Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique *

David W. Cheung[†] Jiawei Han[‡] Vincent T. Ng[††] C.Y. Wong[†]

[†] Department of Computer Science, The University of Hong Kong, Hong Kong. Email: dcheung@cs.hku.hk.

[‡] School of Computing Science, Simon Fraser University, Canada. Email: han@cs.sfu.ca.

[††] Department of Computing, Hong Kong Polytechnic University, Hong Kong. Email: cstyng@comp.polyu.edu.hk.

[†] Department of Computer Science, The University of Hong Kong, Hong Kong. Email: cywong@cs.hku.hk.

Abstract

An incremental updating technique is developed for maintenance of the association rules discovered by database mining. There have been many studies on efficient discovery of association rules in large databases. However, it is nontrivial to maintain such discovered rules in large databases because a database may allow frequent or occasional updates and such updates may not only invalidate some existing strong association rules but also turn some weak rules into strong ones. In this study, an incremental updating technique is proposed for efficient maintenance of discovered association rules when new transaction data are added to a transaction database.

1 Introduction

Database mining has recently attracted tremendous amount of attention in the database research because of its wide applicability in many areas, including decision support, market strategy and financial forecast.

According to many studies in knowledge discovery in databases [10, 4], mining knowledge from databases has the following characteristics.

1. The size of the database is significantly large, it could scale up to gigabytes, terabytes, or even larger, in some applications.

2. The rules discovered is valid only in statistical terms. Users are looking for rules that hold for a significant amount of data, but not necessarily for all the data. Therefore, the number of rules returned from a mining activity could be large.

3. The rules discovered from a database only reflect the current state of the database. To make the rules discovered stable and reliable, a large volume of data should be collected over a substantial period of time.

These observations indicate that the promise of database mining lies in the techniques to handle a large amount of data, to manage a substantial number of rules, and to maintain the rules over a significantly long period of time. Therefore, the following two problems are essential in order to make database mining a feasible technology.

1. Design efficient algorithms for mining different types of rules or patterns.

2. Design efficient algorithms to update, maintain and manage the rules discovered.

The first problem has been studied substantially with many interesting and efficient database mining algorithms reported (e.g., see [1, 2, 3, 5, 6, 8, 9, 11]). Such database-oriented knowledge mining algorithms can be classified into two categories: *concept generalization-based discovery* and *discovery at the primitive concept levels.* The former relies on the generalization of concepts (attribute values) stored in databases and then summarization of the data regularities at a high concept level. One such example is the DBLearn system [3, 5]. The latter relies on the discovery of strong regularities (rules) from the database without concept generalization. Association rule [1, 2, 9] is an important type of rules discovered by this approach.

*The research of the first three authors were supported in part by RGC (the Hong Kong Research Grants Council) grant 338/065/0026. The research of the second author was also supported in part by NSERC (the Natural Sciences and Engineering Research Council of Canada) research grant OGP0037230 and an NCE (the Networks of Centres of Excellence of Canada) research grant, IRIS-HMI5.

However, very little work has been done on the second problem. A method for handling incremental database updates for the rules discovered by the generalization-based approach was briefly discussed in [5]. However, previous work has not been seen on incremental updating of association rules. Since database updates may introduce new association rules and invalidate some existing ones, it is important to study efficient algorithms for incremental update of association rules in large databases, which is the theme of this paper.

In the pioneer work [1], it is shown that the problem of mining association rules can be decomposed into two subproblems. The first problem is to find out all *large itemsets* which are *contained* by a significant number of transactions with respect to a threshold *minimum support.* The second problem is to generate all the association rules from the large itemsets found, with respect to another threshold, the *minimum confidence.* Since it is easy to generate association rules if the large itemsets are available, major efforts in the research community have been focused on finding efficient algorithms to compute the large itemsets in recent studies.

Among all the algorithms proposed, the Apriori (and its modifications) [1] and the DHP (Direct Hashing and Pruning) [9] algorithms are the two most successful. They both run a number of iterations and compute the large itemsets of the same size in each iteration, starting from the size-one itemsets. In each iteration, they first construct a set of candidate itemsets and then scan the database to count the number of transactions that contain each candidate set. The key for optimization lies on the techniques used to create the candidate sets. The smaller the number of candidate sets is, the faster the algorithm would be.

The goal in this work is to solve the efficient update problem of association rules after a nontrivial number of new records have been added to a database. Assuming that the two thresholds, minimum support and confidence, do not change, there are several important characteristics in the update problem.

1. The update problem can be reduced to finding the new set of large itemsets. After that, the new association rules can be computed from the new large itemsets.

2. Generally speaking, an old large itemset has the potential to become small in the updated database.

3. Similarly, an old small itemset could become large in the new database.

4. In order to find the new large itemsets, all the records in the updated database, including those from the original database, have to be checked against every candidate set.

One possible approach to the update problem is to re-run the association rule mining algorithm on the whole updated database. This approach, though simple, has some obvious disadvantages. All the computation done initially at finding out the old large itemsets are wasted and all large itemsets have to be computed again from scratch.

In this paper, an efficient algorithm FUP (stands for F*ast* Up*date*) is presented for computing the large itemsets in the updated database. We will show that the information from the old large itemsets can be reused. Moreover, at finding the new large itemsets, the pool of candidate sets can be pruned substantially. Some optimization technique for reducing the database size during the update process will also be discussed.

Extensive experiments have been conducted to study the performance of FUP and compare it against the cases in which either Apriori or DHP is applied to the updated database to find the new large itemsets. FUP is found to be 2 to 16 times faster than re-running Apriori or DHP. More importantly, the number of candidate sets is found to be about 2-5 % of that in DHP. This shows that FUP is very effective in reducing the number of candidate sets. Also, the overhead of running FUP on an updated database is measured, and found to be only about 5-20% (which is very efficient).

The remaining of the paper is organized as follows. A detailed problem description is given in Section 2. The algorithm FUP is described in Section 3. Performance study is discussed in Section 4. Section 5 discusses the variations of the techniques, and Section 6 concludes our study.

2 Problem Description

2.1 Mining of association rules

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called *items*. Let DB be a database of transactions, where each transaction T is a set of items such that $T \subseteq I$. Given an *itemset* $X \subseteq I$, a transaction T *contains* X if and only if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \emptyset$. 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 *minconf* and a minimum support threshold *minsup*, the problem of mining association rules is to find out all the association rules whose confidence and support are larger than the respective thresholds. We also call an association rule a *strong* rule to distinguish it from the *weak* ones, i.e., those that do not meet the thresholds [6]. For an itemset X, its *support* is definited similarly as the percentage of transactions in DB which contain X. Given a minimum support threshold *minsup*, an itemset X is *large* if its support is no less than *minsup*. The problem of mining association rules is reduced to the problem of finding all large itemsets for a pre-determined minimum support [1].

2.2 Update of association rules

Let L 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 $X \in L$, its *support count*, $X.support$, which is the number of transactions in DB containing X, is available.

After some update activities, an increment db of new transactions is added to the original database DB, and d is 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., $X.support \geq s \times (D + d)$.

Thus the essence of the problem of *updating association rules* is to find the set L' of large itemsets in $DB \cup db$. Note that a large itemset in L may not be a large itemset in L', on the other hand, an itemset X not in L, may become a large itemset in L'.

3 Fast Update Algorithm FUP

Basically, the framework of FUP is similar to that of Apriori and DHP. It contains a number of iterations. The iteration starts at the size-one itemsets, and at each iteration, all the large itemsets of the same size are found. Moreover, the candidate sets at each iteration are generated based on the large itemsets found at the previous iteration. The features of FUP which distinguish it from Apriori and DHP are listed as follows.

1. At each iteration, the supports of the size-k large itemsets in L are updated against the increment db to filter out the *losers*, i.e., those that are no longer large in the updated database. Only the increment db has to be scanned to do the filtering.

2. While scanning the increment, a set of candidate sets, C_k, is extracted from the transactions in db, together with their supports in db counted. (Note that the size of db is in general much smaller than that of the original database DB.) The supports of these sets in C_k are then updated against the DB to find the "new" large itemsets.

3. More importantly, many sets in C_k can be pruned away by a simple check on their supports in db before the update against DB starts. (This check will be discussed in the following.)

4. The size of the updated database is reduced at each iteration by pruning away some items from some transactions in the updated database.

These features combined together form the core in the design of FUP and make FUP a much faster algorithm in comparison with the re-running of Apriori and DHP on the updated database. Our experimental results show a factor of 2 to 16 improvement in performance in the comparison.

The following notations are used in the remaining of the paper. L_k is the set of all size-k large itemsets (called *large k-itemsets*) in DB, and L'_k is the set of all large k-itemsets in $DB \cup db$. C_k is the set of size-k candidate sets in the k-th iteration of FUP. Moreover, $X.support_D$, $X.support_d$ and $X.support_{UD}$ represent the support counts of an itemset X in DB, db and $DB \cup db$, respectively. The following is a detailed description of the algorithm FUP. The first iteration of FUP is discussed followed by the discussion of the remaining iterations.

3.1 First iteration: Removing size-one losers, generating size-one candidate sets, and finding size-one winners

The following properties are useful in the derivation of the large 1-itemsets for the updated database.

Lemma 1 *An 1-itemset X in the original large 1-itemsets L_1 is a* loser *in the updated database $DB \cup db$ (i.e., not in the large 1-itemset L'_1) if and only if $X.support_{UD} < s \times (D + d)$.*

Proof. Based on the definitions of *minimum support* and *large 1-itemset.* □

Lemma 2 *An 1-itemset X not in the original large 1-itemsets L_1 can become a* winner *in the updated database $DB \cup db$ (i.e., being included in the large 1-itemset L'_1) only if $X.support_d \geq s \times d$.*

Proof. Since X is not in the original large 1-itemsets L_1, $X.support_D < s \times D$. If $X.support_d < s \times d$, then $X.support_{UD} = X.support_D + X.support_d < s \times (D + d)$. That is, X cannot become a large item in the updated database. Thus we have the lemma. □

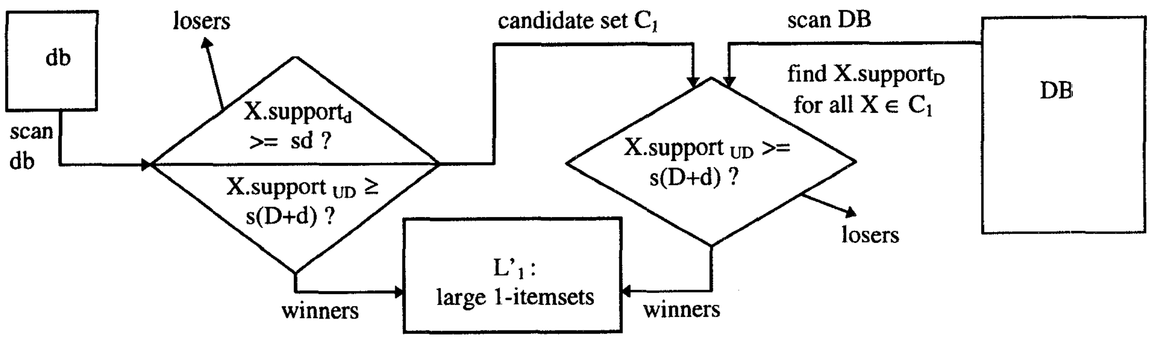

Figure 1: Processes in the first iteration of FUP.

Based on these properties, the finding of large 1-itemset L'_1 in the updated database $DB \cup db$ is outlined as follows. (The steps in the outline are described graphically in Figure 1 for easy understanding.)

1. Scan the increment *db*, for all itemsets $X \in L_1$, update its support count $X.support_{UD}$. Once the scan is completed, all the losers in L_1 are found by checking the condition $X.support_{UD} < s \times (D + d)$ on all $X \in L_1$ (according to Lemma 1.) By removing the losers, the itemsets in L_1 which remain large after the update are identified.

2. In the same scan, a set C_1 is created to store, for each $T \in db$, all size-one itemset $X \subseteq T$ which is not in L_1. This becomes the set of candidate sets and their support in *db* can also be found in the scan. More importantly, according to Lemma 2, if $X \in C_1$ and $X.support_d < s \times d$, X can never be large in $DB \cup db$. Because of this, all the sets in C_1, whose support counts are less than $s \times d$, are pruned off. This gives us a very small candidate set for finding the new size-one large itemsets.

3. A scan is then conducted on DB to update the support count $X.support_{UD}$ for each $X \in C_1$. By checking their support count, new large itemsets from C_1 are found. By combining with those identified in L_1, the set of all size-one large itemsets, L'_1, is generated.

Example 1 A database DB is updated with an increment *db* such that $D = 1000$, $d = 100$ and $s = 3\%$. I_1, I_2, I_3, and I_4 are four items. I_1 and I_2 are the large itemsets in L_1 with $I_1.support_D = 32$, and $I_2.support_D = 31$.

Assume that $I_1.support_d = 4$ and $I_2.support_d = 1$. After a scan on *db*, we have $I_1.support_{UD} = 36 > 1100 \times 3\%$ and $I_2.support_{UD} = 32 < 1100 \times 3\%$. Hence I_2 is a loser, and only I_1 is included in L'_1 (i.e., remains to be large in the updated database.)

Assume that I_3 and I_4 are two itemsets which are not in L_1 but occur in the increment *db*. Both I_3 and I_4 are potential candidate sets. In the scan of *db*, it is found that $I_3.support_d = 6$ and $I_4.support_d = 2$. Since $I_4.support_d < s \times d = 3\% \times 100$, it is removed from the candidate set C_1 (i.e., it is unnecessary to check I_4 against the updated database.) Only I_3 is included in C_1. Suppose that $I_3.support_D = 28$ is obtained in the scan of DB. Thus, $I_3.support_{UD} = 34 > 1100 \times 3\%$, and I_3 is included in L'_1. □

In comparison with the first iteration of Apriori and DHP, FUP first filters out the losers and obtains the first set of winners from the original large 1-itemsets by examining only the incremental database *db*. It also filters out from the remaining candidate set in *db* those items whose occurrence frequencies are too small to be considered as potential winners. Both functions are performed in a single scan of the incremental database *db*. It then scans the original database DB once to check the remaining potential winners. In contrast, Apriori and DHP must take all the data items as size-one candidate sets and check them against the whole updated database. A much smaller candidate set gives FUP a competitive edge in performance when compared with Apriori and DHP.

3.2 Second iteration and beyond: Removing other losers, pruning candidate sets, and finding remaining winners

The following properties are useful in the derivation of the large k-itemsets (where $k > 1$) for the updated database.

Lemma 3 *If* $\{X_1, \ldots, X_{k-1}\}$ *is a* loser *at the* $(k-1)$*-th iteration (i.e., the itemset is in* L_{k-1} *but not in* L'_{k-1}*), a large k-itemset in* L_k *(for any k) containing*

the itemset cannot be a winner in the k-th iteration (i.e., being included in the large k-itemset L'_k).
Proof. This is based on the property that *all the subsets of a large itemset must also be large*, proved in [2]. □

Lemma 4 *A k-itemset $\{X_1, \ldots, X_k\}$ in the original large k-itemsets L_k is a* loser *(i.e., not in the large k-itemset L'_k) in the updated database $DB \cup db$ if and only if $\{X_1, \ldots, X_k\}.support_{UD} < s \times (D + d)$.*
Proof. Based on the definitions of *minimum support* and *large k-itemset.* □

Lemma 5 *A k-itemset $\{X_1, \ldots, X_k\}$ not in the original large k-itemsets L_k can become a* winner *(i.e., being included in the large k-itemset L'_k) in the updated database $DB \cup db$ only if $\{X_1, \ldots, X_k\}.support_d \geq s \times d$.*
Proof. Based on the similar reasoning as for Lemma 2. □

Based on the above properties, the finding of large 2-itemset L'_2 in the updated database $DB \cup db$ is outlined as follows.

1. Similar to the first iteration, losers in L_2 will be filtered out in a scan on *db*. The filtering is done in two steps. Firstly, according to Lemma 3, some losers in L_2 can be filtered out without checking them against *db*. The set of losers $L_1 - L'_1$ have been identified in the first iteration. Therefore, any set $X \in L_2$, which has a subset Y such that $Y \in L_1 - L'_1$, cannot be large and are filtered out from L_2 without checking against *db*. Secondly, a scan is done on *db* and the support count of the remaining sets in L_2 are updated and the large itemsets from L_2 are identified.

2. Similar to the first iteration, the second part at this iteration is to find the new size-two large itemsets. The key is to generate a small set of candidate sets. The set of candidate sets, C_2, is generated, before the above scan on *db* starts, by applying the apriori-gen function on L'_1 [2]. The sets in L_2 are excluded when creating C_2 because they have already been handled. The support count of the itemsets in C_2 are accumulated in the same scan of *db*. The itemsets in C_2 can now be pruned by checking their support count. For all $X \in C_2$, if $X.support_d < s \times d$, X is removed from C_2. Based on Lemmas 5, all the removed sets cannot be large in $DB \cup db$.

3. The last step is to scan DB to update the support count for all the itemsets in C_2. At the end of the scan, all the sets $X \in C_2$, whose support count $X.support_{UD} \geq s \times (D + d)$, are identified as the new large itemsets. The set L'_2, which contains all the large itemsets identified from L_2 and C_2 above, are the set of all the size-two large itemsets.

Example 2 A database DB is updated with an increment db such that $D = 1000$, $d = 100$ and $s = 3\%$. I_1, I_2, I_3, and I_4 are four items and the size-1 and size-2 large itemsets in DB are $L_1 = \{I_1, I_2, I_3\}$ and $L_2 = \{I_1I_2, I_2I_3\}$, respectively. Also $I_1I_2.support_D = 50$ and $I_2I_3.support_D = 31$. Suppose FUP has completed the first iteration and found the "new" size-1 itemsets $L'_1 = \{I_1, I_2, I_4\}$. This example illustrates how FUP will find out L'_2 in the second iteration.

FUP first filters out losers from L_2. Note that $I_3 \in L_1 - L'_1$, therefore, the set $I_2I_3 \in L_2$ is a loser and is filtered out. For the remaining set $I_1I_2 \in L_2$, FUP scans *db* to update its support count. Assume that $I_1I_2.support_{db} = 3$. Since $I_1I_2.support_{UB} = (3+50) > 3\% \times 1100$, therefore, I_1I_2 is large in $DB \cup db$ and is stored in L'_2.

Secondly, FUP will try to find out the "new" large itemsets from *db*. Note that apriori-gen applied on L'_1 generates the candidate set $C_2 = \{I_1I_2, I_1I_4, I_2I_4\}$. Since $I_1I_2 \in L_2$ has already been handled, it is removed from C_2. For the remaining sets I_1I_4 and I_2I_4 in C_2, FUP scans *db* to update their support counts. Suppose $I_1I_4.support_d = 5$ and $I_2I_4.support_d = 2$. Since $I_2I_4.support_d = 2 < 3\% \times 100$, it cannot be a large itemsets in $DB \cup db$. Therefore, I_2I_4 is removed from the candidate set C_2.

For the remaining set $I_1I_4 \in C_2$, FUP scans DB to update its support count. Suppose $I_1I_4.support_D = 30$. Since $I_1I_4.support_{UD} = 30+5 > 3\% \times 1100$, it is a large itemset in the updated database. Therefore I_1I_4 is added into L'_2. At the end of the second iteration, $L'_2 = \{I_1I_2, I_1I_4\}$ is returned. □

The same algorithm is applied to the later iterations until no large itemsets is found. At the k-th iteration of FUP, the whole updated database is scanned once. However, for the large $(k-1)$-th itemsets in the original database, they only have to be checked against the small increment *db*. For the new large itemsets, their candidate sets are extracted from the increment and are pruned according to their support count in the increment. This pool for candidate sets is much smaller than those found by using either Apriori or DHP on the updated database. This shows that FUP is a much faster algorithm than the previous rule mining algorithms on database updates.

3.3 The FUP Algorithm

Based on the above discussion, the FUP algorithm is presented as follows.

Algorithm 1 FUP: A fast update algorithm for maintenance of association rules on database updates.

Input: (1) DB: the original database (with its size, i.e., the total number of transactions, equal to D); (2) L_k: the set of all large k-itemsets in DB, where $k = 1, \ldots, r$; (3) db: an increment database (with its size equal to d); and (4) s: the minimum support threshold.

Output: L': The set of all large itemsets in $DB \cup db$.

Method:

```
The 1st iteration: /* find $L'_1$, the set of all large 1-
    itemsets in $DB \cup db$ */

    $W = L_1$; $C = \emptyset$; $L'_1 = \emptyset$; $P = \emptyset$;
    /* $W$: winners, $C$: candidate sets,
    $L'_1$: initialized, $P$: for optimization */
    for_all $T \in db$ do /* scan $db$ */
      for_all 1-itemset $X \subseteq T$ do {
        if $X \in W$ then $X.support_d$++;
        else {
          if $X \notin C$
          then { $C = C \cup \{X\}$; $X.support_d = 0$; }
          /*init the support count and add $X$ into $C$ */
          $X.support_d$++; }
      };
    for_all $X \in W$ do /*put winners into $L'_1$ */
      if $X.support_{UD} \geq s \times (D + d)$
        then $L'_1 = L'_1 \cup \{X\}$;
    for_all $X \in C$ do /*prune candidate sets in $C$ */
      if $X.support_d < s \times d$
        then { $C = C - \{X\}$; $P = P \cup \{X\}$; }
        /* $P$ will be used for optimization. */
    for_all $T \in DB$ do /* scan $DB$ */
      for_all 1-itemset $X \subseteq T$ do {
        if $X \in C$ then $X.support_D$++;
        if $X \in P$ then removes $X$ from $T$;
        /* Transaction $T$ is reduced */
      };
    for_all $X \in C$ do /*put winners into $L'_1$*/
      if $X.support_{UD} \geq s \times (D + d)$
        then $L'_1 = L'_1 \cup \{X\}$;
    return $L'_1$. /* end of the 1st iteration */

The k-th iteration: /* for $k = 2$ or larger, repeat this
    program fragment to find $L'_k$, the set of all large
    k-itemsets in the updated database, until either
    $L'_k$ returned is empty or $db = \emptyset$ */

    $W = L_k$; $L'_k = \emptyset$ ;
     /* $W$: winners; $L'_k$ initialized */
    $C$ = apriori-gen($L'_{k-1}$) $-L_k$;
     /* the size-$k$ candidate sets */
    for_all k-itemset $X \in W$ do
     /* prune off losers in $W$ */
     for_all (k-1)-itemset $Y \in L_{k-1} - L'_{k-1}$ do
      if $Y \subseteq X$ then { $W = W - \{X\}$; break; }
    for_all $T \in db$ do { /* scan $db$ */
     for_all $X \in$ Subset($W, T$) do $X.support_d$++;
      /* Subset($W, T$) returns all the sets in $W$
      contained in $T$ [2] */
     for_all $X \in$ Subset($C, T$) do $X.support_d$++;
      /* find support of all $X \in C$ */
     Reduce_db(T);
      /*Some items in transactions in $db$ can
      be removed, discussed in next section*/
    }
    for_all $X \in W$ do
     /*put the winners from $W$ into $L'_k$ */
     if $X.support_{UD} \geq s \times (D + d)$
      then $L'_k = L'_k \cup \{X\}$;
    for_all $X \in C$ do /* prune candidate sets in $C$ */
     if $X.support_d < s \times d$ then $C = C - \{X\}$;
    for_all $T \in DB$ do { /* scan $DB$ */
     for_all $X \in$ Subset($C, T$) do $X.support_D$++;
     Reduce_Db(T); }
      /* Some items in transactions in $DB$ can
      be removed, discussed in next section */
    for_all $X \in C$ do
     /* put the winners from $C$ into $L'_k$ */
     if $X.support_{UD} \geq s \times (D + d)$
      then $L'_k = L'_k \cup \{X\}$;
    return $L'_k$. /* The end of the k-th iteration */
```

Rationale: The algorithm follows the lemmas and discussions in Sections 3.1-3.2. Moreover, the statements for the reduction of the size of the database are reasoned at the next subsection. Hence the algorithm correctly finds all the association rules in the updated database and terminates. □

3.4 Reduction of the size of the updated database

FUP applies the techniques used in DHP to reduce the size of the updated database. At the first iteration, all the candidate sets which do not have enough support in the increment db are stored in the set P. Later, during the scan of the original database, all items in P can be removed from all the transactions, because they will not appear in any large itemset in the later iteration.

At any k-th iteration, some items in *db* or *DB*, which is not needed for finding large itemsets in the next iteration, can be identified and hence removed.

At any k-th iteration, during the scan in the increment *db*, while FUP is counting the support for sets in the candidate sets C and W, for each transaction T, the *Reduce_db* function is called. It counts, for each $I \in T$, the number of sets in C and W which contain I. This number gives an upper bound on the number of large k-itemsets that contain I. If this number is smaller than k, then I cannot belong to any large (k+1)-itemset, and hence can be removed from all the transactions. Using this number, *Reduce_db* can prune off some items from *db*.

After the set C has been pruned against *db*, it can be seen that any items in DB which does not belong to any set in L_k or C will not belong to any large $k+1$-itemset. Therefore, in the scanning of DB to compute the supports of sets in C, all items that do not belong to any set in L_k or C can be removed. In the FUP algorithm, the function *Reduce_DB* performs this reduction.

In FUP, we have also integrated the direct hashing technique in [9], which further reduces the number of the candidate sets used in iteration two.

4 Performance Study

In order to assess the performance of FUP, experiments are conducted to compare its performance with that of Apriori and DHP. The experiments were performed on an AIX system on an RS/6000 workstation with model 410. As will be presented in the following, the result shows that FUP is much faster than the most successful mining algorithm with respect to updating association rules. FUP performs 2 to 6 times faster than DHP for a moderate size database of 100,000 transactions. When the database is scaled up to 1,000,000 transactions, the speed-up is 2 to 16 times. As explained before, the key of the speed-up lies on the much smaller amount of candidate sets. In some cases, the number of candidate sets generated were counted, and it was found that the amount generated in FUP is reduced to the range of 1.5 - 5% of that in DHP. This is a very significant reduction.

As mentioned above, we also tested FUP with some very large databases. It was found that FUP actually performs much better in larger databases.

4.1 Generation of synthetic data

The databases used in our experiments are synthetic data generated using the same technique introduced in [1] and modified in [9]. The parameters used

D	Number of transactions in database DB
d	Number of transactions in the increment d
| 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.

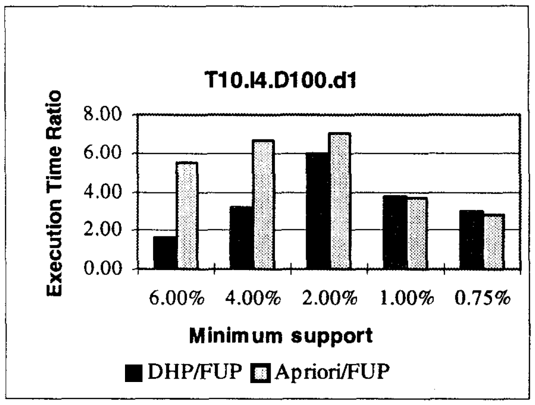

Figure 2: Performance Ratio.

are similar to those in [9] except that the size of the increment is an additional parameter. Table 1 is a list of the parameters used in our synthetic database.

In the following we use the notation Tx.Iy.Dm.dn, modified from the one used in [9], to denote a database in which $D = m$ thousands, $d = n$ thousands, |T| = x, and |I| = y. In our experiments, we set |L| = 2000, N = 1000, and the secondary parameters $S_q = 5$, $P_s = 50$, and $M_f = 2000$. S_q is the clustering size used in the generation of potential large itemsets. P_s is the pool size to store potential large itemsets from which transactions will receive their items. M_f is the multiplying factor associated with the pool. Readers not familiar with these parameters please refer to [1, 9].

The way we create our increment is a straight forward extension of the technique used to synthesize the database. In order to do comparison on a database of size D with an increment of size d. A database of size $(D + d)$ is first generated and then the first D transactions are stored in the database DB and the remaining d transactions is stored in the increment *db*. Since all the transactions are generated from the same statistical pattern, it models very well real life

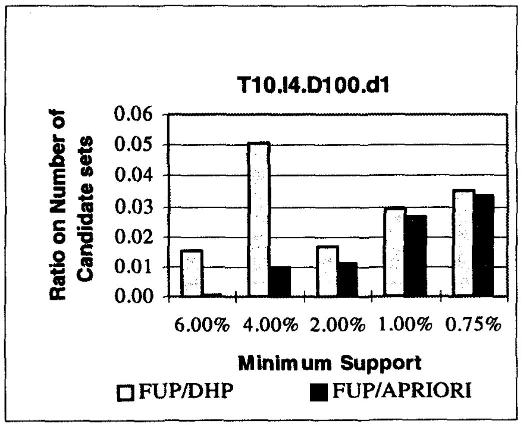

Figure 3: Reduction on Candidate Sets.

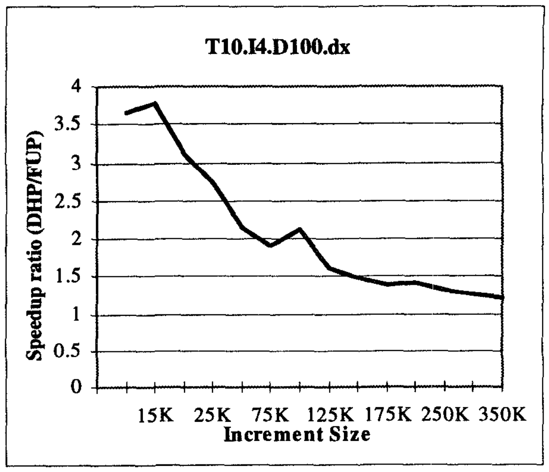

Figure 4: Speed Up Ratio vs Increment Size.

updates.

4.2 FUP versus DHP and Apriori

We have compared the performance of FUP against that of DHP and Apriori. The first comparison was done on an updated database T10.I4.D100.d1. The performance ratios between them are shown in Figure 2. In our implementation of the DHP, a hash table of size 100 is used, and hashing is only used in the generation of the size-2 candidate sets. This is the same policy used in [9]. For small support, FUP is 3 to 6 times faster than DHP, and 3 to 7 times faster than Apriori. For larger support, it is less costly to re-run the mining algorithm on the updated database since the number of large itemsets is relatively smaller. Interestingly, FUP is still 2 to 3 times faster in this case.

4.3 Reduction on the number of candidate sets

As explained before, FUP substantially reduces the number of candidate sets generated. The effect is particularly significant at the first iteration. In Figure 3, the chart shows the ratio of the number of candidate sets generated by FUP when comparing with the two mining algorithms. The amount of reduction ranges from 98% to 95% when FUP is compared to DHP. It is even greater when it is compared with Apriori.

4.4 Performance of FUP with large increment

In general, the larger the increment is, the longer it would take to do the update. Also, the gain in speed-up would slow down. Two sets of experiments have been performed to support this analysis. A database T10.I4.D100.dm with updates of 1K, 5K and 10K were generated, and different updates with different supports were done by FUP and DHP. For the same support, the speed-up ratio decreases when update size increases. For example, when the support is 2%, the ratio decreases from 5.8 to 3.7.

We also want to find out whether the decreasing of the performance ratio as the size increases in the update would eventually bring the performance of FUP down to that of DHP. In the same setting of T10.I4.D100.dm, we increase the increment size m from 10K gradually to 350K for comparison. The performance ratio is plotted in Figure 4. A gradually level off only appears when the increment size is about 3.5 times the size of the original database. The fact that FUP still exhibits performance gain when the increment is much larger than the original database shows that it is very efficient.

4.5 Small overhead of FUP

We also have done some experiments for the purpose of analyzing the overhead incurred by the FUP. In general, if the time to compute the set L' from an updated database $DB \cup db$ is added to the time to compute the original set L of large itemsets from the database DB by a mining algorithm, the sum would be larger than that if the same mining algorithm was applied directly on $DB \cup db$ to compute L'. The difference of these two time values is a measurement of the overhead of the update. If the overhead is small, then it indicates that the update was done very efficiently.

We have designed some experiments to analyze the overhead of FUP by measuring this difference. It was found that the bigger the increment is, the smaller this overhead becomes. In our experiment, what was discovered is that, when the increment is much smaller than the original database, the overhead percentage ranges around 10 – 15%. Once the increment is larger than the original size, the overhead decreases very rapidly from 10 to 5%. This is a very encouraging result because it shows that FUP not only can benefit update with small increment, it actually works very well in the case of large increment.

4.6 Performance in scaled-up databases

Our last experiment is done in a scaled-up database. The database is T10.I4.D1000.d10 which contains 1 million transactions. The performance ratio between DHP and FUP in this scaled-up database, ranges from 3 to 16. The result shows that the gain from FUP will in fact increase if the database becomes larger. This shows that FUP is very adaptive to size increase and can be applied to very large databases.

5 Discussion and Conclusions

We studied an efficient, fast, incremental updating technique for maintenance of the association rules discovered by database mining. The developed method strives to determine the promising itemsets and hopeless itemsets in the incremental portion and reduce the size of the candidate set to be searched against the original large database. The method is implemented and its performance is studied and compared with the best algorithms for mining association rules studied so far. The study shows that the proposed incremental updating technique has superior performance on database updates in comparison with direct mining from an updated database.

The incremental updating technique is applicable to the databases which allow frequent or occasional updates when new transaction data are added to a transaction database. We have also investigated the cases of deletion and modification of a transaction database.

Recently, there have been some interesting studies at finding multiple-level or generalized association rules in large transaction databases [6, 11]. The extension of our incremental updating technique for maintenance of multiple-level or generalized association rules in transaction databases is an interesting topic for future research.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, 207–216, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[3] D.W. Cheung, A. W.-C. Fu, and J. Han. Knowledge discovery in databases: A rule-based attribute-oriented approach. In *Proc. 1994 Int'l Symp. on Methodologies for Intelligent Systems*, pages 164–173, Charlotte, North Carolina, October 1994.

[4] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1995.

[5] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. Knowledge and Data Engineering*, 5:29–40, 1993.

[6] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, Zurich, Switzerland, Sept. 1995.

[7] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. 3rd Int'l Conf. on Information and Knowledge Management*, pages 401–408, Gaithersburg, Maryland, Nov. 1994.

[8] R. Ng and J. Han. Efficient and effective clustering method for spatial data mining. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 144–155, Santiago, Chile, September 1994.

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

[10] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases.* AAAI/MIT Press, 1991.

[11] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Int. Conf. Very Large Data Bases*, Zurich, Switzerland, Sept. 1995.