREMENTAL ALGORITHMS

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

4.1. The FUP Algorithm

The **FUP** (Fast UPdate) algorithm [8, 9, 10] represents the first work in the area of incremental mining. It operates on an iterative basis and in each iteration makes a *complete scan of the current database.* In each scan, the *increment is processed first* and the results obtained are used to guide the mining of the original database DB. An important point to note about the FUP algorithm is that it requires k- passes over the entire database, where k is the cardinality of the longest large itemset. Further, it does not generate the mining results for solely the increment.

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

After FUP, algorithms that utilized the *negative border* information were proposed independently in [11] and [18] with the goal of achieving more efficiency in the incremental mining process. In the sequel, we will use Borders to refer to the algorithm in [11], and TBAR to refer to the algorithm in [18]. Since these algorithms are based on the negative border concept, they will be described in terms of the DELTA design.

4.2. The Borders Algorithm

The original **Borders** algorithm differs from DELTA in that it computes the entire negative border closure at one shot, that is, it uses the Complete Closure option, which could potentially result in the candidate explosion problem mentioned in Section 2.3,

A new version of the Borders algorithm was recently proposed in [5]. This version goes to the other extreme of the closure computation, adopting a Layered Closure approach. As mentioned in Section 2.3, this strategy could result in significantly increasing the number of database passes, and may therefore be problematic for large databases.

A variant of the new algorithm was proposed to handle multi-support mining. The applicability of this algorithm, however, is limited to the very special case of *zero-size* increments, that is, where the database has not changed at all between the previous and the current mining.

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

4.3. The TBAR Algorithm

The **TBAR** algorithm differs from DELTA in two major respects: First, it initially *completely mines* the increment db, that is, $L^{db} \cup N^{db}$ is computed by applying the Apriori algorithm on the increment. We expect that this strategy would prove to be inefficient for large increments since the previous mining results are not used at all in this mining process.

Second, it adopts the Complete Closure approach. The complete closure is however computed only after having mined the increment. Therefore, unlike Borders, the candidate explosion problem is unlikely to occur because more candidates can be pruned. After computing each level of the closure, itemsets in N^{db} are excluded from further candidate generation. However, even with this pruning, there are likely to be too many unnecessary candidates in TBAR, especially for skewed increments since it relies solely on the increment for its pruning.

4.4. Other Algorithms

Recently a new algorithm for first-time mining called **CARMA** was proposed in [13] where its applicability to incremental mining was also briefly mentioned. Although the algorithm is a novel and efficient approach to first-time mining, we note that it suffers from the following drawbacks when applied to incremental mining: (1) It does not maintain negative border information and hence will need to access the original database DB if there are any locally large itemsets in the increment, even though these itemsets may not be globally large. (2) The shrinking support intervals which CARMA maintains for candidate itemsets are not likely to be tight for itemsets that become potentially large while processing the increment. This is because the number of occurrences of such itemsets in DB will be unknown and could be as much as $sup_{min} * |DB|$.

An incremental mining algorithm, called **MLUp**, for updating "multi-level" association rules over a taxonomy hierarchy was presented in [10]. While MLUp's goal is superficially similar to the incremental hierarchical mining discussed in this paper, it has the following major differences: Firstly, a *different* minimum support threshold is used for each level of the hierarchy. Secondly, MLUp restricts its attention to deriving *intra-level* rules, that is, rules within each level. In contrast, our focus in this paper is on the formulation given in [16] where there is only one minimum support threshold and *inter-level* rules form part of the output.

5. PERFORMANCE STUDY

In the previous sections, we presented the **FUP**, **Borders** and **TBAR** incremental mining algorithms, apart from our new **DELTA** algorithm. To evaluate the relative performance of these algorithms and to confirm the claims that we have informally made about their expected behavior, we conducted a series of experiments that covered a range of database and mining workloads. The performance metric in these experiments is the *total execution time* taken by the mining operation. (Note that, as mentioned in Section 4, both FUP and Borders do not compute the mining results for solely the increment, and hence their execution times do not include the additional processing required to generate these results.)

5.1. Baseline Algorithms

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

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

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

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

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

Table 2: Parameter Table

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

Table 3: Taxonomy Parameter Table

pass over db, it also determines the counts within db of itemsets in $L^{db} \cup N^{db}$. Duplicate candidates are avoided by retaining only one copy of each of them. So, in the worst case, it needs to make one pass over the previous database and one pass over the increment.

For evaluating the performance of DELTA on hierarchical databases, we compared it with **Cumulate** and ORACLE as no previous incremental algorithms are available for comparison. We chose Cumulate among the algorithms proposed in [16] since it performed the best on most of our workloads. The hierarchical databases were generated using the same technique as in [16].

5.2. Database Generation

The databases used in our experiments were synthetically generated using the technique described in [3] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator are described in Table 2. These are similar to those used in [3] except that the size and skew of the increment are two additional parameters. Since the generator of [3] does not include the concept of an increment, we have taken the following approach, similar to [8]: The increment is produced by first generating the entire $DB \cup db$ and then dividing it into DB and db.

Additional parameters required for the taxonomy in our experiments on hierarchical databases are shown in Table 3. The values of these parameters are identical to those used in [16].

5.2.1. Data Skew Generation

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

The item that is used to replace the existing item is chosen uniformly from the set of those items that are not already in the itemset. After the large itemsets are changed in this manner, d number of transactions are produced with the new modified set of potentially large itemsets.

5.3. Itemset Data Structures

In our implementation of the algorithms, we generally use the *hashtree* data-structure [3] as a container for itemsets. However, as suggested in [2], the 2-itemsets are not stored in hashtrees but instead in a 2-dimensional triangular array which is indexed by the large 1-itemsets. It has been reported (and also confirmed in our study) that adding this optimization results in a considerable improvement in performance. All the algorithms in our study are implemented with this optimization.

5.4. Overview of Experiments

We conducted a variety of experiments to evaluate the relative performance of DELTA and the other mining algorithms. Due to space limitations, we report only on a representative set here. In particular, the results are presented for the workload parameter settings shown in Table 2 for our experiments on non-hierarchical (flat) databases.

The parameters settings used in our experiments on hierarchical databases are identical except for the number of items (N) and the number of potentially large itemsets (L) which were both set to 10000. The specific values of additional parameters required for the taxonomy are shown in Table 3.

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

The previous database size was always kept fixed at 4 million transactions. Along with varying the support thresholds, we also varied the size of the increment db from 40,000 transactions to 4 million transactions, representing an increment-to-previous database ratio that ranges from 1% to 100%. For our experiments on hierarchical databases, the performance was measured only for supports between 0.75% and 2% since for lower supports, the running time of all the algorithms was in the range of several hours.

Two types of increment distributions are considered: *Identical* where both DB and db have the same itemset distribution, and *Skewed* where the distributions are noticeably different. For the *Skewed* distribution for which results are reported in this paper, the p_{is} and p_{it} parameters were set to 0.33 and 0.5 as mentioned in Table 2. With these settings, at the 0.5 percent support threshold and a 10% increment, for example, there are over 700 large itemsets in db which are not large in DB, and close to 500 large itemsets in DB that are not large in db.

We also conducted experiments wherein the new minimum support threshold is different from that used in the previous mining. The previous threshold was set to 0.5% and the new threshold was varied from 0.2% to 1.5%. Therefore, both the Stronger Threshold and Weaker Threshold cases outlined in Section 2 are considered in these experiments.

6. EXPERIMENTAL RESULTS

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

6.1. Experiment 1: Flat / Equi-Support / Identical Distribution

Our first experiment considers the equi-support situation with identical distribution between DB and db on flat databases. For this environment, the execution time performance of all the mining algorithms is shown in Figures 3a–d for increment sizes ranging from 1% to 100%.

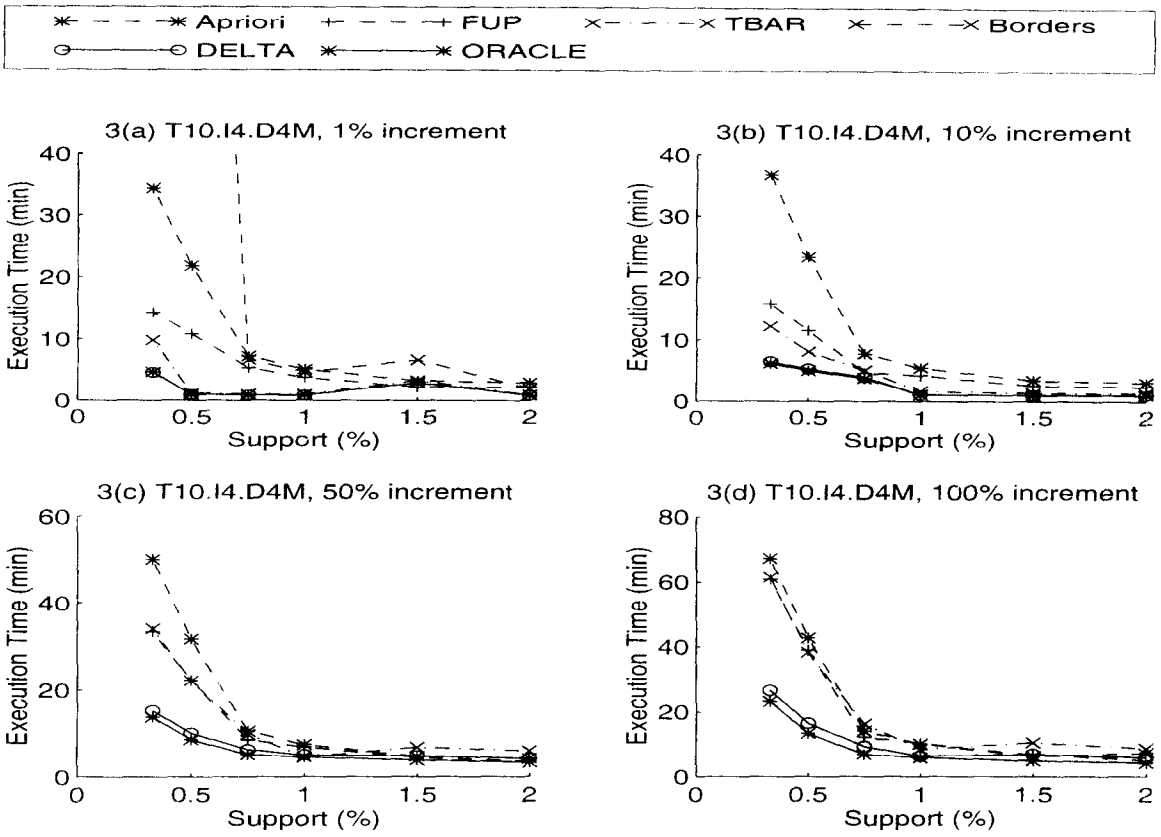

Fig. 3: Flat / Equi-Support / Identical Distribution

Focusing first on FUP, we see in Figure 3 that for all the increment sizes and for all the support factors, FUP performs better than or almost the same as Apriori. Moving on to TBAR, we observe that it outperforms both Apriori and FUP at small increment sizes and low supports. At high supports, however, it is slightly worse than Apriori due to the overhead of maintaining the negative border information. As the increment size increases, TBAR's performance becomes progressively degraded. This is explained as follows: Firstly, TBAR updates the counts of itemsets in $L^{DB} \cup N^{DB}$ over *db* – these itemsets are precisely the same as the set of all candidates generated in running Apriori over DB. Secondly, it performs a complete Apriori-based mining over *db*. When $| db | = | DB |$, the total cost of these two factors is the same as the total cost incurred by the Apriori algorithm. However, TBAR finally loses out because it needs to make a further pass over DB.

Turning our attention to Borders, we find in Figure 3a, which corresponds to the 1 percent increment, that while for much of the support range its performance is similar to that of FUP and TBAR, there is a sharp degradation in performance at a support of 0.75 percent. The reason for this is the "candidate explosion" problem described earlier in Section 4. This was confirmed by measuring the number of candidates for supports of 1 percent and 0.75 percent – in the former case, it was a little over 1000 whereas in the latter, it had jumped to over 30000!

The above candidate explosion problem is further intensified when the increment size is increased, to the extent that its performance is an order of magnitude worse than the other algorithms – therefore we have not shown Borders performance in Figures 3b–d.

Finally, considering DELTA, we find that it significantly outperforms all the other algorithms at lower support thresholds for all the increment sizes. In fact, in this region, *the performance of DELTA almost coincides with that of ORACLE.* The reason for the especially good performance here is the following – low support values result in tighter values of k, the maximal large itemset size, leading to correspondingly more iterations for FUP over the previous database DB, and for TBAR over the increment *db*. In contrast, DELTA requires only three passes over the increment

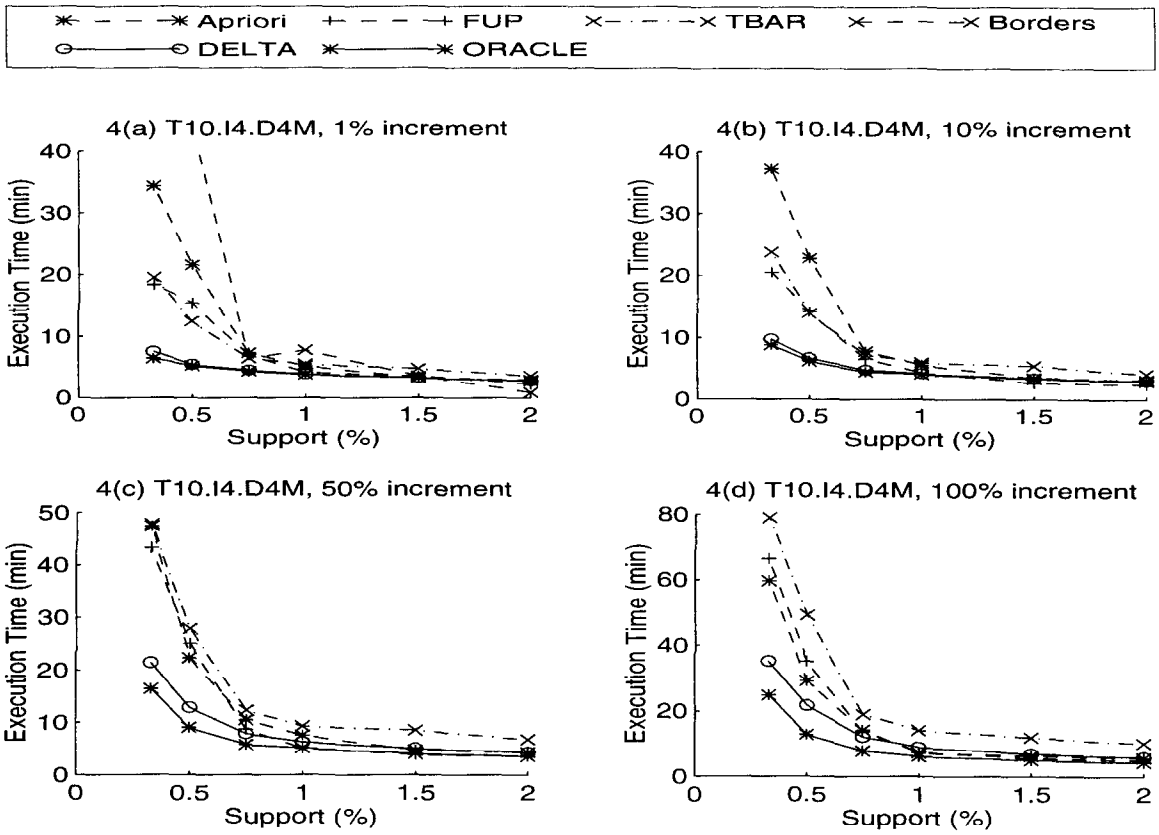

Fig. 4: Flat / Equi-Support / Skewed Distribution

and one pass over the previous database. Further, because of its pruning optimizations, the number of candidates to be counted over the previous database DB is significantly less as compared to TBAR – for example, for a support threshold of 0.5 percent and a 50% increment (Figure 3c), it is smaller by a factor of *two*.

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

6.2. Experiment 2: Flat / Equi-Support / Skewed Distribution

Our next experiment considers the Skewed workload environment, all other parameters being the same as that of the previous experiment. The execution time performance of the various algorithms for this case is shown in Figures 4a–d. We see here that the effect of the skew is pronounced in the case of both TBAR and Borders, whereas the other algorithms (including DELTA) are relatively unaffected.

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

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

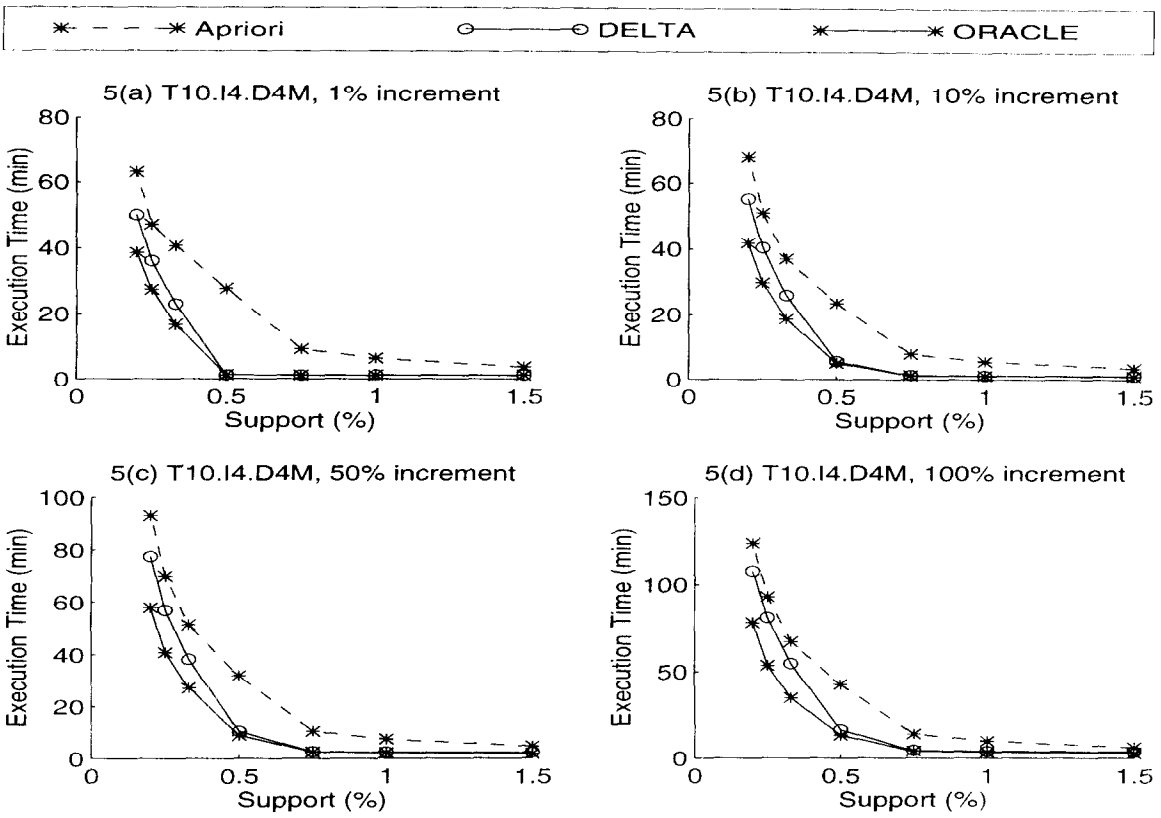

Fig. 5: Flat / Multi-Support / Identical Distribution [Previous Support = 0.5%]

DELTA is not as affected by skew as TBAR since it utilizes the *complete* negative border information to prune away candidates. That is, all itemsets which are known to be small either over $DB \cup db$ or over db are pruned away during closure generation, and not merely those candidates which are small over db. Hence, DELTA is relatively stable with respect to data skew. As in the Identical distribution case, it can be seen in Figures 4a–b that for small increment sizes, its performance almost coincides with that of ORACLE. It however degrades to some extent for large skewed increments because of two reasons: (1) the number of itemsets in $L^{DB} - L^{DB \cup db}$ increases, resulting in more unnecessary candidates being updated over db, and (2) the number of itemsets in $L^{DB \cup db} - L^{DB}$ increases, resulting in more promoted borders followed by more candidates over DB. Even in these latter cases it is seen to perform considerably better than other algorithms. For example, for a minimum support of 0.33% and an increment of 100%, its performance is more than twice as good as that of TBAR.

6.3. Experiment 3: Flat / Multi-Support / Identical Distribution

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

In this experiment, we fixed the initial support to be 0.5% and the new support was varied between 0.2% and 1.5%, thereby covering both the Weaker Threshold and Stronger Threshold possibilities. For this environment, Figures 5a–d show the performance of DELTA relative to that of Apriori for the databases where the distribution of the increments is Identical to that of the previous database.

We note here that at either end of the support spectrum, DELTA performs very similarly to Apriori whereas in the "middle band" it does noticeably better, especially for moderate increment

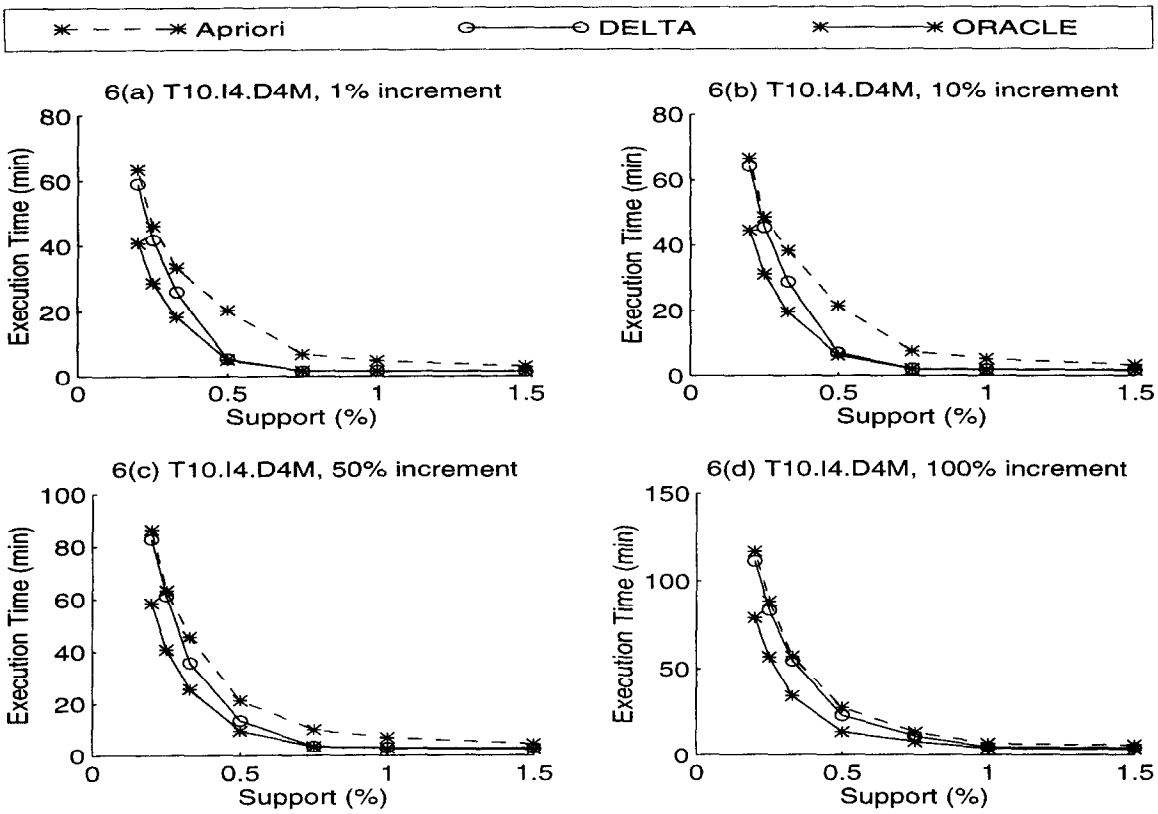

Fig. 6: Flat / Multi-Support / Skewed Distribution [Previous Support = 0.5%]

sizes (Figures 5a–b). In fact, the performance gain of DELTA is maximum when the new minimum support threshold is the same as the previous threshold and tapers off when the support is changed in either direction. At very low support thresholds, the number of large itemsets increases exponentially, and therefore the number of candidates generated in the negative border closure in DELTA will be a few more than the number of candidates generated in Apriori. Most of the candidates will have support less than the previous minimum threshold, and hence all of them have to be counted over the previous database. Therefore, the performance of DELTA approaches that of Apriori in the low support region. In the high support region, on the other hand, most of the candidates do not turn out to be large and hence both algorithms perform almost the same amount of processing.

6.4. Experiment 4: Flat / Multi-Support / Skewed Distribution

Our next experiment evaluates the same environment as that of the previous experiment, except that the distribution of the increments is Skewed with respect to the original database. The execution time performance for this case is shown in Figures 6a–d. We see here that the relative performance of the algorithms is very similar to that seen for the Identical workload environment. Further, as in the equi-support skewed case (Experiment 2), DELTA is stable with respect to skew since it uses information from both *DB* and *db* to prune away candidates. Only when the increment size is 100% do we notice some degradation in the performance of DELTA. However, it performs slightly better than Apriori even for this large increment.

6.5. Experiment 5: Hierarchical / Equi-Support / Identical Distribution

The previous experiments were conducted on flat databases. We now move on to experiments conducted on *hierarchical* databases. In these experiments, we compare the performance of DELTA

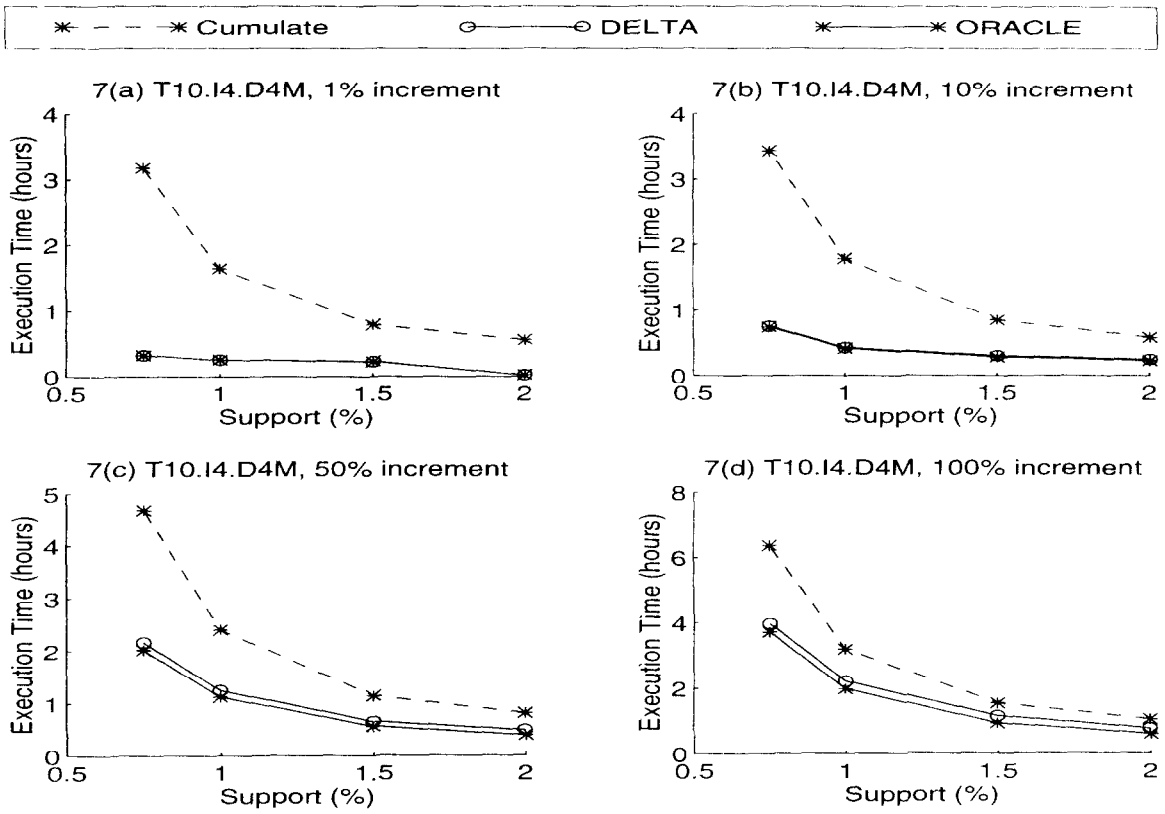

Fig. 7: Hierarchical / Equi-Support / Identical Distribution

with that of Cumulate and ORACLE only since, as mentioned earlier, no incremental algorithms are available for comparison. The execution time performance of the various algorithms for this case is shown in Figures 7a–d. Note that the time taken to complete mining is measured in *hours* here as compared to the *minutes* taken in the previous experiments. The reason for this large increase is that the number of large itemsets is much more (about 10–15 times) – this is because itemsets can be formed both within and across levels of the item taxonomy graph.

For all support thresholds and database sizes, we find that DELTA significantly outperforms Cumulate, and is in fact very close to ORACLE. We see that DELTA exhibits a huge performance gain over Cumulate, upto *as much as 9 times* at the 1% increment and 0.75% support threshold, and as much as 3 times on average. In fact, the performance of DELTA is seen to overlap with that of ORACLE for small increments (Figures 7a–b). The reason for this is the number of candidates in DELTA over both db and DB were only marginally more than that in ORACLE. This is again because the set of large itemsets with its negative border is relatively stable, and DELTA prunes away most of the unnecessary candidates in its second pass over the increment.

Due to space constraints, the experimental results for hierarchical databases where the increment's distribution is Skewed, as also the multi-support environments, are not presented here. They are available in [15] and are similar in nature to those presented earlier in this paper for flat databases.

7. CONCLUSIONS

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

border, and their associated supports, to efficiently identify the same information for the updated database.

We proposed a new algorithm called DELTA which is the result of a synthesis of existing algorithms, designed to address each of their specific limitations. It guarantees completion of mining in three passes over the increment and one pass over the previous database. This compares favorably with previously proposed incremental algorithms like FUP and TBAR wherein the number of passes is a function of the length of the longest large itemset. Also, DELTA does not suffer from the candidate explosion problem associated with the Borders algorithm owing to its better pruning strategy.

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

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

In the special scenario where no pass over the previous database is required since the new results are a subset of the previous results, DELTA's performance is optimal in that it requires only one pass over the increment whereas all the other algorithms either are unable to recognize the situation or require multiple passes over the increment.

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

In summary, DELTA is a practical, robust and efficient incremental mining algorithm. In our future work, we plan to extend the DELTA algorithm to handle quantitative rules [17] and also to develop incremental algorithms for sequence [4] and classification rules [6].

Acknowledgements — This work was supported in part by research grants from the Department of Science and Technology and the Department of Biotechnology, Government of India. We thank Shiby Thomas and Rajeev Rastogi for their expert feedback and suggestions on the material presented here.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, Washingtion, D.C, pp. 207–216, ACM Press (1993).

[2] R. Agrawal and J. Shafer. Parallel mining of association rules: design, implementation and experience. Technical Report RJ10004, IBM Almaden Research Center, San Jose, CA 95120 (1996).

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, Santiago de Chile, Chile, pp. 487–499, Morgan Kaufmann (1994).

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, pp. 3–14. IEEE Computer Society (1995).

[5] Y. Aumann, R. Feldman, O. Lipsttat, and H. Manilla. Borders: an efficient algorithm for association generation in dynamic databases. In *Journal of Intelligent Information Systems*, pp. 61–73, Kluwer Academic Publishers, http://www.isse.gmu.edu/JIIS/ (1999).

[6] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Belmont: Wadsworth (1984).

[7] S. Brossette, A. Sprague, J. Hardin, K. Waites, W. Jones, and S. Moser. Association rules and data mining in hospital infection control and public health surveillance. In *Journal of the American Medical Informatics Association*, pp. 373–381, American Medical Informatics Association, http://www.jamia.org/ (1998).

[8] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: an incremental updating technique. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, New Orleans, Louisiana, pp. 106–114, IEEE Computer Society (1996).

[9] D. Cheung, S. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA)*, Melbourne, Australia, pp. 185-194, World Scientific (1997).

[10] D. Cheung, T. Vincent, and W. Benjamin. Maintenance of discovered knowledge: a case in multi-level association rules. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, Portland, Oregon, pp. 307-310, AAAI Press (1996).

[11] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *Proceedings of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Tucson, Arizona (1997).

[12] M. Garofalakis, S. Ramaswamy, R. Rastogi, and K. Shim. Of crawlers, portals, mice, and men: is there more to mining the web? In *Proceedings of the 28th ACM SIGMOD International Conference on Management of Data*, Philadephia, Pennsylvania, p. 504, ACM Press (1999).

[13] C. Hidber. Online association rule mining. In *Proceedings of the 28th ACM SIGMOD International Conference on Management of Data*, Philadephia, Pennsylvania, pp. 145–156, ACM Press (1999).

[14] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*, San Jose, California, pp. 175–186, ACM Press (1995).

[15] V. Pudi and J.R. Haritsa. *Quantifying the Utility of the Past in Mining Large Databases.* Technical Report TR-2000-01, DSL, Indian Institute of Science, http://dsl.serc.iisc.ernet.in/pub/TR/TR-2000-01.ps (2000).

[16] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, Zürich, Switzerland, pp. 407–419, Morgan Kaufmann (1995).

[17] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of the 25th ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, pp. 1–12, ACM Press (1996).

[18] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD)*, Newport Beach, California, pp. 263–266, AAAI Press (1997).

[19] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, Bombay, India, pp. 134–145, Morgan Kaufmann (1996).