Maintenance of Discovered Knowledge : A Case in Multi-level Association Rules *

David W. Cheung† Vincent T. Ng‡ Benjamin W. Tam †

† Department of Computer Science, The University of Hong Kong, Hong Kong. Email: {dcheung|wktam}@cs.hku.hk.

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

Abstract

Many knowledge discovery (kdd) systems need to spend substantial amount of effort to search for rules and patterns within large amount of data. After some natural evolutions, as a consequence of updates applied to their databases, these systems must update their previously discovered knowledge to reflect the current state of their databases. The straight forward approach of re-running the discovery process on the whole updated database to re-discover the rules and patterns is not cost-effective in general, and is unacceptable in many cases. We have studied the problem of updating discovered association rules and found that it is nontrivial, because updates may not only invalidate some existing strong association rules but also turn some weak rules into strong ones. An incremental technique and a fast algorithm FUP have been proposed previously for the update of discovered single-level association rules. In this study, a more efficient algorithm FUP, which generates a smaller number of candidate sets when comparing with FUP, has been proposed. In addition, we have demonstrated that the incremental technique in FUP and FUP* can be generalized to some other kdd systems. An efficient algorithm MLUp has been proposed for this purpose for the updating of discovered multi-level association rules. Our performance study shows that MLUp has a superior performance over the representative mining algorithm such as ML-T2 in updating discovered multi-level association rules.*

1 Introduction

For some applications in data mining, discovering knowledge (strong rules or regular patterns) from a large amount of data could be very costly, because of the inherent need of searching through the whole database. If the application requires the knowledge discovered to reflect the current state of an updated database, then a new dimension of the discovery problem, the maintenance of the discovered knowledge, must be considered. This poses a challenge to many knowledge discovery (kdd) systems. The problem of maintenance in data mining is to discover efficiently the new set of knowledge given that the database has been updated. In general, part of the knowledge discovered previously could become invalid, and the remaining part remains valid. Moreover, new knowledge not exist in the discovered knowledge could also be discovered in the updated database. In fact, finding the "new" knowledge is the main difficulty in the update problem, because the system potentially has to search the whole updated database for the "new" knowledge.

*The research of the first author was supported in part by RGC (the Hong Kong Research Grants Council) grant 338/065/0026.

In some kdd systems, knowledge are being discovered in an interactive mode. In this case, rules and patterns discovered can be updated regularly on demand by re-interacting with the systems. However, because of different application requirements, many other kdd systems discover rules and patterns in a non-interactive mode. Particularly, in some cases, the requirements would demand the discovery of all, or at least a large number of the rules and patterns, that satisfy certain conditions. For these systems, the discovery is inherently costly and cannot be repeated as frequently as in the interactive case for maintenance purpose. To solve the maintenance problem in these systems by re-applying the complete discovery process on the updated database to discover the new set of rules and patterns is not cost-effective and is unacceptable in many cases. An ideal kdd system should be *maintenance-efficient*. The criteria of a maintenance-efficient kdd system is: it has an efficient method to update the knowledge discovered in the previous mining activities by using the informations in those knowledge, and does not need to re-apply the discovery process on the updated database.

With respect to the problem of maintenance, for a given kdd system, the challenge is to design an efficient method to update the knowledge that it has discovered. If an efficient method exists, then the system is maintenance-efficient. However, if this cannot be achieved, then the goal should be redirected to produce an enhancement or adjustment of the original discovery process so that not only the original mining requirement can be satisfied, but also the knowledge discovered can be maintained efficiently. In order to be applicable, it is important to enhance a maintenance-inefficient system to a maintenance-efficient system. We believe that a kdd system, which is efficient in mining knowledge but not maintenance-efficient, does not have practical usage. Furthermore, there may be cases that the performance of the discovery process has to be compromised slightly in order that an acceptable mechanism for maintenance can be added to a system.

An association rule is a strong rule which implies certain association relationships among a set of objects in a database. Since finding interesting association rules in databases may disclose some useful patterns for decision support, marketing analysis, financial forecast, system fault prediction, and many other applications, it has attracted a lot of attention in recent data mining research [6]. Efficient mining of association rules in transaction and/or relational databases has been studied substantially [1, 2, 8, 10, 12, 7, 11, 13].

In our previous study, we have investigated the maintenance problem in single-level association rules discovery [4]. An efficient algorithm FUP (F*ast* Up*date*) has been proposed, which can incrementally update the association rules discovered, if updates to a database is restricted to insertions of new transactions. In this paper, we will report two progresses in our study of the maintenance problem in the mining of association rules.

1. A faster version FUP* of FUP has been proposed. The improvement of FUP* over FUP is in the candidate set generation procedure. In general, the procedure in FUP* generates significantly less candidate sets then that in FUP for the update of discovered single-level association rules. (Details will be discussed in Section 2.)
2. An algorithm MLUp (stands for M*ulti*-L*evel association rules* Up*date*) has been proposed for the update of the discovered multi-level association rules in relation databases [7]. MLUp is an adaptation of FUP* in the multi-level environment. Extensive experiments have been conducted to study the performance of MLUp and compare it against the representative discovery algorithm ML-T2 proposed in [7] for the mining of multi-level association rules. MLUp is found to be about 2-3 times faster than running ML-T2 on an updated database. More importantly, the number of candidate sets in MLUp is found to be about 2-3% of that in ML-T2. This shows that MLUp is very effective in reducing the number of candidate sets. The efficiency of MLUp found in this study is consistent with our finding in the case of single-level association rule update.

The success of the incremental updating technique used in FUP* (or FUP) and MLUp for single-level and multi-level association rules suggests that, potentially, the technique could be generalized to solve the update problem in some other kdd systems. The remaining of the paper is organized as follows. In Section 2, the problem of updating single-level association rules and the algorithm FUP are briefly reviewed. Subsequently, the faster version FUP* is proposed. In Section 3, the problem of updating multi-level association rules is discussed and the algorithm MLUp for the update of discovered multi-level association rules is discussed. In Section 4, an in-depth performance study of MLUp is presented. Section 5 is the discussion and conclusions.

2 Update of Discovered Single-level Association Rules

2.1 Problem definition

In this section, we will describe briefly the problem of updating the discovered association rules in a transaction database. For details, please see [4]. (The problem and the solution presented in this work can be generalized to relational database in a straight forward way). We will also describe the notion of mining association rules as background knowledge. For details, please refer to [2].

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. 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 [2].

After an initial discovery of all the association rules in a database DB, let L be the set of large itemsets found with respect to the minimum support s, and D be the number of transactions in DB. Also, as a side effect of the initial discovery, the *support count*, $X.support$, which is the number of transactions in DB containing X, for each X in L, is stored in the system.

After some update activities following the initial discovery, 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 problem of *updating discovered 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'.

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 the update algorithm 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.

2.2 The FUP algorithm

There are two basic tasks in updating the association rules. (1) Among all the itemsets in L, separate the *winners*, i.e., those that remain large in the updated database, from the *losers*, i.e., those that are no longer large in the updated database. This task is straight forward and can be done by updating the support counts of the itemsets in L against the increment db. (2) Discover the *new* winners, i.e., those that are not large in the original database DB, but become large in $DB \cup db$.

Potentially, all the itemsets not in L could become large in $DB \cup db$. Since the search domain is so large, it is tempting to infer that a re-run of the discovery process on the updated database is necessary in order to find all the new winners. However, our finding is that only a small number of the itemsets not in L have to be considered in the task of searching for the new winners. We summarize our findings in Lemma 1.

Lemma 1 [4] *A k-itemset X not in the original large k-itemsets L_k can become a* winner*, (i.e., become large) in the updated database $DB \cup db$ only if $X.support_d \geq s \times d$.* □

Lemma 1 is an important result which allows FUP to restrict its search for the new winners in a small set of candidate sets. Following the lemma, a new winner must be an itemset appeared in the increment db, whose support count in db must be at least as large as $s \times d$.

In the following, we will describe the main procedure Search_Winners of FUP. For a given k, the function of Search_Winners is to find out the set of size-k large itemsets L'_k in the updated database. To find the set of large itemsets L', FUP applies the procedure Search_Winners iteratively on the updated database, starting from the size-1 large itemsets until no large itemset can be found. (For a complete description of the FUP algorithm, please see [4]).

Search_Winners *procedure* :

1. Scan the increment db to complete two tasks. (1) Update the support count $X.support_{UD}$ for each $X \in L_k$, and then filter out the winners in L_k whose support count is larger than or equal to the support threshold. (2) Find all size-k itemsets X in db such that $X \notin L_k$ and its support count $X.support_d \geq s \times d$, and then store these itemsets in the set of candidate sets C_k.

2. Scan the original database DB to update the support count $X.support_{UD}$ for each itemset $X \in C_k$. Subsequently, filter out the new winners among the sets in C_k by comparing their support counts against the support threshold.

2.3 A faster update algorithm FUP*

The improvement of FUP* over FUP is in the candidate set generation mechanism. In task (2) of the first step in the Search_Winners procedure, FUP uses the Apriori-gen function defined in [2] to establish a set of candidate sets [4]. In fact, it looks for itemsets in Apriori-gen(L'_{k-1}) which does not belong to L_k but appear in some transaction(s) in db, whose support count in db is larger than or equal to $s \times d$, where the set L'_{k-1} is the set of size-(k-1) large itemset in the update database found in the (k-1)-th iteration of FUP. We find out that the domain in this searching which is the set Apriori-gen(L'_{k-1}) can be further reduced to a smaller set. This finding is supported by the following result.

Lemma 2 *A k-itemset X not in the original large k-itemsets L_k can become a* winner *(i.e., become large) in the updated database $DB \cup db$ only if $Y.support_d \geq s \times d$ for all the subsets $Y \subseteq X$.*

Proof. It follows from Lemma 1 that $X.support_d \geq s \times d$. If $Y \subseteq X$, then $Y.support_d \geq X.support_d$. Hence, the condition holds for all the subsets Y of X. $\square$

(A result similar to Lemma 2 for partitioned databases has been reported in [5]).

As discussed above, during the scan of the increment db, FUP looks for itemsets in db which appear in Apriori-gen(L'_{k-1}) and have a large enough support count in db. From Lemma 2, not only a new winner must have enough support count in db, but also all its subsets. In order words, the support counts of all the subsets of a candidate set must also be larger than or equal to $s \times d$. Therefore, the candidate sets can be restricted to the sets in Apriori-gen(L^*_{k-1}), where L^*_{k-1} are the itemsets in L'_{k-1}, whose support counts in db are larger than or equal to $s \times d$. In general, L^*_{k-1} is smaller than L'_{k-1}, and hence the number of candidate sets in Apriori-gen(L^*_{k-1}) is smaller than that in Apriori-gen(L'_{k-1}). Depending on the data distribution, the size of Apriori-gen(L^*_{k-1}) could be significantly smaller than that of Apriori-gen(L'_{k-1}).

Following from the above lemma and discussion, we enhance FUP to FUP* by refining the Search_Winners procedure. In task (2) of the first step in the Search_Winners procedure, when it is building up the candidate set C_k, FUP* restricts its search domain in the possibly smaller set Apriori-gen(L^*_{k-1}), instead of the set Apriori-gen(L'_{k-1}). With this optimization in the Search_Winners procedure, FUP* may have a much smaller number of candidate sets comparing with FUP, and hence become more efficient. The other steps of Search_Winners in FUP* is the same as those in FUP, and will not repeated here. In the following, we will use Example 1 to illustrate the execution of FUP*. In particularly, the example will show that FUP* can reduce significantly the number of candidate sets.

Example 1 A database DB is updated with an increment db such that $D = 1000$, $d = 100$ and $s = 3\%$. X, Y, Z, and W are four items and the size-1 and size-2 large itemsets in DB are $L_1 = \{X, Y, Z\}$ and $L_2 = \{XY, YZ\}$, respectively. Also $XY.support_D = 32$ and $YZ.support_D = 31$. Suppose FUP* has completed the first iteration and found the "new" size-1 itemsets $L'_1 = \{X, Y, W\}$. Moreover, assuming that the support counts of X, Y, and W found in db are 2, 4 and 5, respectively. This example illustrates how FUP* will find out L'_2 in the second iteration, and also its effectiveness in reducing the number of candidate sets.

FUP* first filters out losers from L_2. Note that $Z \in L_1 - L'_1$, i.e., Z has become a loser; therefore, the set $YZ \in L_2$ must also be a loser and is filtered out. For the remaining set $XY \in L_2$, FUP* scans db to update its support count. Assume that $XY.support_{db} = 2$. Since $XY.support_{UB} = (2 + 32) > 3\% \times 1100$, therefore, XY is large in $DB \cup db$ and is stored in L'_2.

Secondly, FUP* needs to find out the "new" large itemsets from db. For this purpose, FUP* has to find out the set L^*_1 from L'_1, which contains the itemsets in L'_1 that have enough support counts in db. Since $X.support_d = 2 < 3\% \times 100$, $X \notin L^*_1$, i.e., even though X is a winner in the 1st iteration, it will not be used to generate the size-2 candidate sets. On the other hand, both the support counts of Y and W in db are larger than the threshold $3\% \times 100$; therefore they will be used to generate the size-2 candidate sets, i.e., $L^*_1 = \{Y, W\}$. Following that, FUP* applies Apriori-gen on L^*_1 and generates the candidate set $C_2 = \{YW\}$. Note that in FUP, Apriori-gen is applied on $L'_1 = \{X, Y, W\}$ instead of L^*_1, and the set of candidate sets generated will have three itemsets which is three times larger than what is generated in FUP* in this example. This illustrates that FUP* can significantly and effectively reduce the number of candidate sets when comparing with FUP.

Suppose $YW.support_d = 4 > 3\% \times 100$. It follows from Lemma 1 that YW will not be pruned and remain in C_2. Following the pruning of the candidate sets in C_2, FUP* has to update the remaining candidate sets in C_2 against the original database DB. Suppose $YW.support_D = 29$. Since $YW.support_{UD} = 29 + 4 > 3\% \times 1100$, it is a large itemset in the updated database. Therefore YW is added into L'_2. At the end of the second iteration, $L'_2 = \{XY, YW\}$ is returned. $\square$

3 Update of Discovered Multi-level Association Rules

The method used in FUP* (and FUP) could be applied to many other kdd systems to update the knowledge discovered. In particular, it can be used in the systems that are designed to discover various types of associations between generalized items and events. This includes the discovery of multi-level association rules, generalized association rules, sequential patterns, episodes, and quantitative association rules [7, 12, 3, 9, 13]. In the following, we will show that FUP* can be generalized to solve the update problem for multi-level association rules. For this purpose, an algorithm MLUp (stands for M*ulti*-L*evel Association Rules* Up*date*), which is an adaptation of FUP*, will be proposed.

3.1 Mining of multi-level association rules

In the study of mining multi-level association rules, a series of algorithms have been proposed to facilitate a top-down, progressive deepening method based on the algorithms for mining single-level association rules. The method first finds large data items at the top-most level and then progressively deepens the mining process into their large descendants at lower concept levels. For details on the mining of multi-level association rules, please refer to [7].

A representative algorithm for mining multi-level association rules is the ML-T2 algorithm proposed in [7]. For discussion purpose, we will describe the main steps in ML-T2 in the following. (We will use $L[m, k]$ to denote the set of k-large itemsets in level m in order to follow the convention in [7]. Also, we will use $L'[m, k]$ to denote the set of k-large itemsets in level m in the updated database.)

ML-T2 (main steps) :

1. Translate the transaction database, denoted by T, into an encoded transaction table $T[1]$ according to a given taxonomy information.

2. At level 1, the large 1-itemsets $L[1, 1]$ is derived by scanning through $T[1]$. At any other level m, $(m > 1)$, $L[m, 1]$ is derived by scanning through $T[2]$, where $T[2]$ is a result of filtering the transactions in $T[1]$ by removing any item not in $L[1, 1]$.

3. The large k-itemsets, $(k > 1)$, at level m, $(m \geq 1)$, is derived in two steps. (1) Compute the candidate set C_k by applying Apriori-gen on $L[m, k-1]$. (2) Scan $T[2]$ to update the support counts of the candidate sets in C_k. Subsequently, find out all those candidates in C_k whose support count is larger than the threshold for level m.

4. At level m, return the union of $L[m, k]$ for all the k's.

3.2 Update of discovered multi-level association rules

The problem of updating the discovered multi-level association rules is the same as that in the single-level environment. The only difference is that the rules in all the levels have to be updated instead of updating the rules in only one level. Also, the minimum support thresholds at different levels may not be equal. We use s_m to denote the minimum support threshold at level m for $m \geq 1$.

Since there are several variations of the algorithm in mining multi-level association rules, the update algorithm should be designed according to the strategy used in the initial mining process. The algorithm MLUp we are proposing is associated with the representative mining algorithm ML-T2. The following two results are the bases of MLUp.

Lemma 3 *In a multi-level environment, a level-m 1-itemset X not in the original large 1-itemsets $L[m,1]$, ($m \geq 1$), can become a* winner *(i.e., become large) in the updated database $DB \cup db$ only if all ancestors of X are winners.*

Proof. This follows from the definition of large itemsets in the multi-level environment. □

Following Lemma 3, when MLUp scans the increment db to look for new size-1 winners, it not only has to ensure a candidate itemset has the required support count, but must also check that all its ancestors are large in the updated database. (Because of transitivity, MLUp only needs to check a candidate's immediate ancestor).

Lemma 4 *In a multi-level environment, a level-m k-itemset X not in the original large k-itemsets $L[m,k]$, ($m \geq 1$), can become a* winner *(i.e., become large) in the updated database $DB \cup db$ only if $X.support_d \geq s_m \times d$ and $Y.support_d \geq s_m \times d$, for all subset $Y \subseteq X$.*

Proof. This follows directly from Lemmas 1 and 2. □

The implication of the result in Lemma 4 is that the candidate set generation mechanism in FUP* can be applied directly in MLUp for finding new winners in different levels. In the following, we describe the main procedure of the update algorithm MLUp. The input to the algorithm includes the original encoded transaction database $T[1]$, the increment database db, and the old large itemsets $L[m,k]$, ($m \geq 1$, $k \geq 1$), and their support counts. Following the conventions in FUP*, the sizes of $T[1]$ and db are denoted by D and d respectively. Moreover, the minimum support threshold for different level is denoted by s_m, ($m \geq 1$).

MLUp (main steps) :

1. Translate the increment transaction database db into an encoded transaction table $db[1]$ according to the given taxonomy information.

2. At level 1, scan $db[1]$ to update the support counts of the 1-itemsets in $L[1,1]$ to filter out the winners into $L'[1,1]$. In the same scan, find all the 1-itemsets in $db[1]$ which do not belong to $L[1,1]$, whose support count in $db[1]$ is larger than or equal to $s_1 \times d$, and store these 1-itemsets in the candidate set C_1. Subsequently, scan $T[1]$ to find out the new winners in C_1 and store them into $L'[1,1]$. Following that, $T[1]$ is filtered by $L'[1,1]$ to generate the encoded transaction table $T[2]$. Similarly, $db[1]$ is filtered to $db[2]$.

   At level m, ($m > 1$), scan $db[2]$ to update the support counts of the 1-itemsets in $L[m,1]$. An 1-itemset in $L[m,1]$ is a winner only if its immediate ancestor is large in the updated database and its support counts in the updated database is larger then or equal to $s_m \times (D + d)$. In the same scan, find all level-m 1-itemsets in $db[2]$ which do not belong to $L[m,1]$, whose immediate ancestor belongs to $L'[m-1,1]$ and whose support count in $db[2]$ is larger than or equal to $s_m \times d$. Then store these 1-itemsets in the candidate set C_1. Subsequently, scan $T[2]$ to find out the new winners in C_1 and store them in $L'[m,1]$.

3. The large k-itemsets, ($k > 1$), for the updated database at level m is derived in three steps:

   (1) Remove all the k-itemsets in $L[m,k]$ for which one of its ancestors is not large in the updated database. Then scan $db[2]$ to update the support counts of the remaining itemsets in $L[m,k]$ to find out the winners.

   (2) Let $L^*[m,k-1]$ be the subsets of itemsets in $L'[m,k-1]$ whose support count in db is larger then or equal to $s_m \times d$. In the same scan on $db[2]$ performed in (1), find all level-m k-itemsets in Apriori-gen($L^*[m,k-1]$) which do not belong to $L[m,k]$, whose support count in $db[2]$ is larger then or equal to $s_m \times d$, and store them in the candidate set C_k.

(3) Scan DB to update the support counts of the candidate sets in C_k and find all the level-m size-k winners in C_k, and store them in $L'[m, k]$.

4. At level m, return the union of $L'[m, k]$ for all the k's.

4 Performance Study of MLUp

Extensive experiments have been conducted to assess the performance of MLUp. It was compared with the algorithm ML-T2. The experiments were performed on an AIX system on an RS/6000 workstation with model 410. The result shows that MLUp is much faster than re-running ML-T2 to update the discovered association rules. MLUp performs 2 to 3 times faster than ML-T2 on all levels in our experiments. Moreover, the number of candidate sets generated in MLUp is only about 2-3% of that in ML-T2. This improvement is not surprising given that FUP also has similar performance in updating single-level association rules. The databases used in our experiments are synthetic data generated using a technique similar to that in [2].

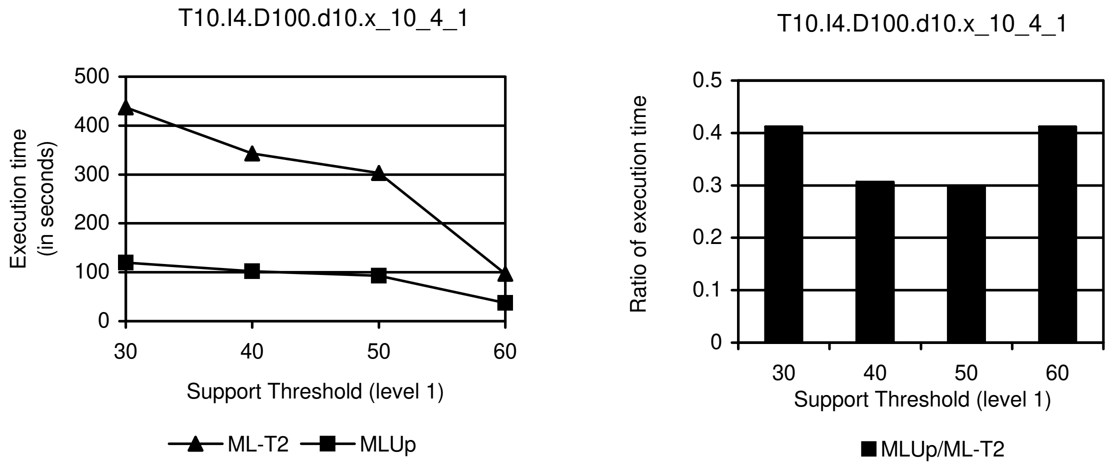

Figure 1: Performance Comparison (level 1)

Our test environments are denoted by T10.I4.D100.d10.s_1_s_2_s_3_s_4, which represents an updated database in which the original database DB has 100 thousands of transactions (D100), the increment db has 10 thousands of transactions (d10). The transactions on average has 10 items (T10), and the average size of the large itemsets is 4 (I4). Moreover, there are four levels in the taxonomy and the minimum supports are denoted by s_i, $(1 \leq i \leq 4)$. (Note that in the presentation of our performance study, we have followed the convention in [7] to use absolute numbers for minimum support thresholds instead of percentages).

The performance comparison between MLUp and ML-T2 in the update of the level-1 association rules is plotted in Figure 1 against different minimum support thresholds. Their performance ratios are also presented as bar charts in the same figure. It can be seen that MLUp is 2-3 times faster than ML-T2. MLUp also has similar speed-up over ML-T2 in the updates in the other levels.

As explained before, MLUp reduces substantially the number of candidate sets generated when comparing with ML-T2. In Figure 2, the number of candidate sets generated in MLUp in the same experiment is compared with that in ML-T2. The ratios in the comparison are presented as bar charts in the same figure. The chart shows that the number of candidate sets generated by MLUp is only about 2-3% of that in ML-T2.

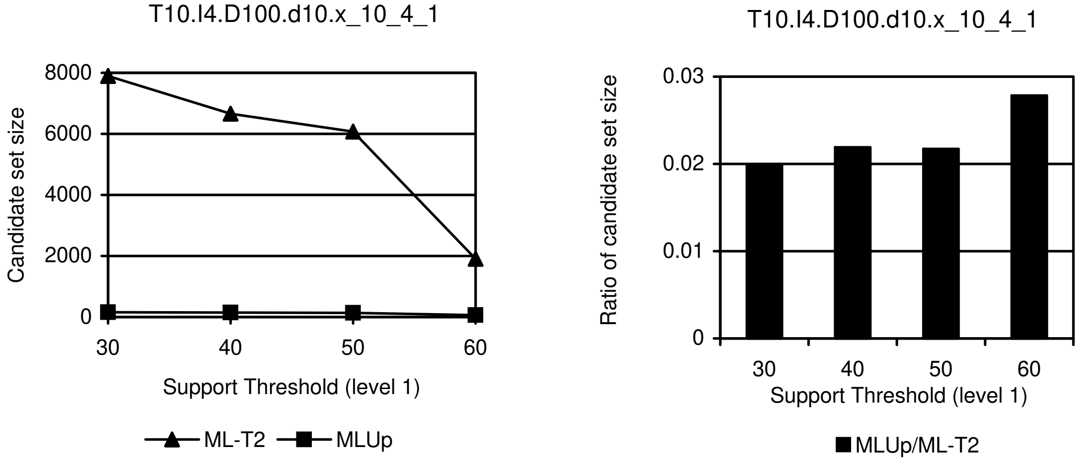

Figure 2: Reduction of Candidate Sets (level 1)

In general, the larger the increment is, the longer it would take to do the update. Also, the gain in the speed-up of MLUp would slow down. A series of updates from 10K to 350K were generated on the databases T10.I4.D100, and the execution times for MLUp and ML-T2 to do the updates on these increments were compared. A gradually level off of the speed-up of MLUp over ML-T2 only appears when the increment size is about 3.5 times the size of the original database. The fact that MLUp still exhibits performance gain when the increment is much larger than the original database shows that it is very efficient.

5 Discussion and Conclusions

A kdd system not only should be efficient in its regular mining activities but also efficient in maintaining the knowledge it has discovered. We have shown that FUP* is an efficient algorithm for updating discovered single-level association rules. It improves the performance of FUP by significantly reducing its candidate sets. By proposing the algorithm FUP* (or FUP), we have shown that mining single-level association rules is maintenance-efficient, if updates to a transaction database contain only insertions of new transactions.

We have also proposed an efficient algorithm MLUp for updating discovered multi-level association rules. It is an adaptation of the FUP* algorithm in the multi-level environment. The algorithm MLUp is implemented and its performance is studied and compared with the ML-T2 algorithm, which is a representative algorithm for mining multi-level association rules. The study shows that MLUp has superior performance in the multi-level environment. The success of the incremental updating technique in both the single-level and multi-level association rules suggests that the technique could be generalized to solve the update problems in some other knowledge discovery systems.

Currently, both FUP* and MLUp are applicable only to a database which allow frequent or occasional updates restricted to insertions of new transactions. We have also investigated the cases of updates including deletions and/or modifications to a transaction database. In FUP* and MLUp, the incremental updating technique has made use of the fact that new winners generated in the updating process must appear and have enough support counts in the increment. However, this does not hold in general in the cases of deletion and modification. For example, in the case of deletion, because the size of the updated database has decreased, some itemsets which are "small" in the orginial database DB, could become large in the updated database, even though it is not contained in any transaction deleted. Consequently, the set of candidate sets cannot be limited to those appear in the increment, and potentially, all itemsets in the updated database have to be considered as candidates. Therefore, the current incremental technique cannot be applied directly to the cases of deletion and modification. However, it is possible to solve the deletion and modification cases if the initial mining process is enhanced to retain more informations to support the update. For example, the initial process can be adjusted to record some useful informations so that the candidate sets for the update process can be limited to a particular sets of itemsets.

We believe that a kdd system should have good maintainability such that its mining process will retain enough informations to support efficient update of the discovered knowledge when the system evolves. This should be an important requirement in the design of kdd systems.

The extension of our incremental update technique for the maintenance of other type of knowledge such as generalized association rules, episodes, sequential patterns, and quantitative association rules is an interesting topic for future research. However, as discussed above, a bigger challenge is to extend this technique to cover the cases of deletion and modification.

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*, pages 207–216, Washington, D.C., 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] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering*, pages 3–14, Taipei, Taiwan, March 1995.

[4] D.W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 1996 Int'l Conf. on Data Engineering*, New Orleans, Louisiana, Feb. 1996.

[5] D.W. Cheung, J. Han, V. Ng, A. Fu and Y. Fu. A Fast Distributed Algorithm for Mining Association Rules. Technical Report, Dept. of Computer Science, The University of Hong Kong, 1996.

[6] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[7] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 420–431, Zurich, Switzerland, Sept. 1995.

[8] 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.

[9] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering Frequent Episodes in Sequences. In *Proc. 1st Int'l Conf. on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Quebec, Canada, Aug. 1995.

[10] 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*, pages 175–186, San Jose, CA, May 1995.

[11] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 432–443, Zurich, Switzerland, Sept. 1995.

[12] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 407–419, Zurich, Switzerland, Sept. 1995.

[13] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data*, Montreal, Canada, June 1996.