Updating of Association Rules Dynamically

King-Kwok Ng and Wai Lam
The Chinese University of Hong Kong
Department of Systems Engineering and Engineering Management
Shatin, Hong Kong
{kkng,wlam}@se.cuhk.edu.hk

Abstract

We propose a new algorithm to handle the problem of updating of association rules. Recent methods on this problem usually employ the Apriori algorithm. We develop a new algorithm, called the Incremental Dynamic Itemset Counting algorithm. It makes use of the dynamic counting technique to deal with this problem in a more efficient way. Experimental results show that our new algorithm outperforms a recent incremental association rule mining algorithm in terms of the computational time. We also investigate a variant of our algorithm and demonstrate its effectiveness.

1. Introduction

Data mining has recently attracted a large amount of attention due to its wide application. In particular, association rule is an important type of knowledge which can be discovered from large databases. Association rules have been widely used in areas like marketing, decision support and store layout design [3]. An example of association rule is $BREAD \Rightarrow MILK$. If it is discovered from a supermarket transaction database, it may mean that "when people buy bread, they usually also buy milk". Besides binary and categorical data, some databases may also contain quantitative, or numerical data. Much work has been done for discovering association rules from such quantitative databases [10, 12, 7]. If a conceptual hierarchy is available for the data items in the database, then generalized, or multiple-level association rules can be discovered [9, 6, 8]. A generalized, or multiple-level association rule may contain items from different levels of the conceptual hierarchy.

Assume we have a database and some association rules discovered from this database. Now, if some new data comes to the database, we will need to update the set of the discovered association rules. One simple way is to run an association rule mining algorithm over the incremented database to get the new set of association rules. However, this approach is not efficient. To address this problem, we explore some incremental association rule mining techniques to avoid the redundant work of the whole mining process over the incremented database. Recently, there have been some studies on the incremental mining of association rules. Cheung et al. developed an approach known as the FUP algorithm [4]. The FUP algorithm mainly incorporates the classical level-wise Apriori algorithm [1] to find large itemsets and their support counts in the original as well as incremental databases. It demonstrates good performance over purely Apriori technique. Thomas et al. developed another approach which makes use of the *negative border* of the original database [11]. It incrementally updates the negative borders when a set of new transactions is available. Although this additional information allows more efficient algorithms to be applied, it induces a considerable amount of overhead in calculating the negative border. We propose a new algorithm for incremental mining of association rules based on dynamic counting. Brin et al. has developed a dynamic counting technique known as the Dynamic Itemset Counting (DIC) for large itemset counting in a *single* database. Direct application of DIC in incremental updating is not efficient since available information stored in the original database is not useful. We still need to count the original database again and result in long computational time. Our new algorithm, called the IDIC_M algorithm and its variant, the IDIC_1 algorithm, can efficiently conduct incremental mining of association rules based on dynamic counting. Moreover, it avoids the overhead incurred in the previous algorithm based on negative borders [11]. We have conducted extensive experiments to compare our algorithms with the FUP algorithm. The results illustrate that our two algorithms have better performance over the FUP algorithm.

0-7695-0496-5/00 $10.00 © 2000 IEEE

2. Problem description

2.1. Mining of association rules

Assume we have a set of literals called items. Let DB be a database of transactions, where each transaction contains a set of different number of items. An association rule is an implication of the form $X \Rightarrow Y$, where X and Y are itemsets and $X \cap Y = \emptyset$. We call an itemset containing k items a k-itemset. The association rule $X \Rightarrow Y$ holds in DB with *confidence* c if $c\%$ of the transactions in DB that contain X also contain Y. The association rule $X \Rightarrow Y$ has *support* s in DB if $s\%$ of the transactions in DB contain $X \cup Y$. Given a minimum confidence threshold θ_f and a minimum support threshold θ_s, the problem of mining association rules is to find out all the association rules whose confidence and support are larger than the respective thresholds. For an itemset X, its support is defined similarly as the percentage of transactions in DB which contain X. Given a minimum support threshold θ_s, an itemset X is large if its support is no less than θ_s. The problem of mining association rules is reduced to the problem of finding all large itemsets for a pre-determined minimum support [3]. This is due to the fact that discovering large itemsets in a database requires most of the computational time used in mining association rules. Association rules can be easily found once a set of large itemsets are available.

2.2. Incremental mining of association rules

Let L_{DB} be the set of large itemsets in the database DB, s be the minimum support, and D be the number of transactions in DB. Assume that for each $X \in L_{DB}$, its support count which is the number of transactions in DB containing X, is available. Suppose we gather a set of new additional transactions db. The problem of incremental mining is to find all itemsets having support s in the new database, i.e., $DB \cup db$.

Let d be the number of transactions in db. With respect to the same minimum support s, an itemset X is large in the updated database $DB \cup db$ if the support of X in $DB \cup db$ is no less than s, i.e., $X.support_{UD} \geq s \times (D + d)$, where $X.support_{UD}$ denotes the support counts for itemset X in $DB \cup db$. In the following, we use $X.support_{DB}$ and $X.support_{db}$ to represent the support counts of an itemset X in DB and db, respectively. Similarly, we denote with L_{db} and L_{UD} the set of large itemsets in db and $DB \cup db$ respectively. Thus the essence of incremental mining of association rules is to find the set L_{UD}. Note that an itemset in L_{DB} may not be an itemset in L_{UD}. Also, an itemset not in L_{DB}, may become a large itemset in L_{UD}.

3. Our new approach

3.1. Challenges and motivation

Dynamic counting technique has been explored by Brin et al. who developed the DIC algorithm for itemset counting in a *single* database [2]. While Apriori requires the same number of passes through the database as the size of the maximum large itemsets, dynamic counting allows candidates of various sizes to be counted at the same time in each pass. This results in the reduced number of passes through the database.

One previous work for incremental association rule mining is the FUP algorithm [4]. It is mainly based on the Apriori algorithm and achieves satisfactory performance. In each iteration of FUP, only candidate itemsets of one size are handled. It is performed by making one pass over the increment db and one pass over the original database DB. Therefore, the number of passes over the original database equals the size of the highest order itemsets to be updated. To reduce the passes over the DB, we developed two new Incremental Dynamic Itemset Counting algorithms, the IDIC_M and the IDIC_1, which are based on dynamic counting. In both algorithms, we compress the passes over db performed in the FUP algorithm by dynamic counting technique. On the other hand, without checking for each k-itemset candidate found immediately after one pass through db against DB, a lot more candidates are generated in each of the next pass through db and hence, the total execution time used to scan db may increase. Also, a large amount of candidate itemsets are generated after scanning db. Two different methods are used respectively to handle this problem. In the IDIC_M algorithm, we compress the passes over DB by apply dynamic counting technique. We allow only the k-itemsets with all its size $k-1$ subsets being large in $DB \cup db$ to be counted in DB. This means that the candidates actually being counted in DB are reduced to only the essential ones and there are not too many of them. However, note that it may need to perform more than one passes over DB. In the other variant algorithm, IDIC_1, we perform exactly one scan over DB to count all the candidate itemsets generated from db. While some excessive candidates may be counted, the number of passes over DB is guaranteed to be minimum. By using either of our proposed algorithms, it is expected that the execution time saved in scanning DB should be much larger than the increased time in counting the extra candidates in db.

Note that in the IDIC_M algorithm, it is not efficient if we apply the original DIC algorithm directly in counting DB for our incremental counting problem. Since, for the original DIC algorithm, we have to start counting only 1-itemsets in the first interval and cannot start counting k-itemsets until the k-th interval. This limitation is not desir-

able in our case since we already know the candidates to be checked in DB. Actually, once we have discovered all potential L_{UD} candidates after scanning *db*, we can just make one pass through DB to check the counts of all those candidates. One possible drawback for such approach is that a large amount of unnecessary candidates may be counted in DB, causing considerable overhead. In contrast, our IDIC_M algorithm will start counting any candidates in DB as soon as it is proven eligible (i.e., it has all its subsets being large in $DB \cup db$). For example, if an 4-itemset has all four of its size-3 subsets being large in L_{UD}, we can start counting it immediately in the first interval of DB instead of counting it in the fourth interval.

Remember that the major target of the FUP algorithm is to minimize the number of candidate itemsets to be generated and counted. This is done by pruning the candidates in each pass through *db* and DB. On the other hand, our two algorithms allow extra candidate itemsets to be generated from *db*, while the efficiency is maintained by the reduced passes through DB. Usually, a large amount of candidates are generated from *db* and we have to be efficient when we count those candidates in DB. Our two variant algorithms try to deal with this problem in two different ways. Also note that we cannot trivially incorporate DIC into the FUP algorithm, since DIC is not a level-wise algorithm and cannot be used directly in place of Apriori in FUP.

3.2. Review of the DIC algorithm

Since our algorithm will make use of the DIC algorithm [2], we first give a brief description of the DIC algorithm. In the DIC algorithm each itemset is classified as one of the following categories:

- confirmed-large : an itemset we have finished counting that exceeds the support threshold.
- confirmed-small : an itemset we have finished counting that is below the support threshold.
- suspected-large : an itemset we are still counting that exceeds the support threshold.
- suspected-small : an itemset we are still counting that is below the support threshold.

The DIC algorithm works as follows:

1. All the 1-itemsets are marked suspected-small. All other itemsets are unmarked.
2. Read M transactions (where M is a predetermined interval size). For each transaction, increment the respective counters for the itemsets marked suspected-large or suspected-small.
3. If a suspected-small itemset has a count that exceeds the support threshold, turn it into a suspected-large. If any immediate superset of it has all of its subsets as confirmed-large or suspected-large, add a new counter for it and mark it suspected-small.
4. If a suspected-small (or suspected-large) itemset has been counted through all the transactions, mark it confirmed-small (or confirmed-large) and stop counting it.
5. If we are at the end of the transaction file, rewind to the beginning.
6. If any suspected-large or suspected-small itemsets remain, go to Step 2.

What DIC does is just to create possible new candidate itemsets at every M-transaction interval. Note that for the Apriori algorithm, new candidate itemsets can be created only at the end of a pass through the database.

3.3. The IDIC_M algorithm

We propose a new algorithm, the IDIC_M algorithm, for incremental mining based on dynamic itemset counting.

Lemma 1 An itemset X is in L_{UD} only if it is either in L_{DB} or in L_{db} or in both L_{DB} and L_{db}. It means that all large itemsets in the updated database, i.e., L_{UD} must be included in $L_{DB} \cup L_{db}$.

Proof. If X is neither in L_{DB} nor L_{db}, then $X.support_{DB} < s \times D$ and $X.support_{db} < s \times d$. Therefore, $X.support_{DB} + X.support_{db} < s \times (D + d)$. Then X is not in L_{UD}.□

The IDIC_M algorithm works as follows:

1. **Scan the *db***

   Scan the increment *db* and use DIC to find all itemsets which are large in *db*, i.e., L_{db}. For each X in L_{db}, record its support in *db* as $X.support_{db}$.

2. **Consider itemsets in L_{DB}**

   For each itemset X in L_{DB}, if $X.support_{DB} + X.support_{db} \geq s \times (D + d)$, it is added to the set of large itemsets in the updated database $DB \cup db$, i.e., L_{UD}, with the updated count. Note that the itemsets we examine in this step already includes all itemsets in $L_{DB} \cap L_{db}$ which can be shown in the following.

   Lemma 2 All itemsets in $L_{DB} \cap L_{db}$ are examined in Step 2 of the IDIC_M algorithm.

   Proof. From the description, we check each itemsets in L_{DB} if we have its count $X.support_{db}$ (Any itemset in L_{DB} we already have its count $X.support_{DB}$). For all itemsets in L_{db} we should have their counts in *db* from

Step 1. Therefore, any itemset that is both large in DB and large in db has been examined in Step 2.□

Example 1 Assume a database DB with size $D = 100000$ is updated with an increment db with size $d = 1000$, where θ_s is 1%. Let I_1, I_2, I_3 be three itemsets in L_{DB}, with $I_1.support_{DB} = 1005$, $I_2.support_{DB} = 1020$ and $I_3.support_{DB} = 1007$, respectively. And we have $I_1.support_{db} = 15$, $I_2.support_{db} = 8$ and $I_3.support_{db} = 2$ after Step 1. Note that I_1 is an large itemset in L_{db} while I_2 and I_3 are not. In step 2, what we do is to check each of the three itemsets if it should be added into L_{UD}. First, since $I_1.support_{UD} = I_1.support_{DB} + I_1.support_{db} > 101000 \times 1\%$, I_1 will be added into L_{UD}. Similarly, $I_2.support_{UD} = I_2.support_{DB} + I_2.support_{db} > 101000 \times 1\%$, and I_2 will also be added into L_{UD}. However, as $I_3.support_{UD} = I_3.support_{DB} + I_3.support_{db} < 101000 \times 1\%$, I_3 will not be added into L_{UD}.

3. **Consider itemsets in $L_{DB} - L_{db}$**

   Check all itemsets in $L_{DB} - L_{db} - L_{UD}$, i.e., those itemsets that are large in DB but not large in db and not already large in $DB \cup db$, by scanning db once to update their counts. For each itemset X in $L_{DB} - L_{db} - L_{UD}$, add it into L_{UD} with the updated count if $X.support_{DB} + X.support_{db} \geq s \times (D + d)$. Note that this step is necessary as we did not keep all the counts for those itemsets which are not large in db.

   Example 2 Note that we only keep the counts of some small itemsets in db. Say, in Step 1, when we are counting db, we find itemsets $\{2,3\}$, $\{3,4\}$ and $\{2,4\}$ to be large in L_{db} and we start to count itemset $\{2,3,4\}$. However, we finally find that $\{2,3,4\}$ is small in db. Then we will have the count for itemset $\{2,3,4\}$ even it is not large in L_{db}. On the other hand, if we find one of the itemsets $\{2,3\}$, $\{3,4\}$ and $\{2,4\}$ to be small in db, we will not try to count itemset $\{2,3,4\}$ and so will not have the count for itemset $\{2,3,4\}$.

   Example 3 Assume a database DB with size $D = 100000$ is updated with an increment db with size $d = 1000$, where θ_s is 1%. Let I_1, I_2, I_3 be three itemsets in L_{DB} with supports $I_1.support_{DB} = 1005$, $I_2.support_{DB} = 1007$ and $I_3.support_{DB} = 1008$, respectively. Also, assume that after Step 2, we find I_3 to be a large itemset in db with $I_3.support_{db} = 17$, while I_1 and I_2 are small in db. We do not have the support count for I_1 in db but we have the count for I_2 in db as $I_2.support_{db} = 8$, which is possible for an itemset not in L_{db} as seen from the previous example. Now, as I_1 is in L_{DB} but not in L_{db} and L_{UD}, so we have to count it once in Step 3. Though I_2 is in L_{DB} and not in L_{db}, it is not counted in Step 3 since it is in L_{UD}. It means that I_2 is already handled in Step 2. Similarly, I_3 is not counted in Step 3 neither, since it is in both L_{DB} and L_{db} and is already handled in Step 2.

4. **Consider itemsets in $L_{db} - L_{DB}$**

   In this step, we determine which itemsets in $L_{db} - L_{DB} - L_{UD}$ are large in $DB \cup db$. We do this by applying a dynamic counting technique, similar to the DIC algorithm, to DB. Before counting, each k-itemset in $L_{db} - L_{DB}$, which has all its possible size $(k-1)$-subsets being in L_{UD}, is added to the candidate set to be counted, provided that it is not already in L_{UD}. Note that any k-itemset can contain at most k subsets of size $k-1$. All 1-itemsets in $L_{db} - L_{DB} - L_{UD}$ are also added to the candidate set. The counts of all the candidates are initialized with their supports in db. After the counting of each interval is finished, new large itemsets are identified and added to L_{UD} with the updated supports. New candidates are generated from L_{UD} by a function *apriori-gen*, developed by Agrawal et al., which takes as input the set of all large $(k-1)$-itemsets and returns a superset of the set of all large k-itemsets [1]. These new candidates are then added to the candidate sets if it is not already in L_{UD}. Continue the process until there are no more candidates to be counted.

   Example 4 Assume we now have in L_{UD} 1-itemsets $\{1\}$, $\{2\}$ and $\{3\}$, 2-itemsets $\{1,2\}$, $\{2,3\}$, $\{5,6\}$, $\{6,7\}$ and $\{5,7\}$. Also, assume that we have itemsets $\{4\}$, $\{1,3\}$, $\{1,2,3\}$ and $\{5,6,7\}$ in L_{db}. In Step 4, we start to count $\{1,3\}$ in the first interval because it has 2 of its 1-subsets $\{1\}$ and $\{3\}$ in L_{UD}. Similarly, we start to count $\{5,6,7\}$ since its 3 2-subsets $\{5,6\}$, $\{6,7\}$ and $\{5,7\}$ are in L_{UD}. The 1-itemset $\{4\}$ is also counted in the first interval. If we find $\{1,3\}$ to be large after, say, counting the first interval, then we will start to count $\{1,2,3\}$ in the second interval, as all 3 of its 3-subsets $\{1,2\}$, $\{2,3\}$ and $\{1,3\}$ are now large in L_{UD}.

Lemma 3 All potential L_{UD} candidates are examined by the IDIC_M algorithm.

Proof. After the above four steps, we have checked all the itemsets in $L_{DB} \cap L_{db}$, L_{DB} and L_{db}. That is all itemsets in $L_{DB} \cup L_{db}$ have already been examined. We have covered all potential L_{UD} candidates, by Lemma 1.□

Note that although the above lemmas are similar to those introduced in [4], they give a more clear view of what can be covered in each steps of our algorithms.

3.4. Experimental results for algorithm IDIC_M

Several sets of experiments were conducted to compare the IDIC_M algorithm with the FUP algorithm and the pure DIC algorithm. Note that each data point in the experimental results is obtained from the average value over 10 trials. The synthesis data used in all the experiments are generated following the techniques in [4].

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

Table 1. Parameter table.

We use the notation Tx.Iy.Dm.dn to denote a database of size $D = 1000m$ is updated with an increment with size $d = 1000n$, while $|T| = x$ and $|I| = y$. In our experiments comparing the IDIC_M algorithm and the FUP algorithm, D and d were chosen in similar way as in [4], and $|T| = 10$, $|I| = 4$, $|L| = 2000$ and $N = 1000$.

We generated $(D + d)$ transactions by one random seed. Then, the first D transactions are stored for DB and the others for db. Note that in the second set of experiments in Section 3.4.2, DB and db were generated separately with different random seeds. For our IDIC_M algorithm, the FUP algorithm and the DIC algorithm, their execution time speedup ratios over pure Apriori algorithm are calculated to evaluate their performance. Note that pure Apriori algorithm here means applying the Apriori algorithm once over the updated database. We first combine the original database and the incremental database into a new updated database. Then, the Apriori algorithm is applied on this new updated database to obtain the new large itemsets we want. Speedup ratio data point on the curve **Apriori/IDIC_M** is calculated by dividing the execution time of the pure Apriori algorithm with the execution time of the IDIC_M algorithm.

3.4.1 IDIC_M versus FUP for different supports

The IDIC_M algorithm and the FUP algorithm were tested for different θ_s ranging from 0.01 to 0.05, with updated database T10.I4.D100.d1 which is similar to the one used in [4]. The interval sizes for counting db and DB were fixed at 1000 and 10000, respectively.

The result is shown in Figure 1. The IDIC_M algorithm clearly has a better execution time speedup ratio for all supports tested. Note that, our algorithm has a substantial advantage over the FUP algorithm when the minimum support threshold is set at at a relatively higher value. The speedup ratio of the IDIC_M algorithm is more than 1.5 times higher than the FUP algorithm when $\theta_s = 0.02$. The IDIC_M algorithm is more than 8 times faster than the pure Apriori algorithm at this support threshold value. We can see that when the support threshold is low, the IDIC_M algorithm does not have a very significant advantage over the FUP algorithm. Since a very large number of candidate itemsets are expected to be generated from db in this case, we can observe that IDIC_M may not be very efficient in dealing with those large amount of candidates.

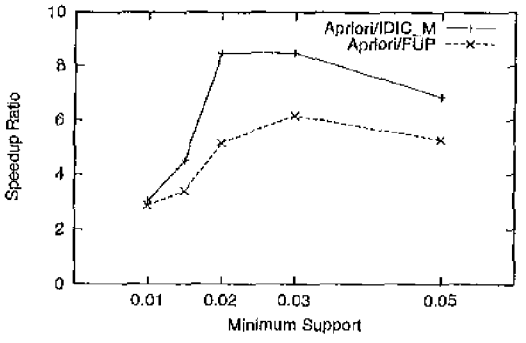

Figure 1. IDIC_M and FUP for different supports.

3.4.2 IDIC_M versus FUP for different incremental sizes

We have done two sets of experiments to compare the performance of the IDIC_M algorithm with the FUP algorithm for different incremental size, d, ranging from 1000 to 50000, with DB fixed at 100000, i.e., T10.I4.D100.dn. θ_s is fixed at 0.015. For the first set of experiments, the DB and db are generated as usual using the same random seed. While for the second set of experiments, the DB and db are generated separately using different random seeds.

The results for first set of experiments are shown in Figure 2. The IDIC_M algorithm outperforms the FUP algorithm for all incremental sizes tested. It works well especially when the incremental size is small. The greatest advantage of our algorithm over the FUP algorithm was obtained when $d = 1000$, where the speedup ratio of IDIC_M is about 7.3 and the speedup ratio of FUP is about 4. We see the trend that as the incremental size continues to increase, the performances of both the IDIC_M algorithm and the FUP algorithm decrease. This is due to the fact that when the incremental size is large relative to the original database size, the value of the known information about the

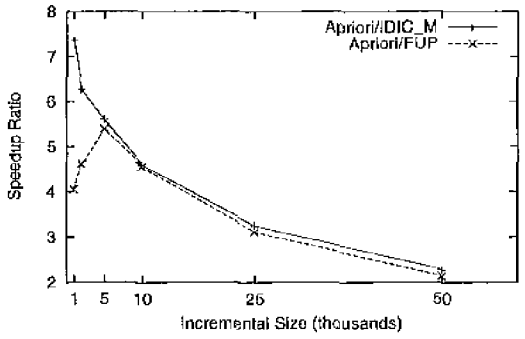

Figure 2. IDIC_M and FUP for different incremental sizes (with the same random seed).

original database will be lower. This experiment can simulate situations where the incoming data are having similar patterns to the original database.

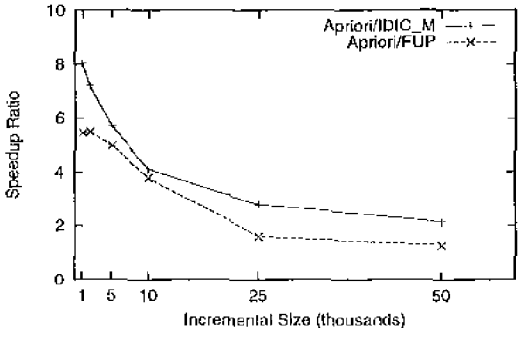

Figure 3. IDIC_M and FUP for different incremental sizes (with different random seeds).

The results for second set of experiments are shown in Figure 3. The IDIC_M algorithm also outperforms the FUP algorithm for all incremental sizes tested. Our algorithm has a considerable advantage over the FUP algorithm except when the incremental size is near to 10000. The greatest advantage of our algorithm over the FUP algorithm was also obtained at $d = 1000$. Note that the curve of the IDIC_M algorithm converges to a higher value than the curve of FUP does. Also, note that our IDIC_M algorithm is much better than the FUP algorithm in this situation where the original database and the incremental database were generated by different random seeds. The situation simulated by this experiment does exist in real life when each batch of new incoming data have a different transaction pattern.

3.4.3 IDIC_M versus DIC for different incremental sizes

Experiments were done to compare the performance of the IDIC_M algorithm with the DIC algorithm for different incremental size, d, ranging from 10000 to 50000, with DB fixed at 50000, i.e., T10.I4.D50.dn. θ_s is fixed at 0.015. We are especially interested in the situation that when the incremental size is relative large compared to the original database size. Note that what we mean by DIC here is that we applied the DIC algorithm once over the updated database.

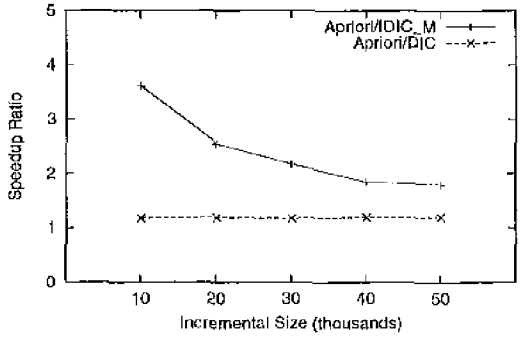

Figure 4. IDIC_M and DIC for different incremental sizes.

The results for the experiments are shown in Figure 4. Although the advantage is decreasing with the incremental size, our IDIC_M algorithm outperforms the DIC algorithm for all incremental sizes tested with considerable amount. The decreasing performance of IDIC_M is due to the less significant of known information about the original database as increment becomes larger. Since DIC is not an incremental mining algorithm, it can maintain a relatively steady advantage over the Apriori algorithm. From the results, we can conclude that the advantage of our IDIC_M algorithm over the FUP algorithm is not only contributed by the use of dynamic counting technique, but also the whole design of our IDIC_M algorithm.

3.5. A variant algorithm: the IDIC_1 algorithm

We now introduce a variant algorithm to the IDIC_M algorithm, called the IDIC_1 algorithm. While there may be more than one passes through DB in the IDIC_M algorithm, the IDIC_1 algorithm guarantees a single pass over the DB. The first three steps of the IDIC_1 algorithm are exactly the same as the IDIC_M algorithm. The Step 4 of the IDIC_1 algorithm works as follows:

In this step, we determine which itemsets in $L_{db} - L_{DB} -$

L_{UD} are large in $DB \cup db$. Instead of applying a dynamic counting technique as in the IDIC_M algorithm, we count all these candidate itemsets in one pass over DB. After the counting is finished, new large itemsets are identified and added to L_{UD} with the updated supports.

Note that all the lemmas in the previous subsection also applies in the IDIC_1 algorithm, which means that all potential L_{UD} candidates are examined by the IDIC_1 algorithm.

3.6. Experimental results for algorithm IDIC_1

Several sets of experiments were conducted to compare the IDIC_1 algorithm with the FUP algorithm and the pure DIC algorithm. The data used in this subsection for testing the IDIC_1 algorithm is similar to that used to test the IDIC_M algorithm in Section 3.4.

3.6.1 IDIC_1 versus FUP for different supports

The IDIC_1 algorithm and the FUP algorithm are tested for different θ_s ranged from 0.01 to 0.05, with updated database T10.I4.D100.d1 which is similar to the one used in [4]. The interval sizes for counting db and DB were fixed at 1000 and 10000, respectively.

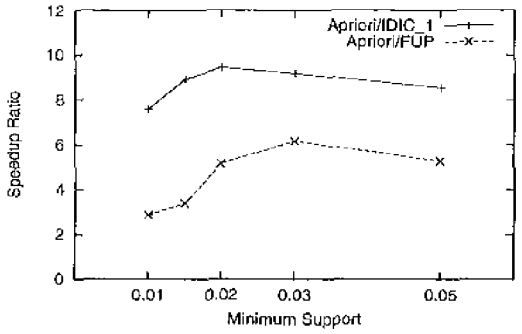

Figure 5. IDIC_1 and FUP for different supports.

The result is shown in Figure 5. The IDIC_1 algorithm clearly has a better execution time speedup ratio for all supports tested. Note that, our algorithm has a steady advantage over the FUP algorithm at different minimum support thresholds. The difference between the speedup ratio of the IDIC_1 algorithm and that of the FUP algorithm is more than 3 for all the supports tested. Compared to the IDIC_M algorithm, we can say that the IDIC_1 algorithm is much more efficient in dealing with the large number of candidate itemsets generated from db when the support threshold is low. It also has a good performance when the support threshold is high.

3.6.2 IDIC_1 versus FUP for different incremental sizes

We have done two sets of experiments to compare the performance of the IDIC_1 algorithm with the FUP algorithm for different incremental size, d, ranging from 1000 to 50000, with DB fixed at 100000, i.e., T10.I4.D100.dn. θ_s is fixed at 0.015. For the first set of experiments, the DB and db are generated as usual using the same random seed. While for the second set of experiments, the DB and db are generated separately using different random seeds.

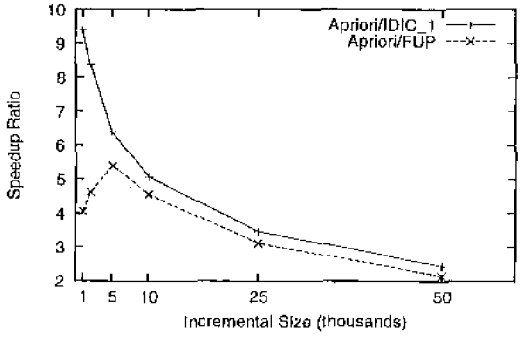

Figure 6. IDIC_1 and FUP for different incremental sizes (with the same random seed).

The results for first set of experiments are shown in Figure 6. The IDIC_1 algorithm clearly outperforms the FUP algorithm for all incremental sizes tested, especially when the incremental size is small. The greatest advantage of our algorithm over the FUP algorithm was obtained when $d = 1000$, where the speedup ratio of IDIC_1 is more than twice the speedup ratio of FUP. We also see the trend that as the incremental size continues to increase, the performances of both the IDIC_1 algorithm and the FUP algorithm decrease.

The results for second set of experiments are shown in Figure 7. The IDIC_1 algorithm also outperforms the FUP algorithm for all incremental sizes tested. Note that the curve of the IDIC_1 algorithm again converges to a higher value than the curve of FUP does.

3.6.3 IDIC_1 versus DIC for different incremental sizes

Experiments were done to compare the performance of the IDIC_1 algorithm with the DIC algorithm for different incremental size, d, ranging from 10000 to 50000, with DB fixed at 50000, i.e., T10.I4.D50.dn. θ_s is fixed at 0.015.

The results for the experiments are shown in Figure 8. Similar to the results for the IDIC_M algorithm, the advan-

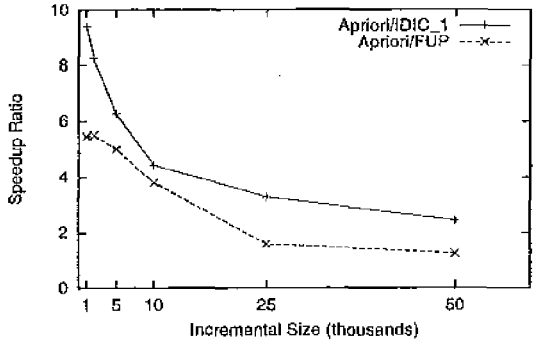

Figure 7. IDIC_1 and FUP for different incremental sizes (with different random seeds).

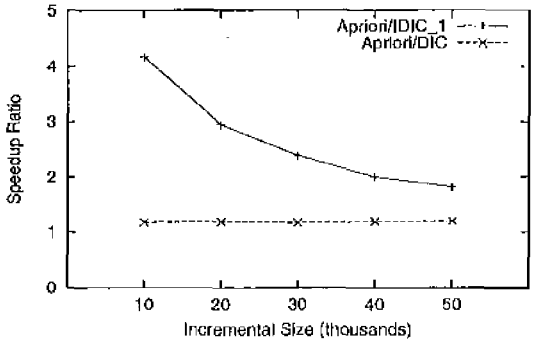

Figure 8. IDIC_1 and DIC for different incremental sizes.

tage of the IDIC_1 algorithm is decreasing with the incremental size. Our IDIC_1 algorithm shows significant advantage over the DIC algorithm. We can have the conclusion similar to the IDIC_M algorithm that dynamic counting technique is not the only cause of the efficiency of our IDIC_1 algorithm.

4. Conclusion

In this paper, we propose new incremental updating algorithms, the IDIC_M and the IDIC_1, to handle the problem of incremental association rule mining using dynamic counting technique. We describe in details our new algorithms and illustrates how dynamic counting works efficiently in this problem. We have fully implemented our algorithms. Experimental results show that our new algorithms have superior performance in comparison with another recent incremental updating algorithm, the FUP algorithm. Other than binary or categorical association rules, the incremental mining of quantitative association rules is also worth studying. As new incoming data may affect the data distribution pattern of the original database, new discretization may have to be done on the incremented database. Hence, the updating of the discovered quantitative association rules will be more complicated. Another possible research topic is the incremental mining of association rules in a distributed environment.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.

[2] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 255–264, AZ, 1997.

[3] M. S. Chen and J. Han. Data mining: An overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.

[4] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of International Conference on Data Engineering*, pages 106–114, 1996.

[5] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. Fu. Efficient mining of association rules in distributed databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):911–922, 1996.

[6] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Swizerland, 1995.

[7] C. M. Kuok, A. Fu, and M. H. Wong. Mining fuzzy association rules in databases. *ACM SIGMOD Record*, 27(1):41–46, 1998.

[8] T. Shintani and M. Kitsuregawa. Parallel mining algorithms for generalized association rules with classification hierarchy. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998.

[9] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Swizerland, 1995.

[10] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1996.

[11] 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 International Conference on Knowledge Discovery and Data Mining*, pages 263–266, 1997.

[12] K. Wang, S. H. W. Tay, and B. Liu. Interestingness-based interval merger for numeric association rules. In *Proceedings of International Conference on Knowledge Discovery and Data Mining*, pages 121–127, 1998.